

Un guide d'apprentissage

Tête la première Design Patterns

Évitez les
erreurs de
couplage gênantes



Voyez pourquoi vos amis
se trompent au sujet du
pattern
Fabrication



Découvrez les
secrets du maître
des patterns



Injectez-vous
directement dans
le cerveau les
principaux patterns



Trouvez comment le
pattern Décorateur a fait
grimper le prix des
actions de Starbuzz Coffee



Apprenez comment
la vie amoureuse de
Jim s'est améliorée
depuis qu'il préfère
la composition à
l'héritage



Eric Freeman & Elisabeth Freeman
avec Kathy Sierra & Bert Bates

Traduction de Marie-Cécile Baland

DIGIT BOOKS

Propriété de Stéphane LARDIC, La Machine à Coudre, 10 rue de la République, 92100 Nanterre

Design patterns

Tête la première

édition mise à jour 2010

Design patterns

Tête la première



Ne serait-ce pas
merveilleux s'il y avait un livre
sur les Design Patterns qui soit plus
amusant qu'un rendez-vous chez le
dentiste et plus facile à comprendre
qu'une déclaration de revenus ? Mais
ce n'est sans doute qu'un
fantasme...

Eric Freeman
Elisabeth Freeman

avec
Kathy Sierra
Bert Bates

Traduction de Marie-Cécile Baland

DIGIT BOOKS

Design patterns - Tête la première

de Eric Freeman , Elisabeth Freeman, Kathy Sierra et Bert Bates

L'édition originale de cet ouvrage a été publiée aux États-Unis par O'Reilly Media sous le titre *Head First Design Patterns*

© 2004 O'Reilly Media

© 2009, 2010 Digit Books pour la traduction française

ISBN 978-2-8150-0003-1 PRIX 39 euros (PDF)

Éditrice :

Dominique Buraud

Couverture conçue par :

Ellie Volckhausen et Marcia Friedman

Spécialistes des patterns :

Eric Freeman, Elisabeth Freeman

Façade et décoration :

Elisabeth Freeman

Stratégie :

Kathy Sierra et Bert Bates

Observateur :

Oliver



Nombre de désignations utilisées par les fabricants et les fournisseurs sont des marques de commerciales déposées. Lorsque ces désignations apparaissent dans ce livre et qu'elles sont connues d'O'Reilly Media, Inc. comme étant des marques déposées, elles viennent en capitales ou avec une capitale à l'initiale.

Bien que tout le soin nécessaire, ait été apporté à la préparation de ce livre, l'éditeur et les auteurs n'assument aucune responsabilité des erreurs ou des omissions, ni d'éventuels dommages résultant de l'utilisation des informations qu'il contient. Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

Autrement dit, si vous utilisez un élément quelconque de *Design patterns tête la première* dans le développement d'un système dédié à une centrale nucléaire, vous le faites à vos risques et périls.

En revanche, nous vous encourageons vivement à utiliser l'application VueDJ.

Il n'a été fait de mal à aucun canard dans la rédaction de ce livre.

Les membres de la Bande des quatre ont consenti à la publication de leur photo. Oui, ils sont réellement aussi mignons que cela.

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

À la Bande des quatre, leur perspicacité et l'expertise dont ils ont fait preuve en collectant et en communiquant les *design patterns* ont changé pour toujours la face du développement logiciel et amélioré la vie des développeurs du monde entier.

Mais sérieusement, *quand* allons-nous voir une deuxième édition ? Après tout, cela ne fait jamais que *dix ans* !

Auteurs/développeurs de Design patterns tête la première

Elisabeth Freeman ↗



Elisabeth est auteur, développeur de logiciels et artiste numérique. Elle s'est impliquée depuis le début sur l'Internet, ayant été la cofondatrice de *The Ada Project* (TAP), un centre de ressources en ligne sur les femmes et l'informatique qui est maintenant un projet de l'ACM. Plus récemment, Elisabeth a coordonné des efforts de recherche et de développement dans le domaine des médias numériques pour la Walt Disney Company, où elle a été l'un des inventeurs de Motion, un système de gestion de contenu qui délivre quotidiennement des téraoctets de vidéo aux utilisateurs de Disney, ESPN et Movies.com.

Informaticienne dans l'âme, Elisabeth est diplômée de l'université Yale et de l'université d'Indiana. Elle a travaillé dans de nombreux domaines, notamment les langages de programmation visuels, la syndication RSS et les systèmes Internet. Elle a également développé des programmes encourageant les femmes à travailler dans le domaine de l'informatique.

Ces jours-ci, vous la trouverez en train de siroter du Java ou du Cocoa sur son Mac, bien qu'elle rêve du jour où le monde entier utilisera Scheme.

Elisabeth adore la nature et la randonnée depuis son enfance écossaise. Quand elle est dehors, son appareil photo n'est jamais loin. C'est également une cycliste passionnée, une végétarienne convaincue et une amie des animaux. Vous pouvez lui envoyer un message à beth@wickedlysmart.com

Eric est un informaticien qui se passionne pour les médias et les architectures logicielles. Il vient de terminer une mission de rêve qui a duré quatre ans – coordonner les efforts en matière d'Internet large bande et d'accès sans fil chez Disney – et se consacre maintenant de nouveau à l'écriture, à la création de logiciels et à travailler sur Java et sur les Macs.

Eric a passé une bonne partie des années 90 à travailler sur des alternatives à la métaphore du bureau avec David Gelernter (et ils se posent toujours tous deux la question « pourquoi suis-je obligé de donner un nom à un fichier ? »). C'est sur la base de ces recherches qu'il a soutenu sa thèse de doctorat à Yale University en 1997. Il a également cofondé Mirror Worlds Technologies pour diffuser une version commerciale de son travail, Lifestreams.

Dans une vie antérieure, Eric a développé des logiciels pour des réseaux et des supercalculateurs. Vous le connaissez peut-être également comme auteur d'ouvrages tels que *JavaSpaces Principles Patterns and Practice*. Il se souvient avec tendresse d'avoir implémenté des systèmes à espaces de tuples sur Thinking Machine CM-5 et d'avoir créé certains des premiers systèmes d'information en ligne pour la NASA à la fin des années 80.

Eric vit actuellement dans le semi-désert, près de Santa Fé. Quand il ne s'adonne ni à l'écriture ni à la programmation, vous le trouverez en train de passer plus de temps à bricoler son home cinéma qu'à le regarder, ou à essayer de restaurer un vieux jeu vidéo *Dragon's Lair* âgé d'au moins vingt ans. Il ne rechignerait pas non plus à passer ses nuits dans le rôle de DJ avec de la musique électronique.

Écrivez-lui à eric@wickedlysmart.com ou visitez son blog à <http://www.ericfreeman.com>

Créateurs de la collection Tête la première (et conspirateurs sur ce livre)

Kathy Sierra



Kathy s'intéresse aux théories de l'apprentissage depuis l'époque où elle était conceptrice de jeux (elle a écrit des programmes pour Virgin, MGM et Amblin). Elle a mis au point la majeure partie du concept de ce livre lorsqu'elle enseignait les nouveaux médias à l'université de Californie à Los Angeles. Plus récemment, elle a été formatrice de formateurs chez Sun Microsystems, où elle a enseigné aux instructeurs la façon de transmettre les dernières technologies et développé plusieurs examens de certification. Avec Bert Bates, elle a utilisé activement les concepts de Java tête la première pour former des milliers de développeurs. Elle a fondé le site javaranch.com, qui a obtenu en 2003 et 2004 le Jolt Cola Productivity Award du magazine *Software Development*. Elle enseigne également Java dans les croisières Geek Cruise (geekcruises.com).

Elle a récemment quitté la Californie pour vivre au Colorado. Elle a dû apprendre de nouveaux mots comme « antigel » et « cache-nez », mais la lumière y est fantastique.

Kathy aime courir, skier, faire du skateboard et jouer avec son cheval islandais. Elle s'intéresse aussi aux phénomènes inhabituels. Ce qu'elle déteste le plus : l'entropie.

Vous pouvez la trouver sur Javaranch, ou lire ses participations occasionnelles au blog de java.net. Écrivez-lui à kathy@wickedlysmart.com.



Bert Bates

Bert est architecte et développeur de logiciels, mais dix ans d'expérience dans le domaine de l'intelligence artificielle l'ont amené à s'intéresser aux théories de l'apprentissage et aux apports des nouvelles technologies à la formation. Depuis lors, il apprend à ses clients à mieux programmer. Récemment, il a dirigé chez Sun l'équipe de développement de plusieurs examens de certification Java.

Il a consacré la première décennie de sa carrière d'informaticien à aider des clients du secteur de la radiodiffusion, comme Radio New Zealand, Weather Channel et A & E (Arts & Entertainment Network). L'un des ses projets favoris a été de tout temps un programme de simulation complet pour l'Union Pacific Railroad.

Bert est de longue date un joueur de go désespérément accro et il travaille à un programme de go depuis bien trop longtemps. C'est un bon guitariste et il est en train de s'exercer au banjo.

Cherchez-le sur Javaranch, sur IGS (*Internet Go Server*) ou écrivez-lui à terrapi@wickedlysmart.com.

Table des matières (résumé)

	Intro	xxiii
1	Bienvenue aux Design Patterns : <i>introduction</i>	1
2	Tenez vos objets au courant : <i>le pattern Observateur</i>	37
3	Décorer les objets : <i>le pattern Décorateur</i>	79
4	Un peu de cuisine orientée objet : <i>les patterns fabriques</i>	109
5	Des objets uniques en leur genre : <i>le pattern Singleton</i>	169
6	Encapsuler l'invocation : <i>le pattern Commande</i>	191
7	S'avoir s'adapter : <i>les patterns Adaptateur et Façade</i>	235
8	Encapsuler les algorithmes : <i>le pattern Patron de méthode</i>	275
9	Des collections bien gérées : <i>les patterns Itérateur et Composite</i>	315
10	L'état des choses : <i>le pattern État</i>	385
11	Contrôler l'accès aux objets : <i>le pattern Proxy</i>	429
12	Patterns de Patterns : <i>Patterns composés</i>	499
13	Les patterns dans le monde réel : <i>Mieux vivre avec les patterns</i>	577
14	Annexe : <i>Les patterns restants</i>	611

Table des matières (le contenu réel)

Intro

Votre cerveau et les Design Patterns. Voilà que *vous* essayez d'apprendre quelque chose tandis que *votre cerveau* fait tout ce qu'il peut pour vous empêcher de mémoriser. Votre cerveau pense « Mieux vaut laisser de la place pour les choses vraiment importantes, comme savoir quels sont les animaux sauvages qu'il convient d'éviter ou se rendre compte que faire du snowboard en maillot de bain n'est pas une bonne idée ». Mais *comment* procéder pour le convaincre que votre vie dépend de votre maîtrise des Design Patterns ?

À qui s'adresse ce livre ?	xxiv
Nous savons ce que pense votre cerveau	xxv
Métacognition	xxvii
Soumettez votre cerveau	xxix
Éditeurs techniques	xxxii
Remerciements	xxxiii

Bienvenue aux Design Patterns

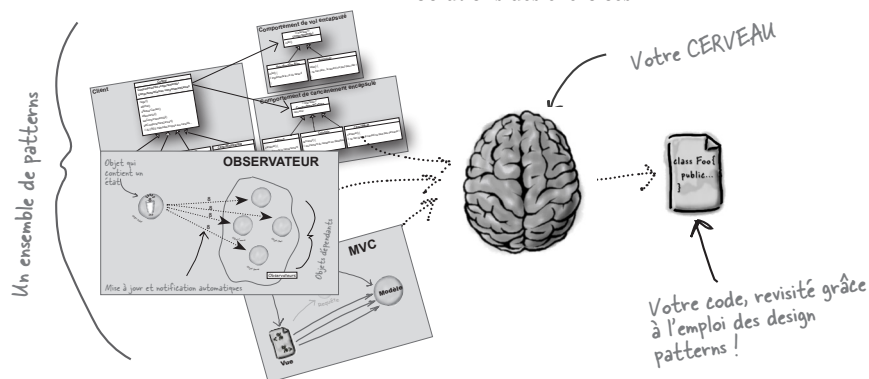
1

Quelqu'un a déjà résolu vos problèmes. Dans ce chapitre, vous allez apprendre comment (et pourquoi) exploiter l'expérience et les leçons tirées par d'autres développeurs qui ont déjà suivi le même chemin, rencontré les mêmes problèmes de conception et survécu au voyage. Nous allons voir l'usage et les avantages des design patterns, revoir quelques principes fondamentaux de la conception OO et étudier un exemple de fonctionnement d'un pattern. La meilleure façon d'utiliser un pattern est de le *charger dans votre cerveau* puis de *reconnaître* les points de vos conceptions et des applications existantes auxquels vous pouvez *les appliquer*. Au lieu de réutiliser du *code*, les patterns vous permettent de réutiliser de *l'expérience*.

Souvenez-vous : connaître des concepts tels que l'abstraction, l'héritage et le polymorphisme ne fait pas de vous un bon concepteur orienté objet. Celui qui maîtrise les patterns sait comment créer des conceptions souples, faciles à maintenir et capables de résister au changement.



Le simulateur de canards	2
Joël réfléchit à l'héritage....	5
Et si nous utilisions une interface ?	6
La seule et unique constante du développement	8
Séparer ce qui change de ce qui reste identique	10
Concevoir le comportement des canards	11
Tester le code	18
Modifier dynamiquement les comportements	20
Vue d'ensemble des comportements encapsulés	22
A-UN peut être préférable à EST-UN	23
Le pattern Stratégie	24
Le pouvoir d'un vocabulaire partagé	28
Comment utiliser les design patterns ?	29
Votre boîte à outils de concepteur	32
Solutions des exercices	34

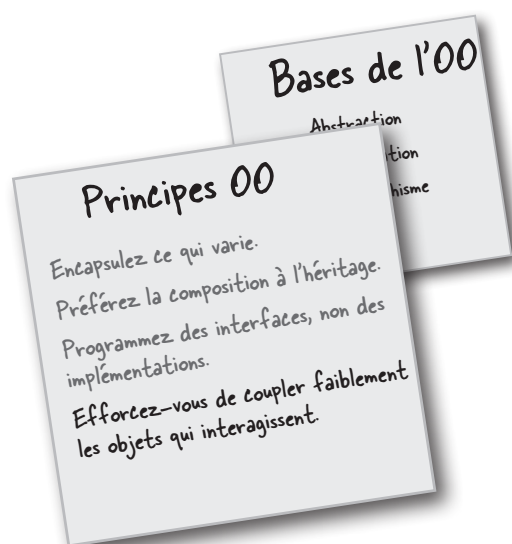


le pattern Observer

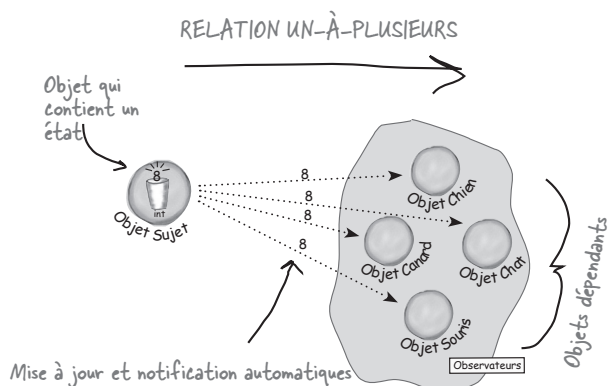
Tenez vos objets au courant

2

Ne manquez plus jamais rien d'intéressant ! Nous avons un pattern qui met vos objets au courant quand il se passe quelque chose qui pourrait les concerner. Ils peuvent même décider au moment de l'exécution s'ils veulent rester informés. Observateur est l'un des patterns le plus utilisé dans le JDK et il est incroyablement utile. Avant la fin de ce chapitre, nous étudierons également les relations «un-à-plusieurs» et le faible couplage (oui, oui, nous avons dit couplage). Avec Observateur, vous serez l'attraction de la soirée Patterns.



La station météorologique	39
Faites connaissance avec le pattern Observateur	44
Diffusion + Souscription = pattern Observateur	45
Comédie express : un sujet d'observation	48
Le pattern Observateur : définition	51
Le pouvoir du faible couplage	53
Concevoir la station météo	56
Implémenter la station météo	57
Utiliser le pattern Observateur de Java	64
La face cachée de java.util.Observable	71
Votre boîte à outils de concepteur	74
Solutions des exercices	78

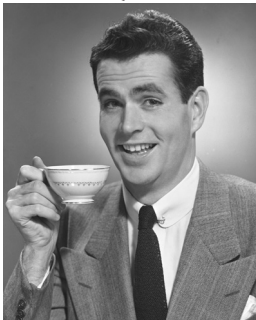


Décorer les objets

3

Ce chapitre pourrait s'appeler « **Les bons concepteurs se méfient de l'héritage** ». Nous allons revoir la façon dont on abuse généralement de l'héritage et vous allez apprendre comment « décorer » vos classes au moment de l'exécution en utilisant une forme de composition. Pourquoi ? Une fois que vous connaîtrez les techniques de décoration, vous pourrez affecter à vos objets (ou à ceux de quelqu'un d'autre) de nouvelles responsabilités *sans jamais modifier le code des classes sous-jacentes*.

Je croyais que les vrais hommes sous-classaient tout. C'était avant de découvrir le pouvoir de l'extension au moment de l'exécution et non à celui de la compilation. Maintenant, regardez-moi !

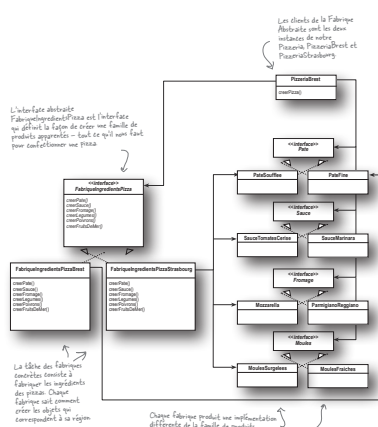


Bienvenue chez Starbuzz Coffee	80
Le principe Ouvert-Fermé	86
Faites connaissance avec le pattern Décorateur	88
Construire une boisson avec des décorateurs	89
Le pattern Décorateur : définition	91
Décorons nos boissons	92
Écrire le code de Starbuzz	95
Décorateurs du monde réel : les E/S Java	100
Écrire votre propre décorateur d'E/S Java	102
Votre boîte à outils de concepteur	105
Solutions des exercices	106

4

Un peu de cuisine orientée objet

Apprêtez-vous à confectionner des conceptions OO faiblement couplées. Créer des objets ne se limite pas à utiliser l'opérateur **new**. Vous allez apprendre que l'instanciation est une activité qui ne devrait pas toujours être publique et qui peut souvent entraîner des *problèmes de couplage*. Et vous n'en avez pas vraiment envie, n'est-ce pas ? Découvrez comment les patterns fabriques peuvent vous aider à vous libérer de dépendances embarrassantes.



Quand vous voyez « new », pensez « concret »	110
À la Pizzeria d'Objectville	112
Encapsuler la création des objets	114
Construire une simple fabrique de pizzas	115
Fabrique Simple : définition	117
Une structure pour la pizzeria	120
Laisser les sous-classes décider	121
Créer une pizzeria	123
Déclarer une méthode de fabrique	125
Il est enfin temps de rencontrer le pattern Fabrication	131
Hiérarchies de classes parallèles	132
Le Pattern Fabrication : définition	134
Une Pizzeria très dépendante	137
Problèmes de dépendance des objets	138
Le principe d'inversion des dépendances	139
Pendant ce temps, à la pizzeria...	144
Familles d'ingrédients...	145
Construire nos fabriques d'ingrédients	146
Un coup d'œil à Fabrique abstraite	153
Dans les coulisses	154
Le pattern Fabrique abstraite : définition	156
Fabrication et Fabrique Abstraite comparés	160
Votre boîte à outils de concepteur	162
Solutions des exercices	164

le pattern Singleton

Des objets uniques en leur genre

5

Notre prochain arrêt est le Pattern Singleton notre passeport pour la création d'objets uniques en leur genre dont il n'existe qu'une seule instance. Vous allez sûrement être ravi d'apprendre que, parmi tous les patterns, le Singleton est le plus simple en termes de diagramme de classes. En fait, ce diagramme ne contient qu'une seule classe ! Mais ne vous méprenez pas : malgré sa simplicité du point de vue de la conception des classes, nous allons rencontrer pas mal de bosses et de nids de poule dans son implémentation. Alors, bouclez vos ceintures.



Un objet et un seul	170
Le Petit Singleton	171
Disséquons l'implémentation du Pattern Singleton	173
Confessions d'un Singleton	174
La fabrique de chocolat	175
Le pattern Singleton : définition	177
Houston Nous avons un problème...	178
Vous êtes la JVM	179
Gérer le multithread	180
Singleton : questions et réponses	184
Votre boîte à outils de concepteur	186
Solutions des exercices	188

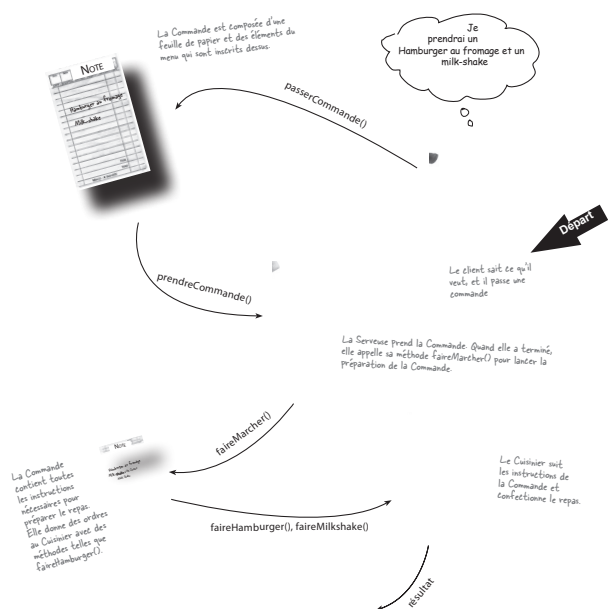


le pattern Commande

6

Encapsuler l'invocation

Dans ce chapitre, nous allons envisager l'encapsulation à un tout autre niveau : nous allons encapsuler l'invocation des *méthodes*. Oui, en encapsulant les appels de méthodes, nous allons pouvoir « cristalliser » des traitements afin que l'objet qui les invoque n'ait pas besoin de savoir comment il sont exécutés : il suffit qu'il utilise nos méthodes cristallisées pour obtenir un résultat. Nous pourrions même faire des choses drôlement futées avec ces appels de méthodes encapsulés, par exemple les sauvegarder pour la journalisation ou les réutiliser pour implémenter des annulations dans notre code.



Maisons de rêve	192
La télécommande	193
Un coup d'œil aux classes des fournisseurs	194
Pendant ce temps, à la cafétéria...	197
Étudions les interactions	198
Rôles et responsabilités	199
De la Cafétéria au pattern Commande	201
Notre premier objet de commande	203
Le pattern Commande : définition	206
Le pattern Commande et la télécommande	208
Implémenter la télécommande	210
Tester la télécommande	212
Il est temps de rédiger cette documentation...	215
Utiliser un état pour implémenter une annulation	220
Toute télécommande a besoin d'un mode groupé !	224
Utiliser une macrocommande	225
Autres utilisations de Commande : files de requêtes	228
Autres utilisations de Commande : journalisation des requêtes	229
Votre boîte à outils de concepteur	230
Solutions des exercices	232

les patterns Adaptateur et Façade

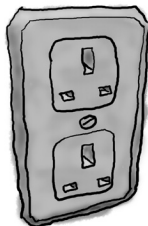
Savoir s'adapter

7

Dans ce chapitre, nous allons entreprendre des choses impossibles, comme faire entrer une cheville ronde dans un trou carré. Cela vous semble impossible ?

Plus maintenant avec les design patterns. Vous souvenez-vous du pattern Décorateur ? Nous avons enveloppé des objets pour leur attribuer de nouvelles responsabilités. Nous allons recommencer, mais cette fois avec un objectif différent : faire ressembler leurs interfaces à quelque chose qu'elles ne sont pas. Pourquoi donc ? Pour pouvoir adapter une conception qui attend une interface donnée à une classe qui implémente une interface différente. Et ce n'est pas tout. Pendant que nous y sommes, nous allons étudier un autre pattern qui enveloppe des objets pour simplifier leur interface.

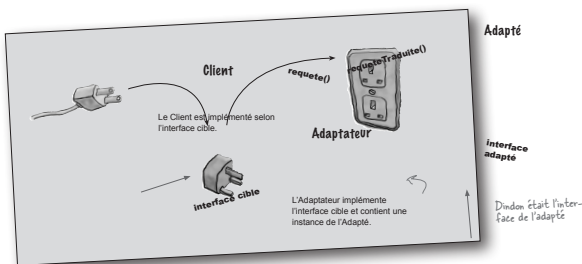
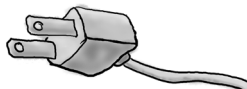
Prise murale européenne



Adaptateur CA



Fiche CA US



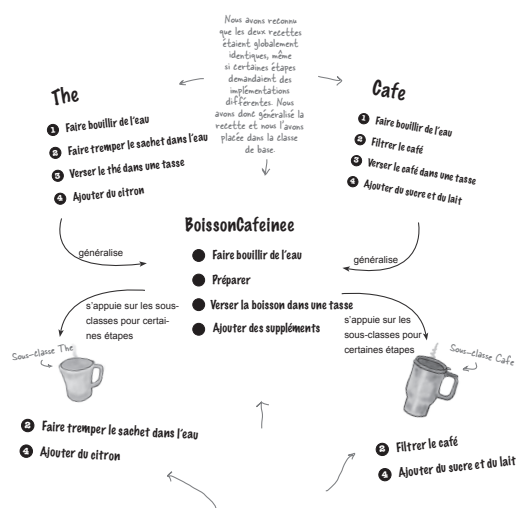
Nous sommes entourés d'adaptateurs	236
Adaptateurs orientés objet	237
Le pattern Adaptateur expliqué	241
Le pattern Adaptateur : définition	243
Adaptateurs de classe et adaptateurs d'objet	244
Face-à-face : l'Adaptateur d'objet et l'Adaptateur de classe	247
Adaptateurs du monde réel	248
Adapter une Enumeration à un Iterator	249
Face-à-face : Décorateur et Adaptateur	252
Home Cinéma	255
Lumières, Caméra, Façade !	258
Construire la façade de votre home cinéma	261
Le pattern Façade : définition	264
Ne parlez pas aux inconnus	265
Votre boîte à outils de concepteur	270
Solutions des exercices	272

le pattern Patron de méthode

Encapsuler les algorithmes

8

Nous n'arrêtons pas d'encapsuler. Nous avons encapsulé la création d'objets et l'invocation de méthodes. Nous avons encapsulé des interfaces complexes, des canards, des pizzas... Par quoi pourrions nous continuer ? Nous allons entreprendre d'encapsuler des fragments d'algorithmes afin que les sous-classes puissent s'y adapter au moment de leur choix. Nous allons même découvrir un principe de conception inspiré par Hollywood.



Des classes pour préparer le café et le thé	277
Puis-je transformer votre café en abstraction ?	280
Approfondir la conception	281
Abstraire suivreRecette()	282
Qu'avons-nous fait ?	285
Faites connaissance avec Patron de méthode	286
Préparons un peu de thé	287
Que nous a apporté le Patron de méthode ?	288
Le pattern Patron de méthode : définition	289
Code à la loupe	290
Méthodes adaptateurs et Patron de méthode...	292
Utiliser la méthode adaptateur	293
Café ? Thé ? Exécutons le test	294
Le principe d'Hollywood	296
Principe d'Hollywood et Patron de méthode	297
Patrons de méthode à l'état sauvage	299
Trier avec Patron de méthode	300
Nous avons des canards à trier..	301
Comparer des canards et des canards	302
Les secrets de la machine à trier les canards	304
Les adaptateurs de Swing	306
Applets	307
Face-à-face : Patron de méthode et stratégie	308
Votre boîte à outils de concepteur	311
Solutions des exercices	312

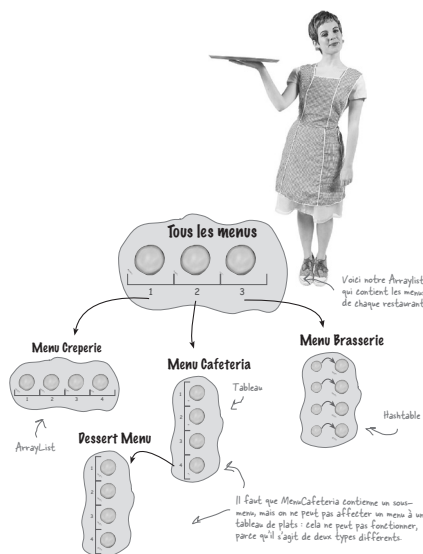
les patterns Itérateur et Composite

Des collections bien gérées

9

Il y a des quantités de façon de placer des objets dans une collection. Tableaux, piles, listes, tables de hachage :

vous avez le choix. Chacune a ses avantages et ses inconvénients. Mais il y aura toujours un moment où votre client voudra opérer des itérations sur ces objets. Allez-vous alors lui montrer votre implémentation ? Espérons que non ! Ce serait un manque de professionnalisme absolu. Eh bien vous n'avez pas besoin de risquer votre carrière. Vous allez voir comment autoriser vos clients à procéder sans même jeter un coup d'œil à la façon dont vous stockez vos objets. Vous apprendrez également à créer des super collections d'objets qui peuvent parcourir des structures de données impressionnantes d'un seul trait. Et si cela ne suffit pas, vous allez découvrir une chose ou deux sur les responsabilités des objets.

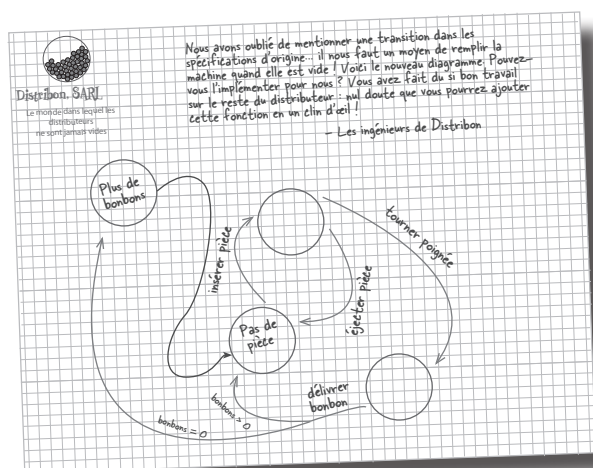


Fusion de la Cafétéria et de la Crêperie d'Objectville	316
Les implémentations de Léon et Noël	318
Pouvons-nous encapsuler l'itération ?	323
Faites connaissance avec le pattern Itérateur	325
Ajoutons un itérateur à MenuCafeteria	326
Examiner la conception actuelle	331
Une conception plus propre avec java.util.Iterator	333
Où cela nous amène-t-il ?	335
Le pattern Itérateur : définition	336
Une seule responsabilité	339
Itérateurs et collections	348
Itérateurs et collections en Java 5	349
Juste au moment où nous pensions avoir terminé...	353
Le pattern Composite : définition	356
Concevoir des menus avec Composite	359
Implémenter le Composite	362
Flash-back sur Itérateur	368
L'itérateur nul	372
La magie d'Itérateur et Composite en duo...	374
Votre boîte à outils de concepteur	380
Solutions des exercices	381

L'état des choses

Un fait méconnu : les patterns Stratégie et État sont des jumeaux qui ont été séparés à la naissance.

Comme vous le savez, le pattern Stratégie a créé une affaire des plus florissantes sur le marché des algorithmes interchangeables. Quant à lui, État a peut-être suivi une voie plus noble : il aide les objets à contrôler leur comportement en modifiant leur état interne. On le surprend souvent à dire à ses clients « Répétez après moi : Je suis bon, je suis intelligent, ma simple présence suffit... »



Comment implémenter un état ?	387
Machines à états : première mouture	388
Premier essai de machine à états	390
Vous vous y attendiez... une demande de changement !	394
Nous sommes dans un drôle d'état...	396
Définir l'interface et les classes pour les états	399
Implémenter les classes pour les états	401
Retravaillons le distributeur	402
Le pattern État : définition	410
État vs Stratégie	411
Bilan de santé	417
Nous avons failli oublier !	420
Votre boîte à outils de concepteur	423
Solutions des exercices	424



11

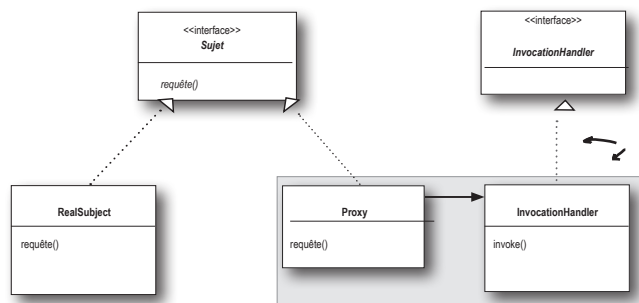
Contrôler l'accès aux objets

Avez-vous déjà joué à « gentil flic, méchant flic » ?

Vous êtes le gentil policier et vous rendez volontiers service de bonne grâce, mais comme vous ne voulez pas que *qui que ce soit* vous sollicite, vous demandez au méchant de servir d'intermédiaire. C'est là le rôle d'un proxy : *contrôler* et gérer les accès. Comme vous allez le voir, un proxy peut se substituer à un objet d'une quantité de façons. Les proxies sont connus pour transporter des appels de méthodes entiers sur l'Internet à la place d'autres objets. On les connaît également pour remplacer patiemment un certain nombre d'objets bien paresseux.



Contrôler les distributeurs	430
Le rôle du proxy distant	434
Détour par RMI	437
Un Proxy pour le distributeur	450
Les coulisses du proxy distant	458
Le pattern Proxy : définition	460
Prêts pour le Proxy virtuel ?	462
Concevoir le proxy virtuel pour les pochettes de CD	464
Les coulisses du proxy virtuel	470
Utiliser la classe Proxy de l'API Java	474
Comédie express : des sujets sous protection	478
Créer un proxy dynamique	479
Le zoo des proxys	488
Votre boîte à outils de concepteur	491
Solutions des exercices	492

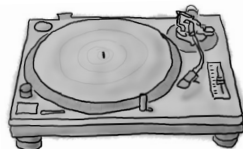
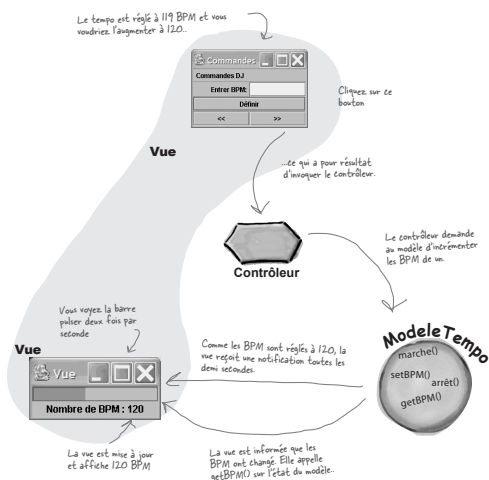


patterns composés

Patterns de patterns

12

Qui aurait cru que les patterns étaient capables de collaborer ? Vous avez déjà été témoin de l'animosité qui règne dans les face-à-face (et vous n'avez pas vu le combat à mort que l'éditeur nous a forcés à retirer de ce livre). Alors qui aurait pu penser que des patterns pourraient finir par s'entendre ? Eh bien, croyez-le ou non, certaines des conceptions OO les plus puissantes font appel à plusieurs patterns. Apprêtez-vous à passer à la vitesse supérieure : il est temps d'étudier les patterns composés.

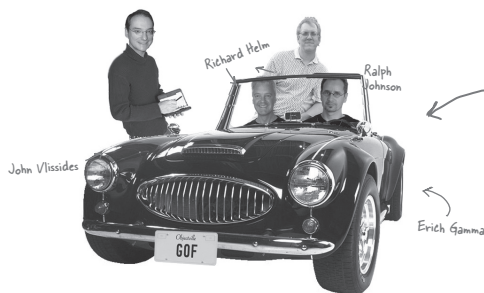
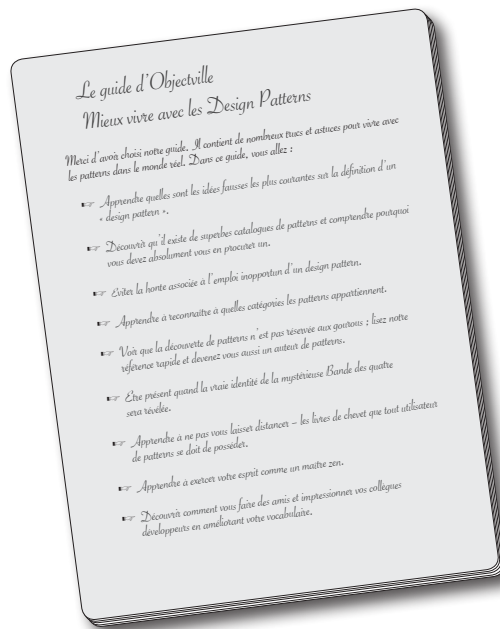


Patterns composés	500
Réunion de famille	501
Ajouter un adaptateur	504
Ajouter un décorateur	506
Ajouter une fabrique	508
Ajouter un composite et un itérateur	513
Ajouter un observateur	516
Résumé des patterns	523
Vue plongeante : le diagramme de classes	524
Modèle-Vue-Contrôleur , la chanson	526
Les design patterns sont la clé de MVC	528
Mettons nos lunettes « spéciales patterns »	532
Contrôler le tempo avec MVC...	534
Le Modèle	537
La Vue	539
Le Contrôleur	542
Explorer le pattern Stratégie	545
Adapter le modèle	546
Maintenant, nous sommes prêts à contrôler un cœur	547
MVC et le Web	549
Design patterns et Model 2	557
Votre boîte à outils de concepteur	560
Solutions des exercices	561

13

Les patterns dans le monde réel

Maintenant, vous êtes prêt à vivre dans un monde nouveau, un monde peuplé de Design Patterns. Mais avant de vous laisser ouvrir toutes ces nouvelles portes, nous devons aborder quelques détails que vous rencontrerez dans le monde réel. Eh oui, les choses y sont un peu plus complexes qu'elles ne le sont ici, à Objectville. Suivez-nous ! Vous trouverez à la page suivante un guide extra qui vous aidera à effectuer la transition...

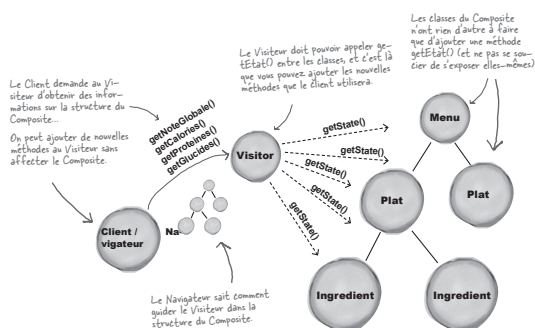


Le guide d'Objectville	578
Design Pattern : définition	579
Regardons la définition de plus près	581
Que la force soit avec vous	582
Catalogues de patterns	583
Comment créer des patterns	586
Donc, vous voulez être auteur de patterns...?	587
Organiser les design patterns	589
Penser en termes de patterns	594
Votre esprit et les patterns	597
N'oubliez pas le pouvoir du vocabulaire partagé	599
Les cinq meilleures façons de partager votre vocabulaire	600
Tour d'Objectville avec la Bande des quatre	601
Votre voyage ne fait que commencer...	602
Autres ressources sur les design patterns	603
Le zoo des patterns	604
Annihiler le mal avec les anti-patterns	606
Votre boîte à outils de concepteur	608
Quitter Objectville...	609

14

Annexe : Les patterns restants

La popularité n'est pas donnée à tout le monde. Beaucoup de choses ont changé ces dix dernières années. Depuis la parution de *Design Patterns : Catalogue de modèles de conception réutilisables*, les développeurs ont appliqué ces patterns des milliers de fois. Ceux que nous avons résumés ici sont des patterns GoF à part entière. Bien qu'ils soient tout aussi officiels que ceux que nous avons étudiés jusqu'à maintenant, leur emploi est moins fréquent. Mais ils sont tout aussi remarquables, et, si la situation le demande, vous pouvez les utiliser la tête haute. Dans cette annexe, notre objectif est de vous donner une vue d'ensemble de ces patterns.



Pont	612
Monteur	614
Chaîne de responsabilité	616
Poids-mouche	618
Interprète	620
Médiateur	622
Memento	624
Prototype	626
Visiteur	628

Index

631



comment lire ce livre

Intro



Dans cette section, nous répondons à la question qui vous brûle les lèvres :
« Pourquoi ont-ils mis ça dans un ouvrage sur les design patterns ? »

À qui s'adresse ce livre ?

Si vous répondez « oui » à ces trois questions :

*Si vous connaissez C#,
c'est bien aussi.*

- ① **Connaissez-vous Java ? (Inutile d'être un gourou.)**
- ② Voulez-vous **apprendre, comprendre, mémoriser** et **appliquer** les design patterns, ainsi que les bons principes de conception sur lesquels ces patterns s'appuient ?
- ③ **Préférez-vous une conversation stimulante autour d'un bon dîner à un cours bien ennuyeux et bien aride ?**

cet ouvrage est pour vous.

Et à qui n'est-il pas destiné ?

Si vous répondez « oui » à toutes ces questions :

- ① **Êtes-vous totalement débutant en Java ?**
(Vous n'avez pas besoin d'être très avancé, et, même si vous ne connaissez pas *Java* mais uniquement C#, vous comprendrez sans doute au moins 80 % des exemples de code. Une expérience en C++ pourrait également suffire.)
- ② Êtes-vous un concepteur ou un développeur OO pressé recherchant un **ouvrage de référence** ?
- ③ Êtes-vous un architecte à la recherche de design patterns **d'entreprise** ?
- ④ Avez-vous **peur d'essayer quelque chose de différent** ?
Préfereriez-vous vous faire arracher une dent plutôt que de mélanger des carreaux et des rayures ? Croyez-vous qu'un ouvrage technique ne peut pas être sérieux si les composants Java y sont anthropomorphisés ?

Ce livre n'est pas pour vous.



*[note du marketing : ce livre est destiné à toute
personne possédant une carte de crédit.]*

Nous savons ce que vous pensez.

«Comment ce livre peut-il être sérieux?»

«À quoi riment toutes ces images?»

«Est-ce qu'on peut vraiment *apprendre* de cette façon?»

«Je mangerais bien une pizza.»

Et nous savons ce que votre cerveau pense.

Votre cerveau est avide de nouveauté. Toujours à l'affût, il ne cesse de chercher et d'attendre quelque chose d'inhabituel. C'est comme ça qu'il est construit, et c'est comme ça qu'il vous aide à rester en vie.

Aujourd'hui, il est peu probable que vous serviez de petit-déjeuner à un tigre. Mais votre cerveau reste sur ses gardes. On ne sait jamais...

Alors que fait-il de toutes les choses ordinaires, routinières et banales que vous rencontrez? Il fait tout ce qu'il *peut* pour les empêcher d'interférer avec son vrai travail: enregistrer ce qui compte. Il ne se soucie pas de *mémoriser* les choses ennuyeuses; il considère *a priori* qu'elles n'ont «aucune importance» et il les filtre.

Mais comment votre cerveau sait-il ce qui est important? Supposons que vous partiez en excursion pour la journée et qu'un tigre surgisse devant vous. Que se passe-t-il dans votre tête?

Les neurones s'embrasent. Les émotions vous submergent. *Les hormones déferlent.*

Et votre cerveau ne sait rien d'autre...

Cela peut être important! Ne l'oubliez pas!

Mais imaginez que vous êtes chez vous ou dans une bibliothèque. C'est un lieu sûr et chaleureux. Pas le moindre tigre à l'horizon... Vous êtes en train d'étudier. De réviser un examen. Ou d'essayer de comprendre un sujet bien technique dont votre patron vous a dit qu'il vous prendra huit à dix jours au plus.

Seul problème: votre cerveau essaie de vous faire un gros cadeau. Il essaie de faire en sorte que des contenus *évidemment* sans importance n'épuisent des ressources déjà rares. Des ressources mieux employées à mémoriser les choses vraiment *essentiels*. Comme les tigres. Comme les incendies. Comme le danger qu'il y a à faire du snowboard en short.

Et il n'y a aucun moyen simple de lui dire: «Hé, cerveau, merci bien, mais même si ce livre est ennuyeux comme la pluie, et même si j'ai le moral à zéro, je *veux* que tu fasses attention.»



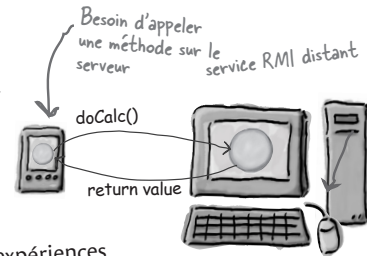
Nous considérons notre lecteur comme un apprenant.

Comment *apprend-on* quelque chose ? Il faut d'abord *comprendre*, puis faire en sorte de ne pas *oublier*. Il ne s'agit pas de remplir sa tête de force. Les dernières recherches en sciences cognitives, en neurobiologie et en psychologie de l'éducation montrent que l'*apprentissage* exige autre chose que du texte imprimé. Nous savons ce qui stimule votre cerveau.

Voici quelques-uns de nos principes :

Permettre de visualiser. Les images sont beaucoup plus faciles à mémoriser que les mots, et facilitent beaucoup l'apprentissage (jusqu'à 89% d'amélioration dans les études sur le rappel et le transfert). Elles aident également à mieux comprendre.

Placer le texte près de l'image auquel il se rapporte, et non à la fin de la page ou plus loin, permet aux apprenants de résoudre deux fois plus vite les problèmes liés aux contenus.



C'est terrible d'être une méthode abstraite. On n'a pas de corps.



`abstract void roam();`

Pas de corps de méthode !
Terminer par un point-virgule.

Adopter un style conversationnel et personnalisé. De récentes expériences ont montré que des étudiants se rappelaient beaucoup mieux un texte s'adressant directement à eux sur le ton de la conversation que des sujets qui avaient lu un texte présenté de façon plus formaliste. Leurs performances étaient supérieures de 40%. Mieux vaut raconter une histoire que pontifier, parler le langage de tous les jours, éviter de se prendre trop au sérieux. À qui prêtez-vous le plus d'attention : à l'ami avec qui vous dînez ou à un conférencier ?

Faire réfléchir. Si vous n'exercez pas activement vos neurones, il ne se passe pas grand chose dans votre tête. Un lecteur doit être motivé, engagé, curieux. Il doit avoir envie de résoudre des problèmes, de tirer des conclusions et de générer de nouvelles connaissances. Pour ce faire, il a besoin de défis, d'exercices, de questions qui incitent à réfléchir et d'activités qui sollicitent les deux hémisphères cérébraux et font appel à plusieurs sens.

Capter — et conserver — l'attention du lecteur. Nous avons tous pensé un jour « Je veux vraiment lire ce livre, mais il me tombe des mains dès la première page ». Votre cerveau prête attention à ce qui sort de l'ordinaire, qui attire l'œil, à ce qui est intéressant, étrange, inattendu. L'étude d'un nouveau sujet technique, même difficile, n'a pas besoin d'être fastidieuse. Votre cerveau apprendra plus facilement si vous ne vous ennuyez pas.

Est-ce qu'une Salle de Bains EST-UNE Baignoire ? Ou bien est-ce une relation A-UN ?



Faire appel aux émotions. Nous savons maintenant que notre aptitude à mémoriser quelque chose dépend largement de son contenu émotionnel.

Vous vous souvenez de ce qui compte pour vous. Vous vous rappelez quand vous ressentez quelque chose. Non, nous ne parlons pas d'histoires à fendre le cœur avec un petit garçon et son chien. Nous parlons d'émotions comme la surprise, la curiosité, l'amusement ou la sensation de toute-puissance que vous éprouvez lorsque vous résolvez un puzzle, que vous apprenez quelque chose que tout le monde considère comme difficile, ou lorsque vous vous rendez compte que vous savez quelque chose que les super-gourous ne savent pas.



Métacognition : apprendre à apprendre

Si vous voulez réellement apprendre, plus vite et plus en profondeur, soyez attentif à la manière dont vous faites attention. Réfléchissez à la façon dont vous réfléchissez. Apprenez à apprendre.

La plupart d'entre nous n'ont pas suivi de cours sur la métacognition ou la théorie de l'apprentissage lorsque nous étions à l'école. Nous étions censés apprendre, mais on nous disait rarement comment faire.

Mais si vous avez ce livre entre les mains, nous supposons que vous voulez apprendre Java. Et vous ne voulez probablement pas y consacrer une éternité.

Pour tirer le meilleur parti de cet ouvrage, d'un autre livre ou de toute expérience d'apprentissage, apprenez à maîtriser votre cerveau.

L'astuce consiste à l'amener à considérer ce que vous allez apprendre comme quelque chose de réellement important, de capital pour votre bien-être, d'aussi important qu'un tigre. Sinon, vous êtes voué à un combat sans fin : votre cerveau fera tout son possible pour vous empêcher de mémoriser.

Mais comment **FAIRE** au juste pour que votre cerveau traite Java comme un tigre affamé ?

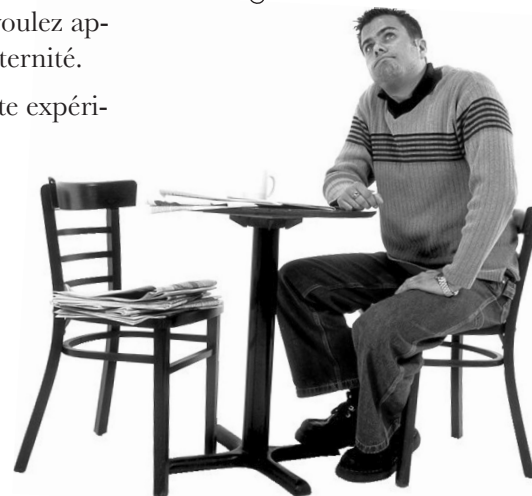
Il y a deux façons de procéder : l'une lente et ennuyeuse, l'autre rapide et efficace. La solution lente consiste purement et simplement à répéter. Vous savez certainement que vous êtes capable d'apprendre et de mémoriser le plus ingrat des sujets si vous répétez sans arrêt la même chose. Au bout d'un nombre suffisant de répétitions, votre cerveau pense : « Ça ne me semble pas important pour lui, mais s'il ressasse ce truc, c'est que ça doit l'être. »

La voie rapide consiste à faire *tout ce qui non seulement augmente mais aussi diversifie l'activité cérébrale*. Les éléments de la page précédente constituent une grande partie de la solution, et ils se sont tous révélés capables d'aider votre cerveau à travailler en votre faveur. Certaines études démontrent que l'insertion de mots *dans* les images qu'ils décrivent (et non dans des légendes ou dans le corps du texte) force le cerveau à tenter de comprendre les relations entre le texte et l'image, et provoque des connexions neuronales plus nombreuses. Plus ces connexions se multiplient, plus il a de chances de comprendre que le contenu est digne d'attention et mérite d'être retenu.

Un style conversationnel aide également. Les gens ont tendance à faire plus attention à une conversation, puisqu'ils sont censés la suivre et éventuellement y prendre part. Chose surprenante : votre cerveau ne se soucie pas nécessairement de savoir que vous « conversez » avec un livre ! En revanche, si le style est aride et formaliste, il le perçoit exactement comme celui d'un conférencier pérorant devant une salle pleine d'auditeurs passifs. Inutile en ce cas de rester éveillé.

Mais les graphismes et le ton de la conversation ne sont qu'un début.

Qu'est-ce que je vais bien pouvoir trouver pour faire rentrer ce truc dans ma tête ?



Voici ce que NOUS avons fait :

Nous avons utilisé des **images**, parce que votre cerveau les préfère au texte. Pour lui, une seule d'entre elles vaut réellement mieux que 1024 mots. Et, lorsque texte et image vont de pair, nous avons fusionné le texte et l'image, parce que votre cerveau fonctionne mieux lorsque les mots sont intégrés à ce qu'il décrivent plutôt que placés dans une légende ou enfouis quelque part dans la page imprimée.

Nous avons utilisé la **répétition**, en exprimant la même chose de différentes manières, en recourant à divers types de supports et en faisant appel à **plusieurs sens**, pour augmenter les chances que le contenu soit encodé dans plusieurs zones de votre cerveau.

Nous avons utilisé concepts et images de façon **inattendue**, parce que votre cerveau aime la nouveauté. Nous avons ajouté au moins un peu de contenu **émotionnel**, parce que votre cerveau prête attention à la biochimie des émotions. C'est ce qui vous aide à mémoriser, même si ce n'est dû à rien d'autre qu'un peu d'**humour**, de **surprise** ou d'**intérêt**.

Nous avons utilisé un style conversationnel, personnel, parce que votre cerveau est plus enclin à être attentif quand il croit que vous êtes engagé dans une conversation que lorsqu'il pense que vous assistez passivement à une présentation. Il fait de même quand vous lisez.

Nous avons inclus plus de 50 **exercices**, parce que votre cerveau apprend et retient mieux quand vous **faites** quelque chose que lorsque vous lisez. Ces exercices sont «difficiles mais faisables» parce que c'est ce que la plupart des gens préfèrent.

Nous avons fait appel à **plusieurs styles d'apprentissage**, parce que vous préférez peut-être les procédures pas-à-pas, alors que quelqu'un d'autre voudra d'abord saisir l'image globale et qu'un troisième lecteur cherchera simplement un exemple de code. Mais, indépendamment de ses préférences, chacun peut tirer parti de la vision d'un même contenu diversement présenté.

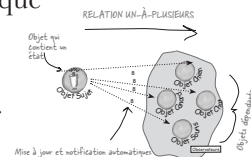
Nous avons inclus des contenus qui s'adressent aux **deux hémisphères de votre cerveau** : plus ce dernier est engagé, plus vous êtes à même d'apprendre, de mémoriser et de vous concentrer plus longtemps. Comme faire travailler l'un des côtés du cerveau permet souvent à l'autre de se reposer, la période durant laquelle votre apprentissage est productif est plus longue.

Nous avons également incorporé des **histoires** et des **exercices** qui présentent **plusieurs points** de vue, parce que votre cerveau est ainsi fait que vous apprenez plus en profondeur s'il est forcé de faire des évaluations et de porter des jugements.

Enfin, nous avons inclus des **défis** dans les exercices en posant des questions qui n'ont pas de réponse simple, parce que votre cerveau apprend et retient mieux lorsqu'il doit travailler (tout comme il ne suffit pas de regarder les autres à la gym pour rester en bonne forme physique). Mais nous avons fait de notre mieux pour que, si vous devez travailler dur, ce soit sur les bons éléments : pour que vous ne consacriez pas un neurone de trop à traiter un exemple difficile à comprendre, ni à analyser un texte difficile, bourré de jargon et horriblement laconique.

Nous avons fait appel à des **personnes** dans les histoires, dans les exemples et dans les images, parce que vous êtes une personne. Et votre cerveau s'intéresse plus aux personnes qu'aux choses.

Nous avons adopté une approche **80/20**. Si vous visez un doctorat en Java, nous supposons que vous ne lirez pas que ce livre. C'est pourquoi nous ne parlons pas de tout, mais seulement de ce que vous utilisez réellement.



Le gourou
des patterns

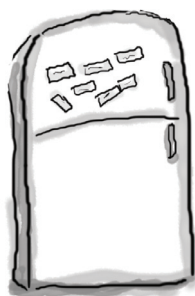


POINTS D'IMPACT

Problèmes de
conception



Voici ce que VOUS pouvez faire pour dompter votre cerveau.



à découper et à coller sur le frigo

Nous avons donc fait notre part. Le reste dépend de vous. Les conseils qui suivent sont un point de départ. Écoutez votre cerveau et déterminez ce qui fonctionne pour vous et ce qui ne fonctionne pas. Essayez de nouvelles techniques.

① **Prenez votre temps. Plus vous comprenez, moins vous avez à mémoriser.**

Ne vous bornez pas à *lire*. Arrêtez-vous et réfléchissez. Quand vous voyez une question, ne vous précipitez pas sur la réponse. Imaginez que quelqu'un la pose réellement. Plus vous forcez votre cerveau à penser, plus vous avez de chances d'apprendre et de retenir.

② **Faites les exercices. Prenez des notes.**

Si nous les avons faits pour vous, ce serait comme de demander à quelqu'un de faire de la gym à votre place. Et ne vous contentez pas de les lire. **Prenez un crayon.** Tout montre que l'activité physique *durant* l'apprentissage permet d'apprendre plus.

③ **Lisez « Il n'y a pas de questions stupides ».**

Et lisez tout. Ce ne sont pas des encadrés optionnels : ils font partie du contenu ! Parfois, les questions sont plus utiles que les réponses.

④ **Ne restez pas toujours à la même place.**

Levez-vous, étirez-vous, déplacez-vous, changez de chaise, changez de pièce. Cela aide votre cerveau à *ressentir* quelque chose et empêche l'apprentissage d'être trop associé à un endroit particulier.

⑤ **Faites de ce livre votre dernière lecture avant de dormir. Au moins la dernière chose difficile.**

Une partie de l'apprentissage (surtout le transfert dans la mémoire à long terme) aura lieu *après* que vous aurez reposé le livre. Votre cerveau a besoin de temps à lui pour travailler plus. Si vous interposez quelque chose de nouveau dans ce laps de temps, une partie de ce que vous viendrez d'apprendre sera perdue.

⑥ **Buvez beaucoup d'eau.**

Votre cerveau fonctionne mieux lorsqu'il est convenablement irrigué. La déshydratation (qui peut apparaître avant la soif) diminue les fonctions cognitives.

⑦ **Parlez-en à voix haute.**

La parole active une autre partie du cerveau. Si vous essayez de comprendre quelque chose ou d'augmenter vos chances de vous le remémorer plus tard, dites-le à haute voix. Mieux encore, essayez de l'expliquer à quelqu'un. Vous apprendrez plus vite et vous découvrirez peut-être des idées qui vous ont échappé à la lecture.

⑧ **Écoutez votre cerveau.**

Veillez à ne pas le surcharger. Si vous vous surprenez à lire en diagonale ou à oublier ce que vous venez de lire, il est temps de faire une pause. Passé un certain point, vous n'apprenez pas plus vite en essayant d'en absorber davantage, et vous risquez même d'altérer le processus.

Ressentez quelque chose !

⑨ **Votre cerveau a besoin de savoir que c'est important.**

Impliquez-vous dans les histoires. Créez vos propres légendes pour les photos. S'agacer d'une mauvaise plaisanterie vaut mieux que ne rien ressentir du tout.

⑩ **Concevez quelque chose !**

Appliquez ce que vous apprenez à une nouvelle conception ou à la réorganisation d'un ancien projet. Faites *quelque chose*, afin d'acquérir une expérience qui dépasse les exercices et les activités de ce livre. Vous n'avez besoin que d'un crayon et d'un problème à résoudre... un problème qui pourrait bénéficier de l'introduction d'un ou plusieurs design patterns.

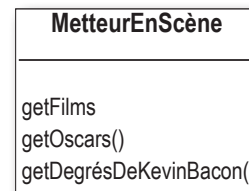
Lisez moi

Il s'agit d'une introduction aux design patterns, non d'un ouvrage de référence. Nous avons délibérément éliminé tout ce qui pouvait faire obstacle à l'*apprentissage*, quel que soit le sujet abordé à un moment donné. Et, lors de votre première lecture, vous devrez commencer par le commencement, parce que ce livre tient compte de ce que vous avez déjà vu et appris.

Nous utilisons un
« faux UML », un
UML simplifié ↗

Nous utilisons des diagrammes pseudo-UML simples.

Même s'il y a de grandes chances que vous ayez déjà rencontré UML, nous ne l'aborderons pas dans ce livre et il ne constitue pas un prérequis. Si vous ne connaissez pas encore UML, pas de souci : nous vous donnerons quelques indications en temps voulu. Autrement dit, vous n'aurez pas à étudier les Design patterns et UML en même temps. Nos diagrammes *ressemblent* à UML – même si nous essayons d'être fidèles au standard, nous faisons parfois quelques petites entorses aux règles, généralement pour des besoins égoïstement artistiques.



Nous n'aborderons pas tous les Design patterns de la création.

Il existe *beaucoup* de Design patterns : les patterns d'origine (connus sous le nom de patterns GoF), les patterns J2EE de Sun, les patterns JSP, les patterns architecturaux, les patterns spécifiques à la conception de jeux et des *quantités* d'autres. Mais comme notre objectif était d'écrire un livre qui ne pèse pas trois tonnes, nous ne les abordons pas tous ici. Nous nous concentrerons donc sur les patterns fondamentaux *les plus importants* parmi les patterns GoF d'origine, et nous nous assurerons que vous comprenez vraiment, réellement et en profondeur quand et comment les utiliser. Vous trouverez également en annexe un bref aperçu des autres (ceux que vous êtes moins susceptible d'utiliser). En tout état de cause, lorsque vous aurez terminé Design patterns *Tête la première*, vous serez capable de prendre n'importe quel catalogue de patterns et de vous y mettre rapidement.

Les activités ne sont PAS optionnelles.

Les exercices et les activités ne sont pas là pour décorer : ils font partie intrinsèque de ce livre. Certains sont là pour vous aider à mémoriser, d'autres à comprendre et d'autres encore pour vous aider à appliquer ce que vous avez appris. **Ne sautez pas les exercices.** Les mots-croisés sont les seules choses que vous n'êtes pas *obligé* de faire, mais ils donneront à votre cerveau une chance de réfléchir aux mots dans un contexte différent.

Nous employons le mot « composition » dans l'acception générale qu'il a en OO, qui est plus large que son sens strict en UML.

Lorsque nous disons « un objet est composé avec un autre objet », nous voulons dire que ces deux objets sont liés par une relation « A-UN ». Notre emploi reflète l'usage traditionnel du terme et c'est celui de l'ouvrage de la Bande des quatre (vous apprendrez ce que c'est plus tard). Plus récemment, UML a raffiné ce terme pour distinguer plusieurs types de composition. Si vous êtes expert en UML, vous pourrez toujours lire ce livre et faire facilement correspondre notre emploi du mot composition avec des concepts plus détaillés.

La redondance est intentionnelle et importante.

Un livre Tête la première possède un trait distinctif : nous voulons *réellement* que vous compreniez. Et nous voulons que vous vous souveniez de ce que vous avez appris lorsque vous le refermez. La plupart des ouvrages de *référence* ne sont pas écrits dans cet objectif, mais, dans celui-ci, il s'agit d'apprendre. C'est pourquoi vous verrez les mêmes concepts abordés plusieurs fois.

Les exemples de code sont aussi courts que possible.

Nos lecteurs nous disent à quel point il est frustrant de devoir parcourir 200 lignes de code pour trouver les deux instructions qu'ils ont besoin de comprendre. Ici, la plupart des exemples sont présentés dans le contexte le plus réduit possible, pour que la partie que vous voulez apprendre demeure simple et claire. Ne vous attendez donc pas à ce que le code soit robuste, ni même complet. Les exemples sont spécifiquement conçus pour l'apprentissage et ne sont pas toujours pleinement fonctionnels.

Dans certains cas, nous n'avons pas inclus toutes les instructions import nécessaires : si vous programmez en Java, nous supposons que vous savez par exemple qu'ArrayList se trouve dans java.util. Si ces instructions ne font pas partie du noyau de l'API J2SE normale, nous le mentionnons. Nous avons également publié tout le code source sur Web pour que vous puissiez le télécharger. Vous le trouverez en anglais sur :

<http://www.wickedlysmart.com/headfirstdesignpatterns/code.html>

et en français sur <http://www.digitbooks.fr/catalogue/9782815000031.html>

De même, pour nous concentrer sur l'aspect pédagogique du code, nous n'avons pas placé nos classes dans des packages (autrement dit, elles sont toutes dans le package Java par défaut). Nous ne recommandons pas cette pratique dans le monde réel. Lorsque vous téléchargerez nos exemples de code, vous constaterez que toutes les classes sont dans des packages.

Les exercices « Muscler vos neurones » n'ont pas de réponse.

Pour certains, il n'existe pas de « bonne réponse ». Pour d'autres, c'est à vous qu'il appartient de décider si vos réponses sont justes ou non. Dans certains de ces exercices, vous trouverez des indications destinées à vous mettre dans la bonne direction.

Terminologie de l'édition française

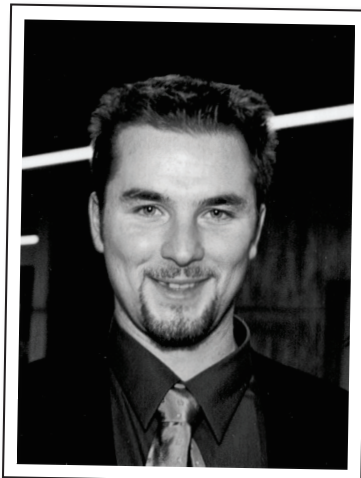
Dans cette édition, nous avons suivi pour l'essentiel la terminologie de Jean-Marie Lasvergères dans sa traduction de l'ouvrage d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley 1995, parue sous le titre *Design patterns, catalogue de modèles de conception réutilisables*, Paris, Vuibert, 1999. Nous avons notamment repris les noms des patterns, leurs définitions « officielles » et les noms de leurs composants dans les diagrammes de classes. Vous trouverez néanmoins quelques exceptions à cette règle, notamment en ce qui concerne le pattern Proxy, et certains noms de classes comme Créateur pour Facteur (pattern Fabrication) ou Cible pour But (pattern Adaptateur), leur emploi étant devenu plus courant. De même, nous avons adapté certains noms de méthode pour respecter les conventions de nommage Java.

Éditeurs techniques

Jef Cumps



Valentin Crettaz



Barney Marispini



Ike Van Atta ↘



Leader sans peur
et sans reproche de
l'équipe de révision
de Design patterns
Tête la première.



Johannes deJong ↗



↖
Mark Spritzler



Dirk Schreckmann

Jason Menard





Philippe Maquet

À la mémoire de Philippe Maquet

1960-2004

Ton extraordinaire expertise technique, ton enthousiasme sans faille et ton profond respect des apprenants nous inspireront toujours

Nous ne t'oublierons jamais.

Remerciements

Chez O'Reilly :

Nos remerciements les plus chaleureux à **Mike Loukides** qui est à l'origine du projet et a contribué à transformer le concept « Tête la première » en collection. Et un grand merci à la force motrice derrière Tête la première, **Tim O'Reilly**. Merci à la « maman de la collection », la talentueuse **Kyle Hart**, à la star du rock and roll **Ellie Volkhausen** pour sa couverture inspirée et à **Colleen Gorman** pour sa préparation de copie impitoyable. Enfin, merci à **Mike Hendrickson** pour s'être fait le champion de cet ouvrage et pour avoir construit l'équipe.

Nos intrépides relecteurs :

Nous sommes extrêmement reconnaissants au directeur de la révision technique, **Johannes de Jong**, Tu es notre héros, Johannes. Et nous ne dirons jamais assez à quel point les contributions du co-responsable de l'équipe de réviseurs de **Javaranch**, feu **Philippe Maquet**, a été précieuse. Il a illuminé à lui tout seul la vie de milliers de développeurs et l'impact qu'il a eu sur leur existence (et les nôtres) sera éternel.

Jef Cumps est terriblement doué pour détecter les problèmes dans les premiers jets, et il a fait encore une fois une énorme différence. Merci Jef ! **Valentin Crettaz** (tenant de la programmation orientée aspect), qui a été avec nous depuis le premier Tête la première, a prouvé (comme toujours) à quel point son expertise technique et son intuition étaient précieuses. Tu es étonnant, Valentin (mais laisse tomber cette cravate).

Deux nouveaux venus dans l'équipe de révision de TLP, Barney Marispini et Ike Van Atta, ont effectué un travail remarquable sur le livre — vous nous avez fourni un feedback vraiment crucial. Merci d'avoir rejoint l'équipe.

Les modérateurs et gourous de Javaranch, **Mark Spritzler**, **Jason Menard**, **Dirk Schreckmann**, **Thomas Paul** et **Margarita Isaeva**, nous ont également fourni une excellente aide technique. Et, comme toujours, un merci spécial au cow-boy en chef de javaranch.com, **Paul Wheaton**.

Merci aux finalistes du concours de Javaranch : « Votez pour la couverture de Design patterns Tête la première ». Si Brewster a gagné en soumettant l'essai qui nous a persuadés de choisir la jeune femme que vous voyez sur la couverture. Les autres finalistes étaient Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni et Jeff Fisher.

Ce n'est pas fini*

De la part d'Eric et d'Elisabeth

Écrire un livre « Tête la première » ressemble à une équipée sauvage accompagnée par deux guides étonnants, Kathy Sierra et **Bert Bates**. Avec Kathy et Bert, on jette toutes les conventions aux orties et on pénètre dans un monde dans lequel on raconte des histoires, on se préoccupe de théorie de l'apprentissage, de sciences cognitives et de culture pop, et où le lecteur joue toujours le premier rôle. Merci à vous deux de nous avoir permis d'entrer dans cet univers étonnant : nous espérons avoir fait honneur au concept Tête la première. Sérieusement, c'était une expérience hors du commun. Merci pour vos précieux conseils, pour nous avoir poussés à aller de l'avant, et, par-dessus tout, pour nous avoir fait confiance (avec votre bébé). Vous êtes sûrement « terriblement smart » et vous êtes aussi les trentagénaires les plus cools de notre connaissance.

Un grand merci à **Mike Loukides** et **Mike Hendrickson**. Mike L. nous a accompagnés à chaque étape du chemin. Mike, ton feedback perspicace nous a aidés à façonner le livre et tes encouragements nous ont aidés à continuer. Mike H., merci pour avoir persisté pendant cinq ans à nous convaincre d'écrire un livre sur les patterns. Nous avons fini par le faire, et nous sommes heureux d'avoir attendu Tête la première.

Un merci très spécial à **Erich Gamma**, qui a fait bien plus que répondre à l'appel du devoir en relisant ce livre (il en a même emporté une première version en vacances). Erich, ton intérêt nous a inspirés et ta relecture technique exhaustive l'a amélioré sans commune mesure. Merci également à toute la Bande des quatre pour leur soutien et leur intérêt et pour nous avoir rendu visite à Objectville. Nous sommes également redevables à **Ward Cunningham** et à la communauté des patterns qui ont créé le Portland Pattern Repository – une ressource qui nous a été indispensable lorsque nous écrivions ce livre.

Il faut un village pour écrire un ouvrage technique. **Bill Pugh** et **Ken Arnold** nous ont donné leur avis d'experts sur Singleton. **Joshua Marinacci** nous a fournis d'excellents conseils et astuces sur Swing. L'article de **John Brewer**, Why a Duck?, nous a suggéré l'idée du simulateur de canards (et nous sommes heureux qu'il aime lui aussi ces volatiles). **Dan Friedman** a inspiré l'exemple du Petit Singleton. **Daniel Steinberg** a joué le rôle d'« agent de liaison technique » et de réseau de soutien émotionnel. Et merci à **James Dempsey** de chez Apple pour nous avoir autorisés à utiliser sa chanson sur MVC.

Enfin, nous remercions personnellement l'équipe de réviseurs de **Javaranch** pour leurs remarques de premier ordre et leur soutien chaleureux. Nous vous devons beaucoup plus que vous ne le pensez.

De la part de Kathy et Bert

Nous aimerions remercier Mike Hendrickson d'avoir trouvé Eric et Elisabeth... mais cela nous est impossible. À cause d'eux, nous avons découvert (à notre grande horreur) que nous n'étions pas les seuls à pouvoir écrire un livre Tête la première ;-). Mais si les lecteurs veulent croire que ce sont réellement Kathy et Bert qui ont fait les trucs sympas dans ce livre, qui sommes-nous pour les détromper ?

De la part de Marie-Cécile

La traduction technique est souvent un exercice assez austère. Merci à Dominique Buraud de m'avoir permis de prendre beaucoup de plaisir à traduire les deux premiers volumes de la collection Tête la Première.

Et de même qu'il faut un village pour écrire un ouvrage technique, il faut un village pour le traduire. Merci à tous ceux qui m'ont fourni pour ces deux livres explications et suggestions sur Java et les Design patterns, mais aussi sur des sujets aussi variés que la musique électronique, le style cyberpunk, le bon usage du point-virgule, la fabrication des pizzas ou les tournures idiomatiques : Luc Carité, Christian Cler, Marie-Josée Keller et Fred Kolinski, Violette Kubler, David et Béatrice Morton, Françoise Pougeol, Catherine Pougeol et Lilo de chez Pizza Clip.

Et merci enfin à **Michel Beteta** et **Frédéric Laurent** pour la précision de leur relecture.

*La raison de tous ces remerciements est que nous testons la théorie selon laquelle toute personne mentionnée dans un livre en achètera au moins un exemplaire, voire plusieurs si elle en fait part à sa famille et à ses amis. Si vous voulez figurer dans les remerciements de notre prochain livre et si vous avez une grande famille, écrivez-nous.

1 Introduction aux Design Patterns

Bienvenue aux Design Patterns

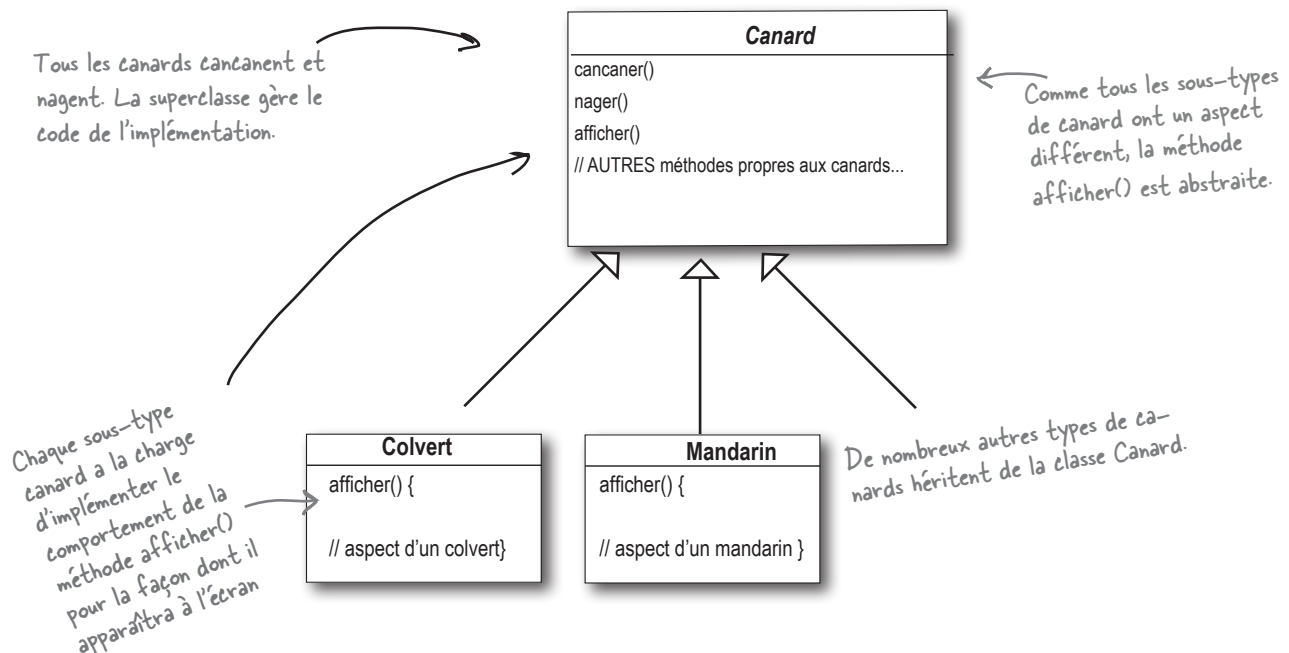


Maintenant
qu'on vit à Objectville, il faut
qu'on se mette aux Design Patterns...
Tout le monde ne parle que de ça.
On va bientôt être le clou de la soirée
patterns que Jacques et Lucie
organisent tous les mercredis !

Quelqu'un a déjà résolu vos problèmes. Dans ce chapitre, vous allez apprendre comment (et pourquoi) exploiter l'expérience et les leçons tirées par d'autres développeurs qui ont déjà suivi le même chemin, rencontré les mêmes problèmes de conception et survécu au voyage. Nous allons voir l'usage et les avantages des design patterns, revoir quelques principes fondamentaux de la conception OO et étudier un exemple de fonctionnement d'un pattern. La meilleure façon d'utiliser un pattern est de le *charger dans votre cerveau* puis de *reconnaître* les points de vos conceptions et des applications existantes auxquels vous pouvez *les appliquer*. Au lieu de réutiliser du *code*, les patterns vous permettent de réutiliser de l'*expérience*.

Tout a commencé par une simple application, SuperCanard

Joël travaille pour une société qui a rencontré un énorme succès avec un jeu de simulation de mare aux canards, *SuperCanard*. Le jeu affiche toutes sortes de canards qui nagent et émettent des sons. Les premiers concepteurs du système ont utilisé des techniques OO standard et créé une superclasse Canard dont tous les autres types de canards héritent.

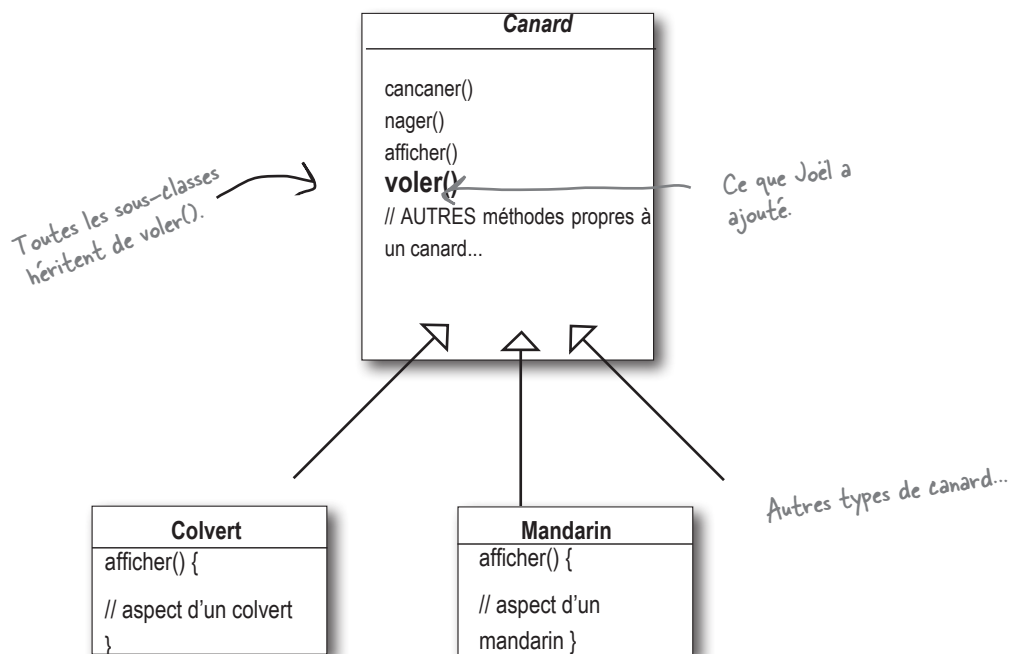


L'an passé, la société a subi de plus en plus de pression de la part de la concurrence. À l'issue d'une semaine de séminaire résidentiel consacré au brainstorming et au golf, ses dirigeants ont pensé qu'il était temps de lancer une grande innovation. Il leur faut maintenant quelque chose de *réellement* impressionnant à présenter à la réunion des actionnaires qui aura lieu aux Baléares *la semaine prochaine*.

Maintenant, nous voulons que les canard **VOLENT**

Les dirigeants ont décidé que des canards volants étaient exactement ce qu'il fallait pour battre la concurrence à plate couture. Naturellement, le responsable de Joël leur a dit que celui-ci n'aurait aucun problème pour bricoler quelque chose en une semaine. "Après tout", a-t-il dit, "Joël est un programmeur OO... ça ne doit pas être si difficile !"

Il suffit que j'ajoute une méthode `voler()` dans la classe `Canard` et tous les autres canards en hériteront. L'heure est venue de montrer mon vrai génie.



Il y a quelque chose qui ne va pas

Joël, je suis à la réunion des actionnaires. On vient de passer la démo et il y a plein de canards en plastique qui volent dans tout l'écran. C'est une plaisanterie ou quoi ? Tu as peut-être envie de passer du temps sur Monster.fr ?



Que s'est-il passé ?

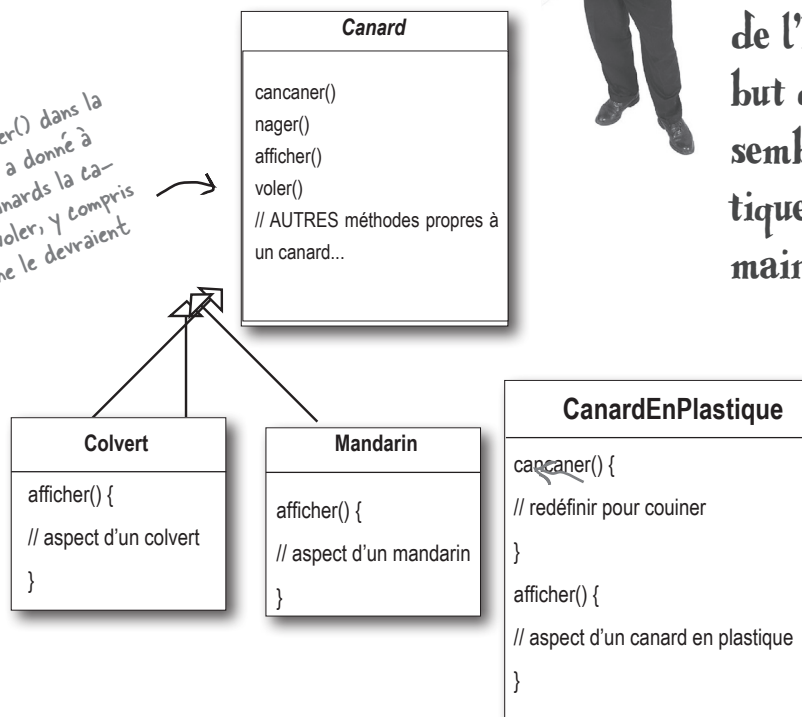
Joël a oublié que *toutes* les sous-classes de Canard ne doivent pas *voler*. Quand il a ajouté le nouveau comportement à la superclasse Canard, il a également ajouté un comportement qui n'était pas approprié à certaines de ses sous-classes. Maintenant, il a des objets volants inanimés dans son programme SuperCanard.

Une mise à jour locale du code a provoqué un effet de bord global (des canards en plastique qui volent) !

O K ,
on dirait qu'il y a un léger défaut dans ma conception. Je ne vois pas pourquoi ils ne peuvent pas appeler ça une "fonctionnalité". C'est plutôt marrant...


Ce qu'il prenait pour une super application de l'héritage dans un but de réutilisation semble plus problématique quand il s'agit de maintenance.

En plaçant voler() dans la superclasse, il a donné à TOUS les canards la capacité de voler, y compris ceux qui ne le devraient pas.




Puisque les canards en plastique ne volent pas, cancaner() est redéfinie pour couiner.


Joël réfléchit à l'héritage...



Je pourrais toujours me contenter de redéfinir la méthode voler() dans CanardEnPlastique, comme je le fais pour cancaner()...




Je pourrais toujours me contenter de redéfinir la méthode voler() dans CanardEnPlastique, comme je le fais pour cancaner()...



```

CanardEnPlastique

cancaner() { // couiner }
afficher () { // canard en plastique }
voler() {
    // redéfinir pour ne rien faire
}
        
```



Voici une autre classe de la hiérarchie. Remarquez que les leurres ne volent pas plus que les canards en plastique. En outre, ils ne cancanent pas non plus.

```

Leurre

cancaner() {
    // redéfinir pour ne rien faire
}

afficher() { // leurre}

voler() {
    // redéfinir pour ne rien faire
}
        
```



À vos crayons

Dans la liste ci-après, quels sont les inconvénients à utiliser *l'héritage* pour définir le comportement de Canard ? (Plusieurs choix possibles.)

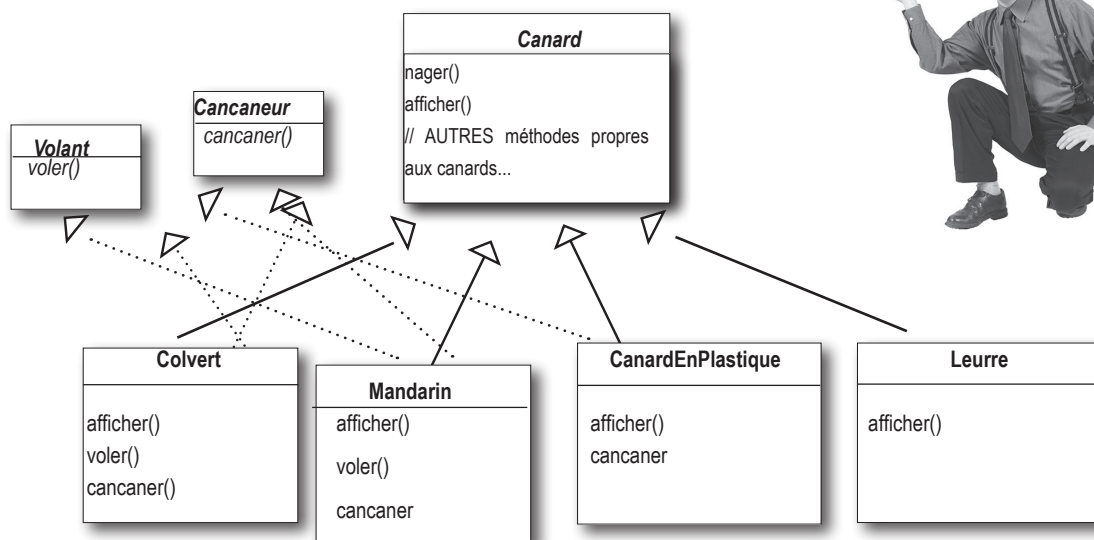
- ☐ A. Le code est dupliqué entre les sous-classes.
- ☐ B. Les changements de comportement au moment de l'exécution sont difficiles
- ☐ C. Nous ne pouvons pas avoir de canards qui dansent.
- ☐ D. Il est difficile de connaître tous les comportements des canards.
- ☐ E. Les canards ne peuvent pas voler et cancaner en même temps.
- ☐ F. Les modifications peuvent affecter involontairement d'autres canards.

Et si nous utilisions une interface ?

Joël s'est rendu compte que l'héritage n'était probablement pas la réponse : il vient de recevoir un mémo annonçant que les dirigeants ont décidé de réactualiser le produit tous les six mois (ils n'ont pas encore décidé comment). Il sait que les spécifications vont changer en permanence et qu'il va peut-être être obligé de redéfinir `voler()` et `cancaner()` pour toute sous-classe de `Canard` qui sera ajoutée au programme... *ad vitam aeternam*.

Il a donc besoin d'un moyen plus sain pour que seuls *certain*s types de canard (mais pas *tous*) puissent voler ou cancaner.

Je pourrais extraire la méthode `voler()` de la superclasse `Canard`, et créer une **interface** `Volant()` qui aurait une méthode `voler()`. Ainsi, seuls les canards qui sont censés voler implémenteront cette interface et auront une méthode `voler()`... et je pourrais aussi créer une interface `Cancaneur` par la même occasion, puisque tous les canards ne cancanent pas.



Et VOUS ? Que pensez-vous de cette conception ?

C'est, comment dire... l'idée la plus stupide que tu aies jamais eue. Et le code dupliqué ? Si tu pensais que redéfinir quelques méthodes était une mauvaise idée, qu'est-ce que ça va être quand tu devras modifier un peu le comportement de vol dans les 48 sous-classes de Canard qui volent ?!

Que feriez-vous à la place de Joël ?



Nous savons que *toutes* les sous-classes ne doivent *pas* avoir de comportement qui permette aux canards de voler ou de cancaner. L'héritage ne constitue donc pas la bonne réponse. Mais si créer des sous-classes qui implémentent Volant et/ou Cancaneur résout une *partie* du problème (pas de canards en plastique qui se promènent malencontreusement dans l'écran), cela détruit complètement la possibilité de réutiliser le code pour ces comportements et ne fait que créer un *autre* cauchemar sur le plan de la maintenance. De plus, tous les canards qui *doivent* voler ne volent peut-être pas de la même façon...

À ce stade, vous attendez peut-être qu'un design pattern arrive sur son cheval blanc et vous sauve la mise. Mais où serait le plaisir ? Non, nous allons essayer de trouver une solution à l'ancienne – *en appliquant de bons principes de conception OO*.



Ne serait-ce pas merveilleux s'il y avait un moyen de construire des logiciels de sorte que les modifications aient le moins d'impact possible sur le code existant ? Cela nous permettrait de passer moins de temps à retravailler le code et plus de temps à inventer des trucs plus cools...

La seule et unique constante du développement

Bien, quelle est la seule chose sur laquelle vous puissiez toujours compter en tant que développeur ?

Indépendamment de l'endroit où vous travaillez, de l'application que vous développez ou du langage dans lequel vous programmez, quelle est la seule vraie constante qui vous accompagnera toujours ?

LE CHANGEMENT

(prenez un miroir pour voir la réponse)

Quel que soit le soin que vous ayez apporté à la conception d'une application, elle devra prendre de l'ampleur et évoluer au fil du temps. Sinon, elle *mourra*.



À vos crayons

De nombreux facteurs peuvent motiver le changement. Énumérez quelles pourraient être les raisons de modifier le code de vos applications (nous en avons listé deux pour vous aider à démarrer).

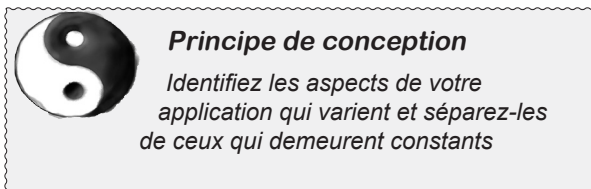
Mes clients ou mes utilisateurs décident qu'ils veulent autre chose ou qu'ils ont besoin d'une nouvelle fonctionnalité.

Mon entreprise a décidé qu'elle allait changer de système gestion de bases de données et qu'elle allait également acheter ses données chez un autre fournisseur dont le format est différent. Argh !

Attaquons le problème...

Nous savons donc que le recours à l'héritage n'a pas été un franc succès, puisque le comportement des canards ne cesse de varier d'une sous-classe à l'autre et que ce comportement n'est pas approprié à toutes les sous-classes. Les interfaces Volant et Cancaneur semblaient tout d'abord prometteuses – seuls les canards qui volent implémenteraient Volant, etc. – sauf que les interfaces Java ne contiennent pas de code : il n'y a donc pas de code réutilisable. Chaque fois que vous voulez changer un comportement, vous êtes obligé de le rechercher et de le modifier dans toutes les sous-classes dans lesquelles il est défini, en introduisant probablement quelques nouveaux bogues en cours de route !

Heureusement, il existe un principe de conception fait sur mesure pour cette situation.



↖ Le premier de nos nombreux principes de conception. Nous leur consacrerons plus de temps tout au long de cet ouvrage.

Autrement dit, si l'un des aspects de votre code est susceptible de changer, par exemple avec chaque nouvelle exigence, vous savez que vous êtes face à un comportement qu'il faut extraire et isoler de tout ce qui ne change pas.

Voici une autre façon de formuler ce principe : ***extraire les parties variables et les encapsuler vous permettra plus tard de les modifier ou de les augmenter sans affecter celles qui ne varient pas.***

Malgré sa simplicité, ce concept constitue la base de presque tous les design patterns. Tous les patterns fournissent un moyen de permettre à *une partie d'un système de varier indépendamment de toutes les autres.*

Cela dit, il est temps d'extraire les comportements de canard des classes Canard !

Extrayez ce qui varie et « encapsulez-le » pour ne pas affecter le reste de votre code.

Résultat ? Les modifications du code entraînent moins de conséquences inattendues et vos systèmes sont plus souples !

Séparer ce qui change de ce qui reste identique

Par où commencer ? Pour l'instant, en dehors des problèmes de `voler()` et de `cancaner()`, la classe `Canard` fonctionne bien et ne contient rien qui semble devoir varier ou changer fréquemment. À part quelques légères modifications, nous allons donc la laisser pratiquement telle quelle.

Maintenant, pour séparer les « parties qui changent de celles qui restent identiques », nous allons créer deux *ensembles* de classes (totalement distinctes de `Canard`), l'un pour *voler* et l'autre pour *cancaner*. Chaque ensemble de classes contiendra toutes les implémentations de leur comportement respectif. Par exemple, nous aurons une classe qui implémente le *cancanement*, une autre le *couinement* et une autre le *silence*.

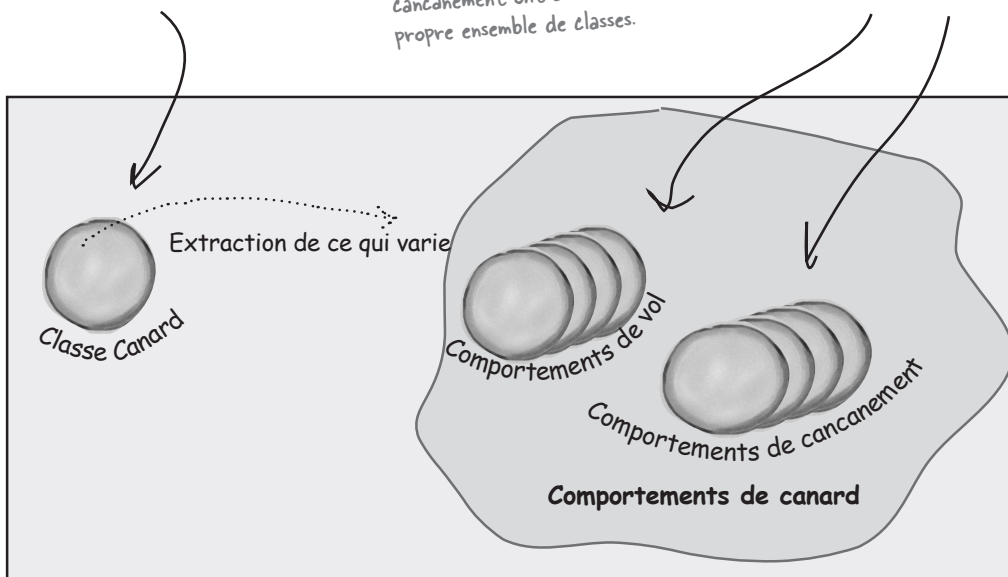
Nous savons que `voler()` et `cancaner()` sont les parties de la classe `Canard` qui varient d'un canard à l'autre.

Pour séparer ces comportements de la classe `Canard`, nous extrayons ces deux méthodes de la classe et nous créons un nouvel ensemble de classes pour représenter chaque comportement.

La classe `Canard` est toujours la superclasse de tous les canards, mais nous extrayons les comportements de vol et de cancanement et nous les plaçons dans une autre structure de classes.

Maintenant, le vol et le cancanement ont chacun leur propre ensemble de classes.

C'est là que vont résider les différentes implémentations des comportements

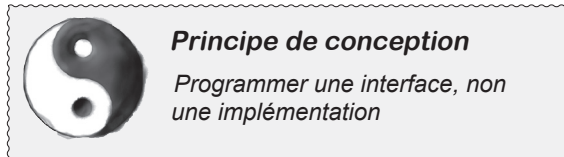


Conception des comportements de canard

Comment allons-nous procéder pour concevoir les classes qui implémentent les deux types de comportements ?

Nous voulons conserver une certaine souplesse. Après tout, c'est d'abord la rigidité du comportement des canards qui nous a causé des ennuis. Et nous savons que nous voulons *affecter* des comportements aux instances des classes Canard. Par exemple instancier un nouvel objet Colvert et l'initialiser avec un *type* spécifique de comportement de vol. Et, pendant que nous y sommes, pourquoi ne pas faire en sorte de pouvoir modifier le comportement d'un canard dynamiquement ? Autrement dit, inclure des méthodes *set* dans les classes Canard pour pouvoir *modifier* la façon de voler du Colvert *au moment de l'exécution*.

Avec ces objectifs en tête, voyons notre deuxième principe de conception :



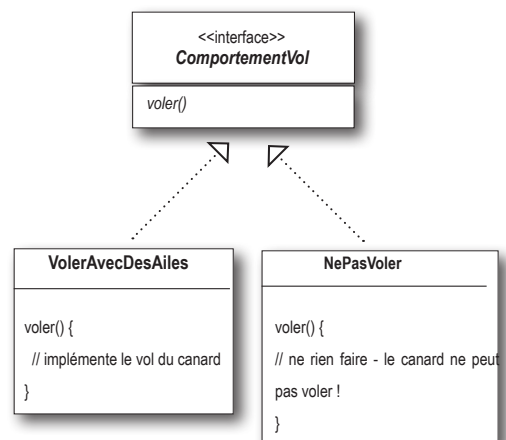
Nous allons utiliser une interface pour représenter chaque comportement – par exemple ComportementVol et ComportementCancan – et chaque implémentation d'un *comportement* implémentera l'une de ces interfaces. Cette fois, ce ne sont pas les classes *Canard* qui implémenteront les interfaces pour voler et cancaner. En lieu et place, nous allons créer un ensemble de classes dont la seule raison d'être est de représenter un comportement (par exemple « couiner »), et c'est la classe *comportementale*, et non la classe Canard, qui implémentera l'interface comportementale.

Ceci contraste avec notre façon antérieure de procéder, dans laquelle un comportement provenait d'une implémentation concrète dans la superclasse Canard ou d'une implémentation spécialisée dans la sous-classe elle-même. Dans les deux cas, nous nous reposons sur une *implémentation*. Nous étions contraints à utiliser cette implémentation spécifique et il n'y avait aucun moyen de modifier le comportement (à part écrire plus de code).

Avec notre nouvelle conception, les sous-classes de Canard auront un comportement représenté par une *interface* (ComportementVol et ComportementCancan), si bien que l'*implémentation* réelle du comportement (autrement dit le comportement spécifique concret codé dans la classe qui implémente ComportementVol ou ComportementCancan) ne sera pas enfermé dans la sous-classe de Canard.

Désormais, les comportements de Canard réviseront dans une classe distincte – une classe qui implémente une interface comportementale particulière.

Ainsi, les classes Canard n'auront besoin de connaître aucun détail de l'implémentation de leur propre comportement.





Je ne vois pas pourquoi il faut utiliser une interface pour ComportementVol. On peut faire la même chose avec une superclasse abstraite. Est-ce que ce n'est pas tout l'intérêt du polymorphisme ?

« Programmer une interface » signifie en réalité « Programmer un supertype ».

Le mot *interface* a ici un double sens. Il y a le *concept* d'interface, mais aussi la construction Java **interface**. Vous pouvez *programmer une interface* sans réellement utiliser une **interface** Java. L'idée est d'exploiter le polymorphisme en programmant un supertype pour que l'objet réel à l'exécution ne soit pas enfermé dans le code. Et nous pouvons reformuler « programmer un supertype » ainsi :

« le type déclaré des variables doit être un supertype, généralement une interface ou une classe abstraite. Ainsi, les objets affectés à ces variables peuvent être n'importe quelle implémentation concrète du supertype, ce qui signifie que la classe qui les déclare n'a pas besoin de savoir quels sont les types des objets réels ! »

Vous savez déjà probablement tout cela, mais, juste pour être sûrs que nous parlons de la même chose, voici un exemple simple d'utilisation d'un type polymorphe – imaginez une classe abstraite *Animal*, avec deux implémentations concrètes, *Chien* et *Chat*.

Programmer une implémentation donnerait le code suivant :

```
Chien c = new Chien ();
c.aboyer();
```

Déclarer la variable "c" de type Chien (une implémentation concrète d'Animal) nous oblige à coder une implémentation concrète.

Mais **programmer une interface ou un supertype** donnerait ceci :

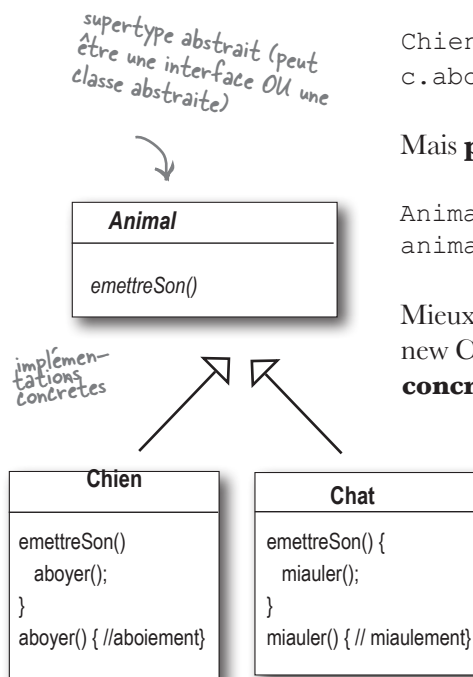
```
Animal animal = new Chien ();
animal.emettreSon();
```

Nous savons que c'est un Chien, mais nous pouvons maintenant utiliser la référence animal de manière polymorphe.

Mieux encore, au lieu de coder en dur l'instanciation du sous-type (comme `new Chien ()`) dans le code, **affectez l'objet de l'implémentation concrète au moment de l'exécution** :

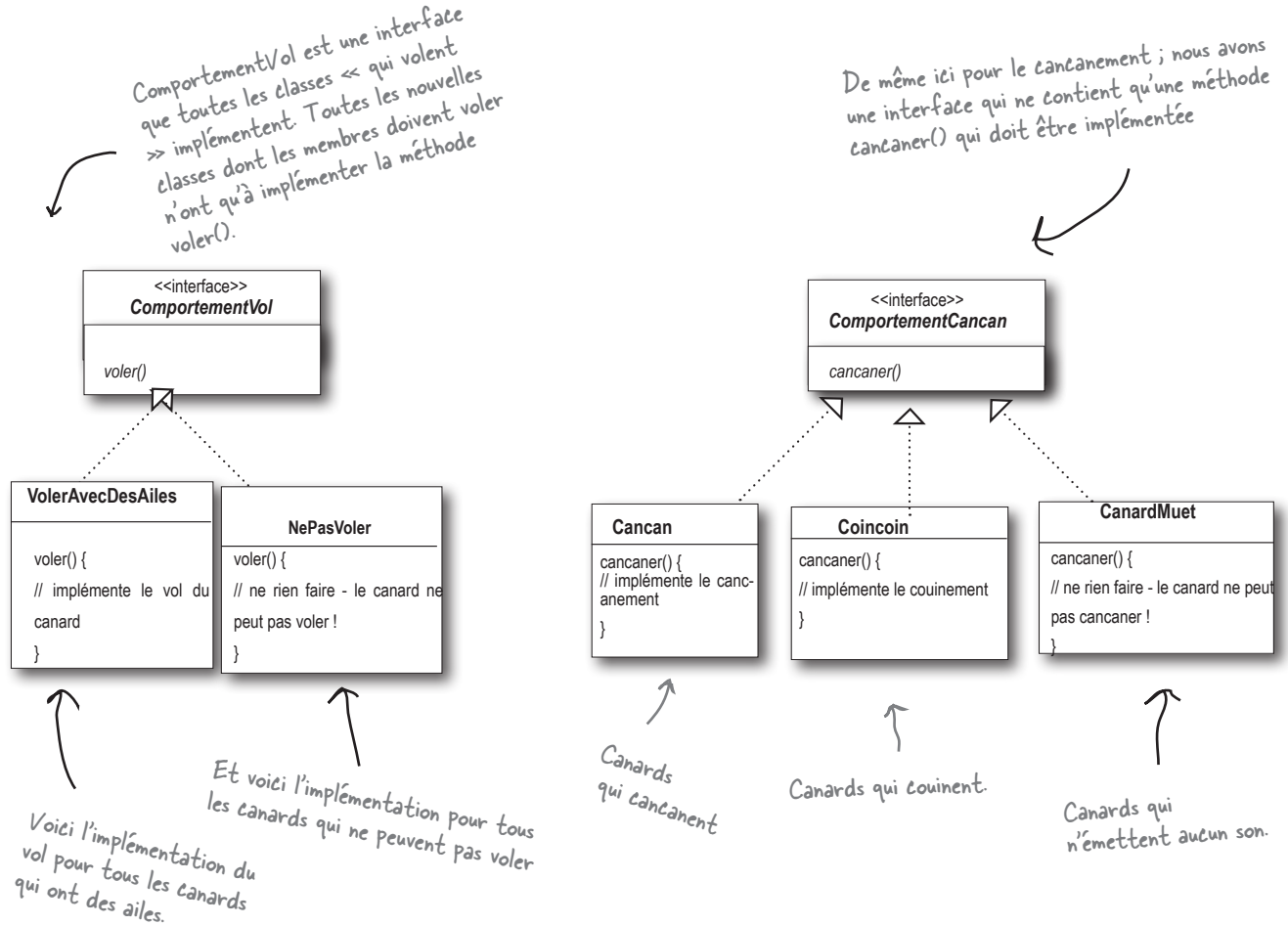
```
a = getAnimal();
a.emettreSon();
```

Nous ne savons pas QUEL EST le sous-type réel de l'animal... tout ce qui nous intéresse, c'est qu'il sait comment répondre à `emettreSon()`.



Implémenter les comportements des canards

Nous avons ici deux interfaces, ComportementVol et ComportementCancan, ainsi que les classes correspondantes qui implémentent chaque comportement concret :



Avec cette conception, les autres types d'objets peuvent réutiliser nos comportements de vol et de cancanement parce que ces comportements ne sont plus cachés dans nos classes Canard !

Et nous pouvons ajouter de nouveaux comportements sans modifier aucune des classes comportementales existantes ni toucher à aucune des classes Canard qui utilisent les comportements de vol.

Nous obtenons ainsi les avantages de la RÉUTILISATION sans la surcharge qui accompagne l'héritage

Il n'y a pas de Questions Stupides

Q: Est-ce que je dois toujours implémenter mon application d'abord, voir les éléments qui changent, puis revenir en arrière pour séparer et encapsuler ces éléments ?

R: Pas toujours. Quand vous concevez une application, vous anticipez souvent les points de variation, puis vous avancez et vous construisez le code de manière suffisamment souple pour pouvoir les gérer. Vous verrez que vous pouvez appliquer des principes et des patterns à n'importe quel stade du cycle de vie du développement.

Q: Est-ce qu'on devrait aussi transformer Canard en interface ?

R: Pas dans ce cas. Vous verrez qu'une fois que nous aurons tout assemblé, nous serons contents de disposer d'une classe concrète Canard et de canards spécifiques, comme Colvert, qui héritent des propriétés et des méthodes communes. Maintenant que nous avons enlevé de Canard tout ce qui varie, nous avons tous les avantages de cette structure sans ses inconvénients.

Q: Cela fait vraiment bizarre d'avoir une classe qui n'est qu'un comportement. Est-ce que les classes ne sont pas censées représenter des entités ? Est-ce qu'elles ne doivent pas avoir un état ET un comportement ?

R: Dans un système OO, oui, les classes représentent des entités qui ont généralement un état (des variables d'instance) et des méthodes. Dans ce cas, l'entité se trouve être un comportement. Mais même un comportement peut avoir un état et des méthodes ; un comportement de vol peut avoir des variables d'instance qui représentent les attributs du vol (nombre de battements d'ailes par minute, altitude et vitesse maximales, etc.).

À vos crayons

En utilisant notre nouvelle conception, que feriez-vous pour ajouter la propulsion à réaction à l'application SuperCanard ?

Voyez-vous une classe qui pourrait utiliser le comportement de Cancan et qui n'est pas un canard ?

1) Créer une classe VolARéaction qui implémente l'interface ComportementVol.
2) Par exemple un appeau (un instrument qui imite le cri du canard).

Réponses :

Intégrer les comportements des canards

La clé est qu'un Canard va maintenant déléguer ses comportements au lieu d'utiliser les méthodes voler() et cancaner() définies dans la classe Canard (ou une sous-classe).

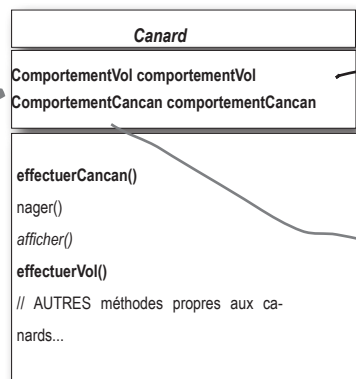
Voici comment :

1. Nous allons d'abord ajouter à la classe Canard deux variables d'instance nommées *comportementVol* et *comportementCancan*, qui sont déclarées du type de l'interface (non du type de l'implémentation concrète). Chaque objet Canard affectera ces variables de manière polymorphe pour référencer le type de comportement *spécifique* qu'il aimerait avoir à l'exécution (VolerAvecDesAiles, Coincoin, etc.).

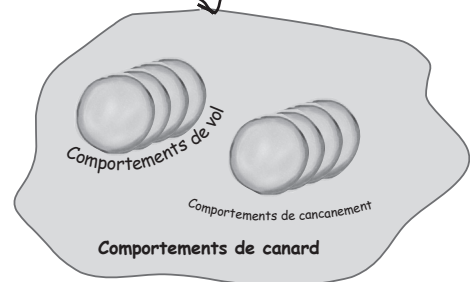
Nous allons également ôter les méthodes voler() et cancaner() de la classe Canard (et de toutes ses sous-classes) puisque nous avons transféré ces comportements dans les classes ComportementVol et ComportementCancan. Nous allons remplacer voler() et cancaner() dans la classe Canard par deux méthodes similaires, nommées effectuerVol() et effectuerCancan(). Vous allez bientôt voir comment elles fonctionnent.

Les variables comportementales sont déclarées du type de l'INTERFACE qui gère le comportement

Ces méthodes remplacent voler() et cancaner().



Lors de l'exécution, les variables d'instance contiennent une référence à un comportement spécifique.



2. Implémentons maintenant effectuerCancan():

```
public class Canard {
    ComportementCancan comportementCancan;
    // autres variables

    public void effectuerCancan() {
        comportementCancan.cancaner();
    }
}
```

Chaque Canard a une référence à quelque chose qui implémente l'interface ComportementCancan.

Au lieu de gérer son cancanement lui-même, l'objet Canard délègue ce comportement à l'objet référencé par comportementCancan. Les variables du comportement sont déclarées du type de l'INTERFACE comportementale.

Rien de plus simple, n'est-ce pas ? Pour cancaner, un Canard demande à l'objet référencé par comportementCancan de le faire à sa place.

Dans cette partie du code, peu nous importe de quelle sorte d'objet il s'agit. **Une seule chose nous intéresse : il sait cancaner !**

Suite de l'intégration...

3. Bien. Il est temps de s'occuper de la façon dont les **variables d'instance comportementVol et comportementCancan** sont affectées. Jetons un coup d'œil à la classe Colvert :

```
public class Colvert extends Canard {  
  
    public Colvert() {  
        comportementCancan = new Cancan();  
        comportementVol = new VolerAvecDesAiles();  
    }  
  
}
```

Puisqu'un Colvert utilise la classe Cancan pour cancaner, quand effectuerCancan() est appelée, la responsabilité du cancanement est déléguée à l'objet Cancan et nous obtenons un vrai cancan.

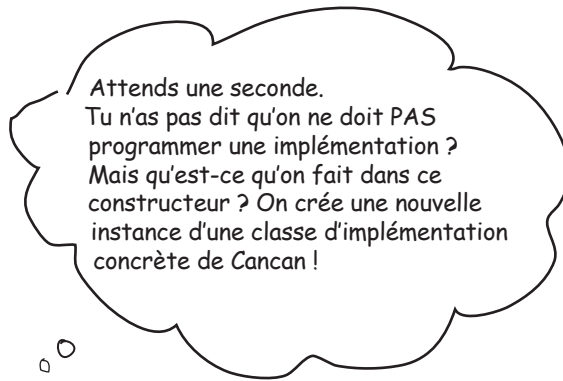
Et il utilise VolerAvecDesAiles comme type de ComportementVol.

Souvenez-vous que Colvert hérite les variables d'instance comportementCancan et comportementVol de la classe Canard.

```
    public void afficher() {  
        System.out.println("Je suis un vrai colvert");  
    }  
}
```

Le cancan de Colvert est un vrai **cancan** de canard vivant, pas un **coincoin** ni un **cancan muet**. Quel est le mécanisme ? Quand un Colvert est instancié, son constructeur initialise sa variable d'instance comportementCancan héritée en lui affectant une nouvelle instance de type Cancan (une classe d'implémentation concrète de ComportementCancan).

Et il en va de même pour le comportement de vol – le constructeur de Colvert ou de Mandarin initialise la variable d'instance comportementVol avec une instance de type VolerAvecDesAiles (une classe d'implémentation concrète de ComportementVol).



Bien vu. C'est exactement ce que nous faisons... *pour l'instant.*

Plus tard, nous aurons dans notre boîte à outils d'autres patterns qui nous permettront d'y remédier. Remarquez pourtant que si nous affectons *bien* les comportements aux classes concrètes (en instanciant une classe comportementale comme Cancan ou VolerAvecDesAiles et affectant l'instance à notre variable de référence comportementale), nous pourrions *facilement* changer cela au moment de l'exécution.

Nous avons toujours beaucoup de souplesse, mais la façon d'affecter les variables d'instance n'est pas très satisfaisante. Mais réfléchissez : puisque la variable d'instance comportementCancan est du type de l'interface, nous pourrions (par la magie du polymorphisme) affecter dynamiquement une instance d'une classe d'implémentation ComportementCancan différente au moment de l'exécution.

Faites une pause et demandez-vous comment vous implémenteriez un canard pour que son comportement change à l'exécution. (Vous verrez le code qui le permet dans quelques pages.)

Tester le code de Canard

1. Tapez et compilez le code de la classe Canard ci-après (Canard.java) et celui de la classe Colvert de la page 16 (Colvert.java).

```
public abstract class Canard {  
  
    ComportementVol comportementVol;  
    ComportementCancan comportementCancan;  
    public Canard() {  
    }  
  
    public abstract void afficher();  
  
    public void effectuerVol() {  
        comportementVol.voler();  
    }  
  
    public void effectuerCancan() {  
        comportementCancan.cancaner();  
    }  
  
    public void nager() {  
        System.out.println("Tous les canards flottent, même les leurres!");  
    }  
}
```

← Déclare deux variables de référence pour les types des interfaces comportementales. Toutes les sous-classes de Canard (dans le même package) en héritent.

← Délègue à la classe comportementale.

2. Tapez et compilez le code de l'interface ComportementVol (ComportementVol.java) et les deux classes d'implémentation comportementales (VolerAvecDesAiles.java et NePasVoler.java).

```
public interface  
ComportementVol {  
    public void voler();}
```

L'interface que toutes les classes comportementales << qui volent >> implémentent.

```
public class VolerAvecDesAiles implements ComportementVol {  
    public void voler() {  
        System.out.println("Je vole !!");  
    }  
}
```

Implémentation du comportement de vol pour les canards qui **VOLENT**...

```
public class NePasVoler implements ComportementVol {  
    public void voler() {  
        System.out.println("Je ne sais pas voler")  
    }  
}
```

Implémentation du comportement de vol pour les canards qui ne **VOLENT PAS** (comme les canards en plastique et les leurres).

Tester le code de Canard (suite)...

3. Tapez et compilez le code de l'interface ComportementCancan (ComportementCancan.java) et celui des trois classes d'implémentation comportementales (Cancan.java, CancanMuet.java et CoinCoin.java).

```
public interface ComportementCancan {
    public void cancaner() ;
}
```

```
public class Cancan implements ComportementCancan {
    public void cancaner() {
        System.out.println("Cancan");
    }
}
```

```
public class CancanMuet implements ComportementCancan {
    public void cancaner() {
        System.out.println("Silence");
    }
}
```

```
public class Coincoin implements ComportementCancan {
    public void cancaner() {
        System.out.println("Coincoin");
    }
}
```

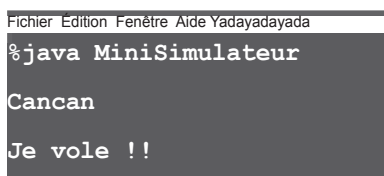
4. Tapez et compilez le code de la classe de test (MiniSimulateur.java).

```
public class MiniSimulateur {
    public static void main(String[] args) {
        Canard colvert = new Colvert();
        colvert.effectuerCancan();
        colvert.effectuerVol();
    }
}
```

Cette ligne appelle la méthode héritée `effectuerCancan()` de `Colvert`, qui délègue alors à `ComportementCancan` de l'objet (autrement dit appelle `cancaner()` sur la référence héritée `comportementCancan` du canard).

Puis nous faisons de même avec la méthode héritée `effectuerVol()` de `Colvert`.

5. Exécutez le code !



```
Fichier Edition Fenêtre Aide Yadayadayada
%java MiniSimulateur
Cancan
Je vole !!
```


Modifier le comportement dynamiquement

Quel dommage que nos canards possèdent tout ce potentiel de dynamisme et qu'ils ne l'utilisent pas !
Imaginez que vous vouliez fixer le type de comportement du canard *via* des méthodes set dans la sous-classe de Canard au lieu de l'initialiser dans le constructeur.

1. Ajoutez deux nouvelles méthodes à la classe Canard :

```
public void setComportementVol(ComportementVol cv) {
    comportementVol = cv;
}

public void setComportementCancan(ComportementCancan cc) {
    comportementCancan = cc;
}
```

Canard
ComportementVol comportementVol;
ComportementCancan comportementCancan;
nager()
afficher()
effectuerCancan()
effectuerVol()
setComportementVol()
setComportementCancan()
// AUTRES méthodes propres aux canards...

Nous pouvons appeler ces méthodes chaque fois que nous voulons modifier le comportement d'un canard à la volée.

note de l'éditeur : jeu de mots gratuit

2. Créez un nouveau type de Canard (PrototypeCanard.java).

```
public class PrototypeCanard extends Canard {
    public PrototypeCanard() {
        comportementVol = new NePasVoler();
        comportementCancan = new Cancan();
    }

    public void afficher() {
        System.out.println("Je suis un prototype de canard");
    }
}
```

Notre nouveau canard vient au monde...sans aucun moyen de voler.

3. Créez un nouveau type de ComportementVol (PropulsionAReaction.java).

```
public class PropulsionAReaction implements ComportementVol {
    public void voler() {
        System.out.println("Je vole avec un réacteur !");
    }
}
```

Qu'à cela ne tienne ! Nous créons un nouveau comportement de vol : la propulsion à réaction




4. Modifiez la classe de test (MiniSimulateur.java), ajoutez PrototypeCanard et munissez-le d'un réacteur.

```
public class MiniSimulateur {
    public static void main(String[] args) {
        Canard colvert = new Colvert();
        colvert.effectuerCancan();
        colvert.effectuerVol();
    }
}
```

```
Canard proto = new PrototypeCanard();
proto.effectuerVol();
proto.setComportementVol(new PropulsionAReaction());
proto.effectuerVol();
```

avant



Le premier appel de `effectuerVol()` délègue à l'objet `comportementVol` défini dans le constructeur de `PrototypeCanard`, qui est une instance de `NePasVoler`.

Ceci invoque la méthode `set` héritée du prototype, et ...voilà ! Le prototype est soudain doté d'une fonctionnalité de vol... à réaction

Si cela fonctionne, le canard a changé de comportement de vol dynamiquement ! Ce serait IMPOSSIBLE si l'implémentation résidait dans la classe `Canard`

5. Exécutez-le !

```
Fichier Édition Fenêtre Aide Yabadabadoo
%java MiniSimulateur
Cancan
Je vole !!
Je ne sais pas voler
Je vole avec un réacteur !
```



après

Pour modifier le comportement d'un canard au moment de l'exécution, il suffit d'appeler la méthode `set` correspondant à ce comportement.

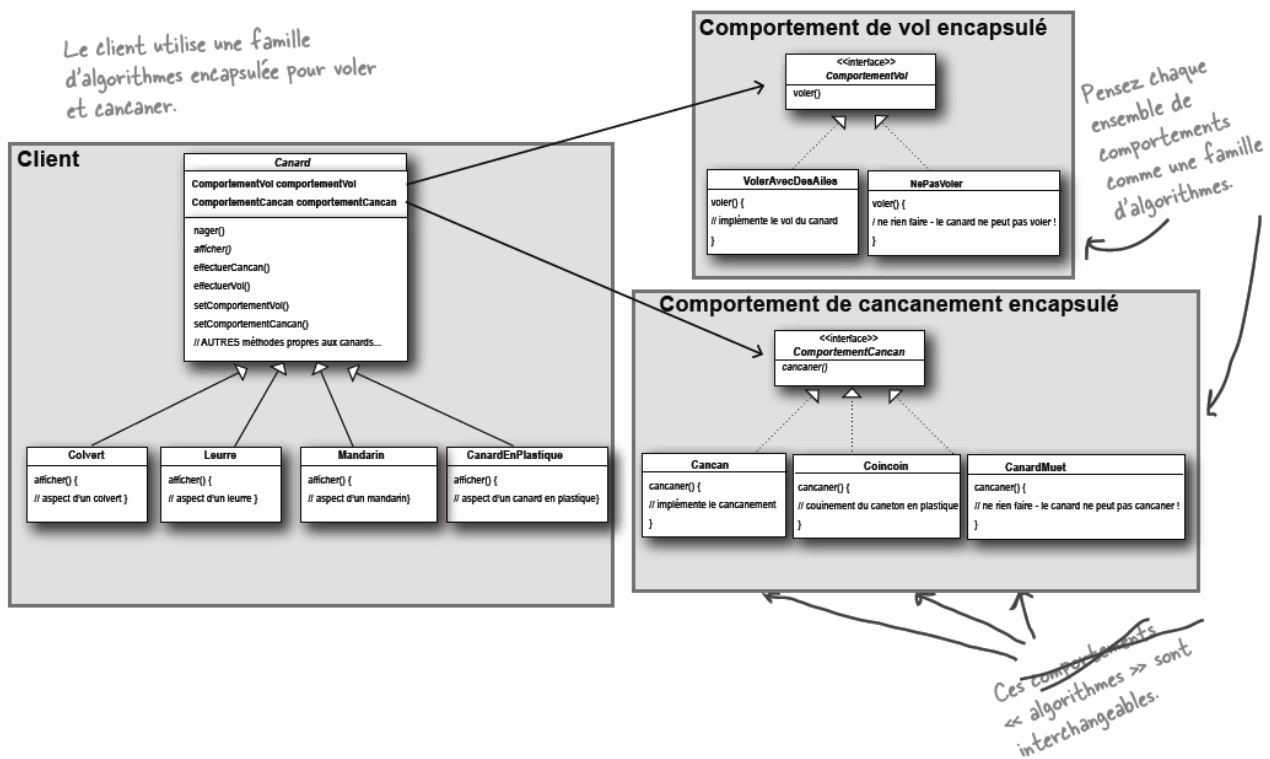
Vue d'ensemble des comportements encapsulés

Bien. Maintenant que nous avons plongé au cœur de la conception du simulateur, il est temps de venir respirer à la surface et de jeter un coup d'œil global.

Vous trouverez ci-après toute la structure de classes retravaillée. Nous avons tout ce qu'il nous faut : des canards qui dérivent de Canard, des comportements de vol qui implémentent ComportementVol et des comportements de cancanement qui implémentent ComportementCancan.

Remarquez aussi que nous avons commencé à décrire les choses un peu différemment. Au lieu de penser les comportements des canards comme un *ensemble de comportements*, nous allons commencer à les penser comme une *famille d'algorithmes*. Réfléchissez-y : dans la conception de SuperCanard, les algorithmes représentent ce qu'un canard ferait différemment (différentes façons de voler ou de cancaner), mais nous pourrions tout aussi bien utiliser les mêmes techniques pour un ensemble de classes qui implémenteraient les différentes façons de calculer la taxe d'habitation selon les différentes communes.

Soyez très attentif aux *relations* entre les classes. Prenez un crayon et notez la relation appropriée (EST-UN, A-UN et IMPLÉMENTE) sur chaque flèche du diagramme de classes.



A-UN peut être préférable à EST-UN

La relation A-UN est une relation intéressante : chaque canard a un ComportementVol et un ComportementCancan auxquels ils délègue le vol ou le cancanement.

Lorsque vous assemblez deux classes de la sorte, vous utilisez la **composition**. Au lieu d'hériter leur comportement, les canards l'obtiennent en étant *composés* avec le bon objet comportemental.

Cette technique est importante ; en fait, nous avons appliqué notre troisième principe de conception :



Principe de conception

Préférez la composition à l'héritage

Comme vous l'avez constaté, créer des systèmes en utilisant la composition procure beaucoup plus de souplesse. Non seulement cela vous permet d'encapsuler une famille d'algorithmes dans leur propre ensemble de classes, mais vous pouvez également **modifier le comportement au moment de l'exécution** tant que l'objet avec lequel vous composez implémente la bonne interface comportementale.

La composition est utilisée dans de nombreux design patterns et vous en apprendrez beaucoup plus sur ses avantages et ses inconvénients tout au long de cet ouvrage.



MUSCLEZ VOS NEURONES

Un appeau est un instrument que les chasseurs emploient pour imiter les appels (cancanements) des canards. Comment implémenteriez-vous un appeau *sans* hériter de la classe Canard ?



Maître et disciple...

Maître : Petit scarabée, dis-moi ce que tu as appris sur la Voie de l'orientation objet.

Disciple : Maître, j'ai appris que la promesse de la Voie de l'orientation objet était la réutilisation.

Maître : Continue, scarabée...

Disciple : Maître, l'héritage permet de réutiliser toutes les bonnes choses, et nous parviendrons à réduire radicalement les temps de développement et à programmer aussi vite que nous coupons le bambou dans la forêt.

Disciple : Scarabée, consacre-t-on plus de temps au code **avant** que le développement ne soit terminé ou **après** ?

Disciple : La réponse est **après**, maître. Nous consacrons toujours plus de temps à la maintenance et à la modification des logiciels qu'à leur développement initial.

Maître : Alors, scarabée, doit-on placer la réutilisation **au-dessus** de la maintenabilité et de l'extensibilité ?

Disciple : Maître, je commence à entrevoir la vérité.

Maître : Je vois que tu as encore beaucoup à apprendre. Je veux que tu ailles méditer un peu plus sur l'héritage. Comme tu l'as constaté, l'héritage a des inconvénients et il y a d'autres moyens de parvenir à la réutilisation.

À propos des design patterns...



Félicitations pour votre
premier pattern !

Vous venez d'appliquer votre premier design pattern, le pattern STRATEGIE. Oui, vous avez utilisé le pattern Stratégie pour revoir la conception de l'application SuperCanard. Grâce à ce pattern, le simulateur est prêt à recevoir toutes les modifications que les huiles pourraient concocter lors de leur prochain séminaire aux Baléares.

Nous n'avons pas pris le plus court chemin pour l'appliquer, mais en voici la définition formelle :

Le pattern Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Ressortez CETTE définition
quand vous voudrez impression-
ner vos amis ou influencer votre
patron.


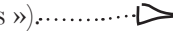



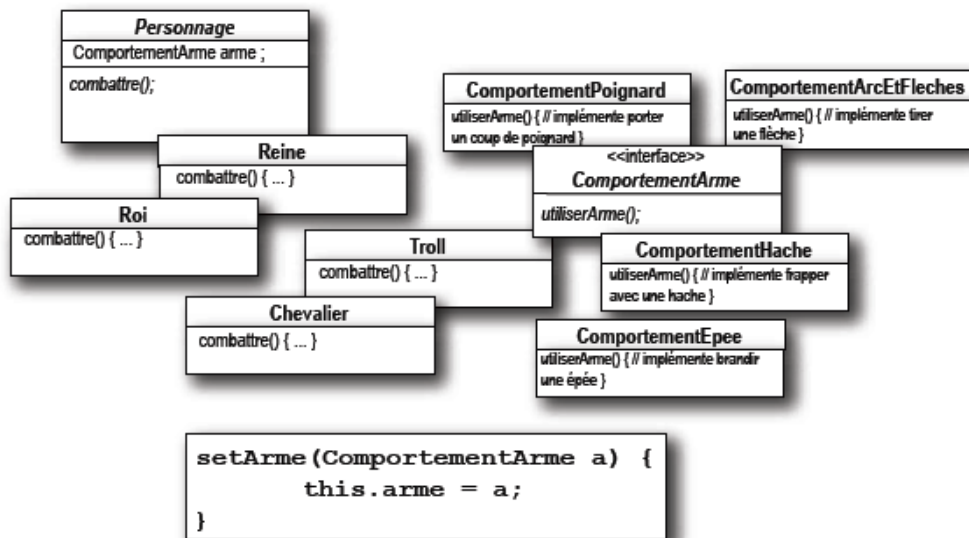
Problème de conception

Vous trouverez ci-dessous un ensemble de classes et d'interfaces pour un jeu d'aventure. Elles sont toutes mélangées. Il y a des classes pour les personnages et des classes pour les comportements correspondant aux armes que les personnages peuvent utiliser. Chaque personnage ne peut faire usage que d'une arme à la fois, mais il peut en changer à tout moment en cours de jeu. Votre tâche consiste à les trier..

(Les réponses se trouvent à la fin du chapitre.)

Votre tâche :

1. Réorganiser les classes.
2. Identifier une classe abstraite, une interface et huit classes ordinaires.
3. Tracer des flèches entre les classes.
 - a. Tracez ce type de flèche pour l'héritage (« extends »). 
 - b. Tracez ce type de flèche pour l'interface (« implements »). 
 - c. Tracez ce type de flèche pour « A-UN ». 
4. Placer la méthode `setArme()` dans la bonne classe.



Entendu à la cafétéria...

Alice

J'aimerais une galette de blé noir avec du jambon, du fromage et un oeuf non brouillé, une autre avec des petits lardons, des oignons et un peu de crème fraîche, un verre de cidre, et pour le dessert deux boules de glace à la vanille avec de la sauce au chocolat et de la crème chantilly et un café décaféiné !

Flo

Donnez moi une complète miroir, une paysanne, une bolée, une dame blanche et un DK !



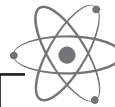
Quelle est la différence entre ces deux commandes ? Aucune ! Elles sont parfaitement identiques, sauf qu'Alice utilise quatre fois plus de mots et risque d'épuiser la patience du ronchon qui prend les commandes.

Que possède donc Flo qu'Alice n'a pas ? **Un vocabulaire commun** avec le serveur. Non seulement il leur est plus facile de communiquer, mais le serveur a moins à mémoriser, puisqu'il a tous les « patterns » de la crêperie en tête.

Les design patterns vous permettent de partager un vocabulaire avec les autres développeurs. Une fois que vous disposez de ce vocabulaire, vous communiquez plus facilement avec eux et vous donnez envie de découvrir les patterns à ceux qui ne les connaissent. Cela vous permet également de mieux appréhender les problèmes d'architecture en **pensant au niveau des patterns** (des structures) et pas au niveau des détails, des *objets*.

Entendu dans le box voisin...

Alors, j'ai créé cette classe Diffusion.
Elle conserve la trace de tous les objets qui l'écoutent, et, chaque fois qu'une nouvelle donnée arrive, elle envoie un message à chaque auditeur. Ce qui est génial, c'est que les auditeurs peuvent se joindre à la diffusion à tout moment, et qu'ils peuvent même se désabonner. C'est une conception vraiment dynamique et faiblement couplée !



MUSCLEZ VOS NEURONES

Connaissez-vous d'autres vocabulaires communs en dehors de la conception OO et des cafétérias ? (Pensez aux garagistes, aux charpentiers, aux grands chefs, au contrôle de trafic aérien). Qu'est-ce que le jargon permet de communiquer ?

Quels sont les aspects de la conception OO qui sont véhiculés par les noms des patterns ? Quelles sont les qualités évoquées par le nom « Pattern Stratégie » ?

Paul,
pourquoi ne
pas dire simplement
que tu appliques le
pattern **Observateur** ?



Exactement.
Si vous
communiquez avec des
patterns, les autres développeurs
savent immédiatement et
précisément de quoi vous parlez.
Mais prenez garde à la « patternite
»... Vous savez que vous l'avez
contractée quand vous commencez
à appliquer des patterns pour
programmer Hello World...

Le pouvoir d'un vocabulaire commun

Lorsque vous communiquez en utilisant des patterns, vous faites plus que partager un JARGON.

Les vocabulaires partagés sont PUISSANTS.

Quand vous communiquez avec un autre développeur ou avec votre équipe en employant un nom de pattern, vous ne communiquez pas seulement un mot mais tout un ensemble de qualités, de caractéristiques et de contraintes que le pattern représente.

Les patterns vous permettent d'exprimer plus de choses en moins de mots. Quand vous utilisez un pattern dans une description, les autres développeurs comprennent tout de suite avec précision la conception que vous avez en tête.

Parler en termes de patterns permet de rester plus longtemps « dans la conception ».

Décrire un système logiciel en termes de patterns vous permet de demeurer au niveau de la conception plus longtemps, sans devoir plonger dans les petits détails de l'implémentation des objets et des classes.

Un vocabulaire commun peut turbopropulser votre équipe de développement. Une équipe versée dans les design patterns avance beaucoup plus rapidement grâce à l'élimination des malentendus.

Un vocabulaire commun incite les développeurs juniors à apprendre plus vite. Les développeurs juniors écoutent les développeurs expérimentés. Quand les développeurs senior utilisent des design patterns, les juniors sont plus motivés pour les apprendre. Construisez une communauté d'utilisateurs de patterns dans votre société.

« Nous appliquons le pattern
Stratégie pour implémenter les différents
comportements de nos canards. » Cet énoncé
vous informe que les comportements de canard
ont été encapsulés dans un groupe de classes
distinct, facile à étendre et à modifier, même,
si nécessaire, au moment de l'exécution.

Combien avez-vous vu de réunions de
conception qui s'enlisaient rapidement
dans des détails d'implémentation ?

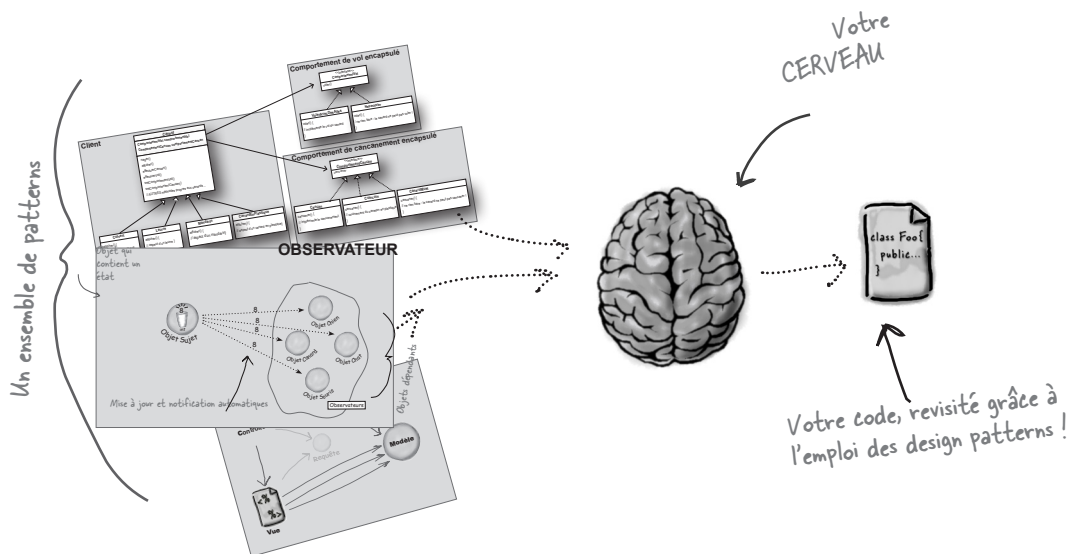
À mesure que votre équipe com-
mencera à partager ses idées et son
expérience en termes de patterns,
vous construirez une communauté de
patterns.

Pensez à lancer un groupe d'étude des
patterns dans votre société. Peut-
être même pourriez-vous être payé à
apprendre...

Comment utiliser les design patterns ?

Nous avons tous utilisé des bibliothèques et des frameworks préexistants. Nous les prenons, écrivons le code en utilisant leur API, le compilons dans nos programmes et tirons parti d'une grande quantité de code que quelqu'un d'autre a écrit. Pensez au API Java et à toutes les fonctionnalités qu'elles vous offrent : réseau, interfaces utilisateurs, E/S, etc. Les bibliothèques et les frameworks font gagner beaucoup de temps dans la construction d'un modèle de développement où nous pouvons nous contenter de choisir les composants que nous insérerons directement. Mais... ils ne nous aident pas à structurer nos propres applications de façon qu'elles soient plus souples, plus faciles à comprendre et à maintenir. C'est là que les design patterns entrent en scène.

Les design patterns ne s'intègrent pas directement dans votre code, ils passent d'abord par votre CERVEAU. Une fois que vous avez chargé les patterns dans votre cerveau et que vous vous débrouillez bien avec, vous pouvez commencer à les appliquer à vos propres conceptions et à retravailler votre ancien code quand vous constatez qu'il commence à se rigidifier et à se transformer en un inextricable fouillis de code spaghetti.



Il n'y a pas de Questions Stupides

Q: Si les design patterns sont tellement géniaux, pourquoi quelqu'un ne les a-t-il pas transformés en bibliothèque pour que je n'aie plus rien à faire ?

R: Les design patterns sont au-dessus des bibliothèques. Ils nous indiquent comment structurer les classes et les objets pour résoudre certains problèmes. C'est à nous de les adapter à nos applications.

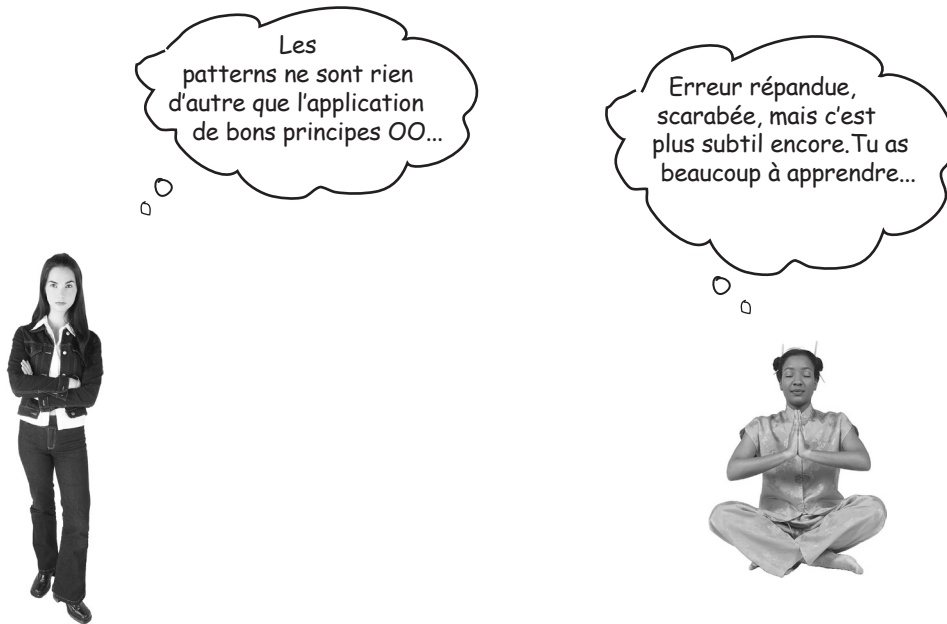
Q: Les bibliothèques et les frameworks ne sont donc pas des design patterns ?

R: Les bibliothèques et les frameworks ne sont pas des design patterns : ils fournissent des implémentations spécifiques que nous lions à notre code. Mais il arrive que les bibliothèques et les frameworks s'appuient sur les design patterns dans leurs implémentations. C'est super, parce qu'une fois que vous aurez

commencé à comprendre les patterns, vous maîtriserez plus vite les API qui sont structurées autour de patterns.

Q: Alors, il n'y a pas de bibliothèques de design patterns ?

R: Non, mais vous allez découvrir qu'il existe des catalogues qui énumèrent les patterns que vous pouvez employer dans vos applications.



Développeur sceptique

Gourou bienveillant

Développeur : O.K, hmm, pourquoi n'est-ce pas seulement une affaire de bonne conception objet ? Je veux dire, tant que j'applique l'encapsulation et que je connais l'abstraction, l'héritage et le polymorphisme, est-ce que j'ai vraiment besoin des design patterns ? Est-ce que ce n'est pas plus simple que cela ? Est-ce que ce n'est pas la raison pour laquelle j'ai suivi tous ces cours sur l'OO ? Je crois que les design patterns sont utiles pour ceux qui connaissent mal la conception OO.

Gourou : Ah, c'est encore un de ces malentendus du développement orienté objet : connaître les bases de l'OO nous rend automatiquement capables de construire des systèmes souples, réutilisables et faciles à maintenir.

Développeur : Non ?

Gourou : Non. En l'occurrence, la construction de systèmes OO possédant ces propriétés n'est pas toujours évidente, et seul beaucoup de travail a permis de la découvrir.

Développeur : Je crois que je commence à saisir. Ces façons de construire des systèmes orientés objets pas toujours évidentes ont été collectées...

Gourou : Oui, et constituent un ensemble de patterns nommés Design Patterns.

Développeur : Et si je connais les patterns, je peux me dispenser de tout ce travail et sauter directement à des conceptions qui fonctionnent tout le temps ?

Souvenez-vous que la connaissance de concepts comme l'abstraction, l'héritage et le polymorphisme ne fait pas de vous un bon concepteur orienté objet. Un gourou de la conception réfléchit à la façon de créer des conceptions souples, faciles à maintenir et qui résistent au changement.



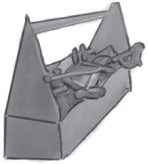
Gourou : Oui, jusqu'à un certain point. Mais n'oublie pas que la conception est un art. Un pattern aura toujours des avantages et des inconvénients. Mais si tu appliques des patterns bien conçus et qui ont fait leurs preuves au fil du temps, tu auras toujours beaucoup d'avance.

Développeur : Et si je ne trouve pas de pattern ?

Gourou : Les patterns sont sous-tendus par des principes orientés objet. Les connaître peut t'aider quand tu ne trouves pas de pattern qui corresponde à ton problème.

Développeur : Des principes ? Tu veux dire en dehors de l'abstraction, de l'encapsulation et...

Gourou : Oui, l'un des secrets de la création de systèmes OO faciles à maintenir consiste à réfléchir à la façon dont ils peuvent évoluer, et ces principes traitent de ces problèmes.



Votre boîte à outils de concepteur

Vous avez presque terminé le premier chapitre ! Vous avez déjà mis quelques outils dans votre boîte à outils OO. Récapitulons-les avant de passer au chapitre 2.

Bases de l'OO

Abstraction
Encapsulation
Polymorphisme
Héritage

← Nous supposons que vous connaissez les concepts OO de base : comment organiser les classes en exploitant le polymorphisme, comment l'héritage est une sorte de conception par contrat et comment l'encapsulation fonctionne. Si vous êtes un peu rouillé sur ces sujets, ressortez votre livre Java Tête la première et revoyez-les, puis relisez rapidement ce chapitre.

Principes OO

Encapsulez ce qui varie.
Préférez la composition à l'héritage.
Programmez des interfaces, non des implémentations.

↪ Nous les reverrons plus en détail et nous en ajouterons d'autres à la liste.

Patterns OO

Stratégie – définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

En voici un, d'autres suivront !

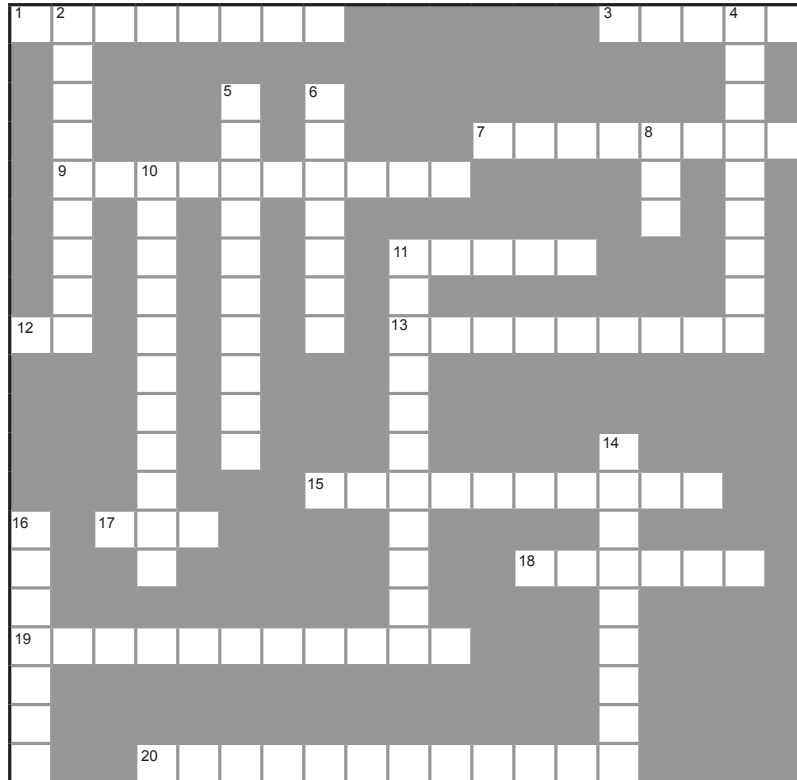
POINTS D'IMPACT

- Connaître les bases de l'OO ne fait pas de vous un bon concepteur.
- Les bonnes conceptions OO sont souples, extensibles et faciles à maintenir.
- Les patterns vous montrent comment construire des systèmes OO de bonne qualité.
- Les patterns résument une expérience éprouvée de la conception objet.
- Les patterns ne contiennent pas de code, mais des solutions génériques aux problèmes de conception. Vous devez les adapter aux applications spécifiques.
- Les patterns ne sont pas *inventés*, ils sont *découverts*.
- La plupart des patterns et des principes traitent des problèmes de changement dans le logiciel.
- La plupart des patterns permettent à une partie d'un système de varier indépendamment de toutes les autres.
- On essaie souvent d'extraire ce qui varie d'un système et de l'encapsuler.
- Les patterns fournissent un langage commun qui peut optimiser votre communication avec les autres développeurs.



Donnons à votre cerveau droit quelque chose à faire.

Ce sont vos mots-croisés standard. Tous les mots de la solution sont dans ce chapitre.



Horizontalement

1. Méthode de canard.
3. Modification abrégée.
7. Les actionnaires y tiennent leur réunion.
9. _____ ce qui varie.
11. Java est un langage orienté _____.
12. Dans la commande de Flo.
13. Pattern utilisé dans le simulateur.
15. Constante du développement.
17. Comportement de canard.
18. Maître.
19. Les patterns permettent d'avoir un _____ commun.
20. Les méthodes set permettent de modifier le _____ d'une classe.

Verticalement

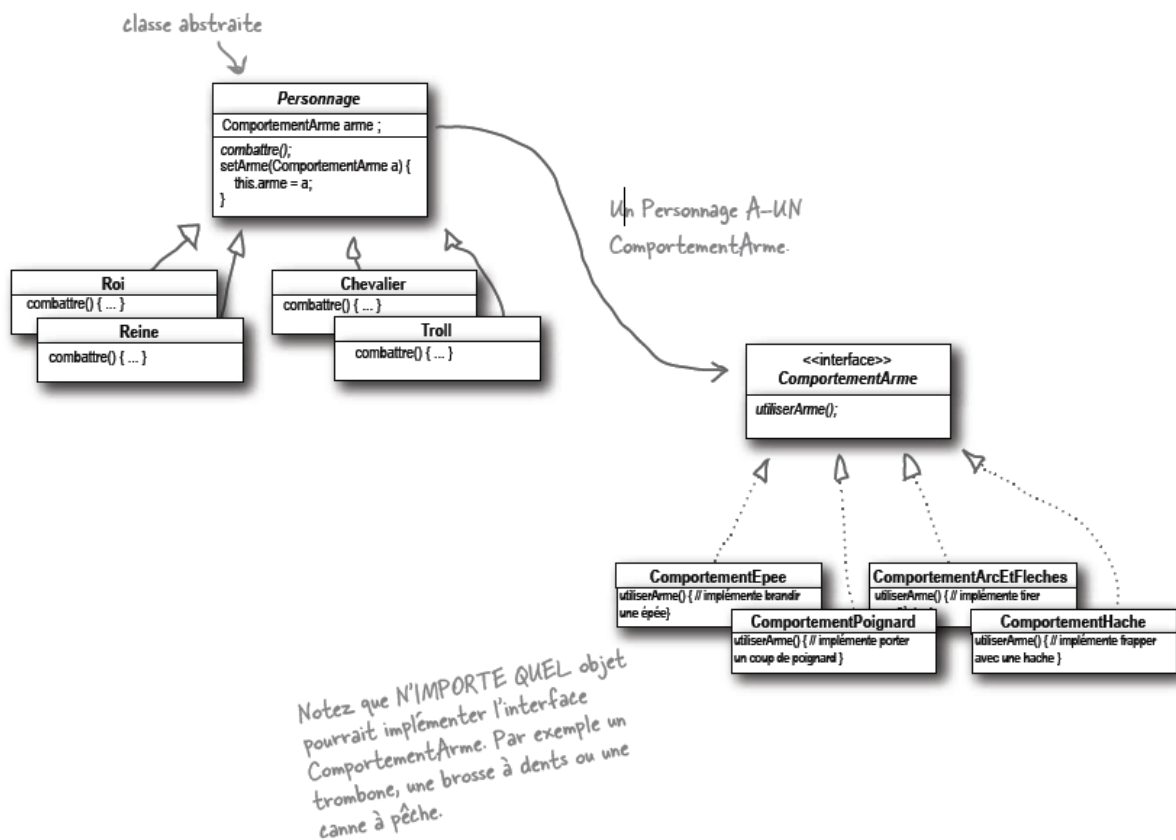
2. Bibliothèque de haut niveau.
4. Programmez une _____, non une implémentation.
5. Les patterns la synthétisent.
6. Ils ne volent ni ne cancanent.
8. Réseau, E/S, IHM.
10. Préférez-la à l'héritage.
11. Paul applique ce pattern.
14. Un pattern est une solution à un problème _____.
16. Sous-classe de Canard.



Solution du problème de conception

Personnage est la superclasse abstraite de tous les autres personnages (Roi, Reine, Chevalier et Troll) tandis que ComportementArme est une interface que toutes les armes implémentent. En conséquence, tous les personnages et toutes les armes sont des classes concrètes.

Pour changer d'arme, chaque personnage appelle la méthode `setArme()` qui est définie dans la superclasse `Personnage`. Lors d'un combat, la méthode `utiliserArme()` est appelée sur l'arme courante d'un personnage donné afin d'infliger de grands dommages corporels à un autre personnage.



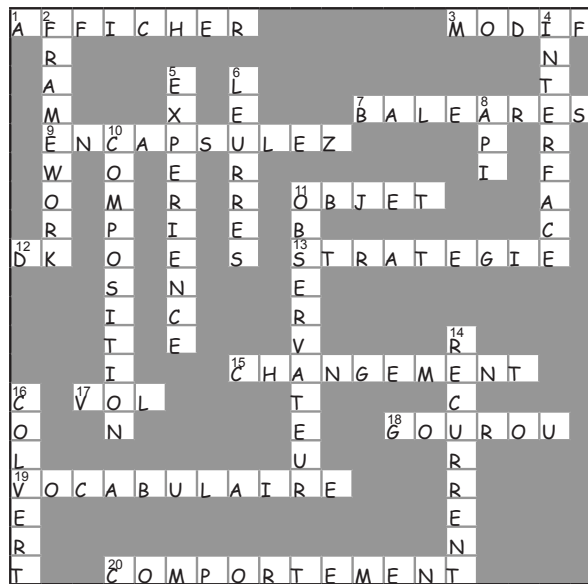
Solutions



À vos crayons

Dans la liste ci-après, quels sont les inconvénients à utiliser l'héritage pour définir le comportement de Canard ? (Plusieurs choix possibles.)

- ☒ A. Le code est dupliqué entre les sous-classes.
- ☒ B. Les changements de comportement au moment de l'exécution sont difficiles.
- ☐ C. Nous ne pouvons pas avoir de canards qui dansent.
- ☒ C. Il est difficile de connaître tous les comportements des canards.
- ☒ D. Les canards ne peuvent pas voler et cancaner en même temps.
- ☐ E. Les modifications peuvent affecter involontairement d'autres canards.



À vos crayons

Quels sont les facteurs qui peuvent induire des changements dans vos applications ? Votre liste est peut-être différente, mais voici quelques-unes de nos réponses. Est-ce que cela vous rappelle quelque chose ?

Mes clients ou mes utilisateurs décident qu'ils veulent autre chose ou qu'ils ont besoin d'une nouvelle fonctionnalité.

Mon entreprise a décidé qu'elle allait changer de système de SGBD et qu'elle allait également acheter ses données chez un autre fournisseur dont le format est différent. Argh !

La technologie évolue et nous devons mettre à jour notre code afin d'utiliser d'autres protocoles.

Après tout ce que nous avons appris en construisant notre système, nous aimerions revenir en arrière et intégrer quelques améliorations.

2 le pattern Observateur

Tenez vos objets au courant



Hé Kevin, j'appelle tout le monde pour dire que la réunion du groupe Patterns a été repoussée à dimanche soir. On va parler du pattern Observateur. C'est le meilleur, Kevin. C'est le MEILLEUR je te dis !

Ne manquez plus jamais rien d'intéressant ! Nous avons un pattern qui met vos objets au courant quand il se passe quelque chose qui pourrait les concerner. Ils peuvent même décider au moment de l'exécution s'ils veulent rester informés. Observateur est l'un des patterns le plus utilisé dans le JDK et il est incroyablement utile. Avant la fin de ce chapitre, nous étudierons également les relations « un-à-plusieurs » et le faible couplage (oui, oui, nous avons dit couplage). Avec Observateur, vous serez l'attraction de la soirée Patterns.

Félicitations !

Votre équipe vient de remporter le marché de la construction de la station météorologique de dernière génération, consultable en ligne, de MétéoExpress, SA.



MétéoExpress, SA
100, allée de la Tornade
98000 Saint-Pascal

Cahier des charges

Félicitations ! Vous avez été sélectionné pour construire notre station météorologique de dernière génération consultable en ligne !

La station sera basée sur notre objet DonneesMeteo (brevet en cours), qui enregistre les conditions météorologiques à un moment donné (température, hygrométrie et pression atmosphérique). Nous aimerions que vous créiez une application qui fournira d'abord trois affichages : conditions actuelles, statistiques et prévisions simples, tous trois mis à jour en temps réel au fur et à mesure que l'objet DonneesMeteo acquiert les données les plus récentes.

De plus, cette station météo doit être extensible. MétéoExpress veut commercialiser une API pour que les autres développeurs puissent réaliser leurs propres affichages et les insérer directement. Nous souhaiterions que vous fournissiez cette API !

MétéoExpress est convaincu d'avoir un excellent modèle métier : une fois les clients accrochés, nous prévoyons de les facturer pour chaque affichage qu'ils utilisent. Et le meilleur pour la fin : vous serez payés en stock options.

Nous attendons avec impatience vos documents de conception et votre version alpha.

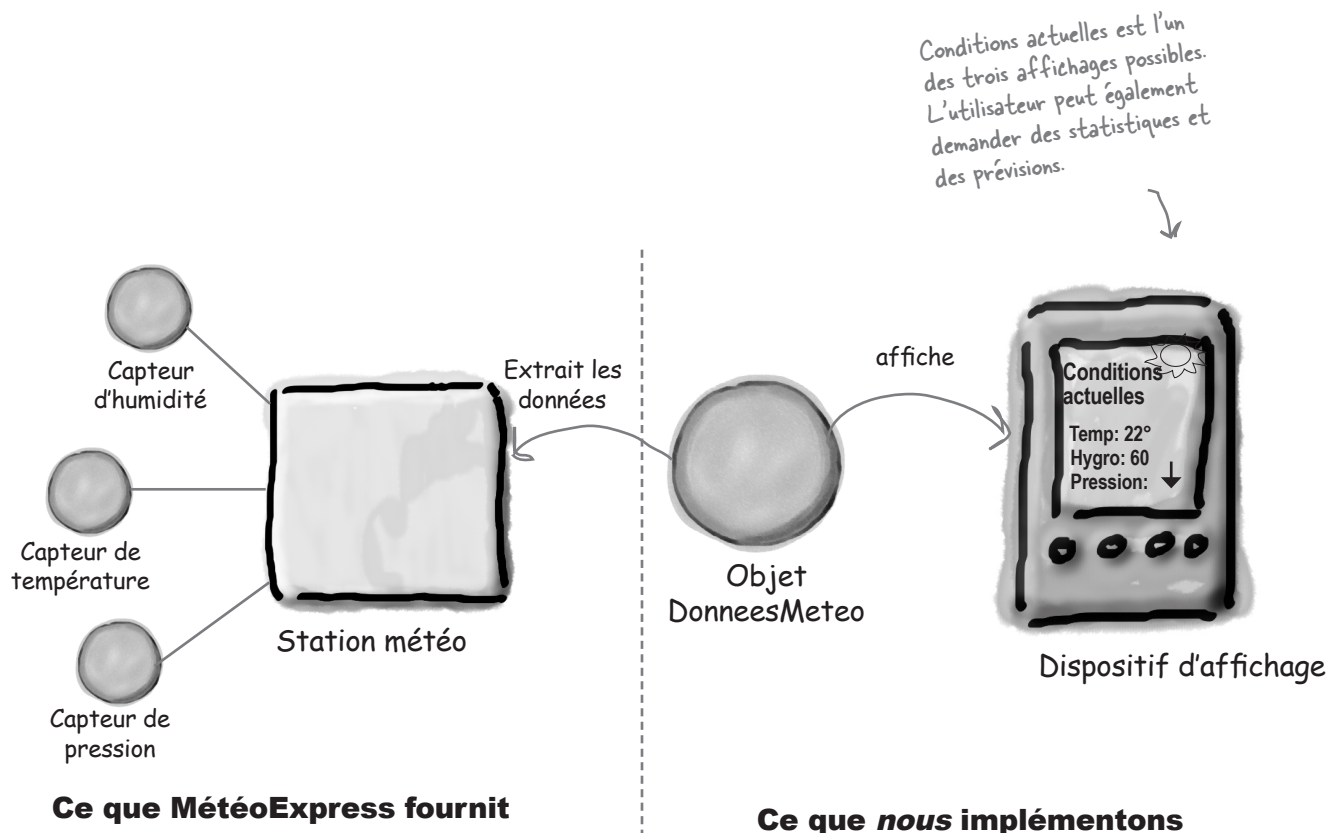
Cordialement,

Jean-Loup Ragan, PDG

P.S. Nous vous envoyons par chrono les fichiers source de DonneesMeteo.

Vue d'ensemble de l'application

Les trois composants du système sont la station météo (l'équipement physique qui recueille les données sur le temps), l'objet `DonneesMeteo` (qui mémorise les données provenant de la station et actualise l'affichage) et l'affichage lui-même que les utilisateurs consultent pour connaître les conditions météorologiques actuelles.

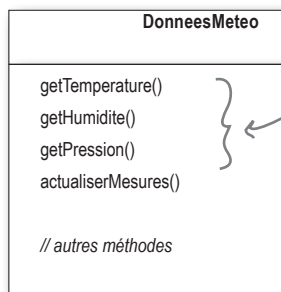


L'objet `DonneesMeteo` sait comment communiquer avec la station physique pour obtenir les données à jour. Il actualise alors l'affichage de trois éléments différents : Conditions actuelles (affiche la température, l'hygrométrie et la pression atmosphérique), Statistiques et Prévisions.

Si nous choisissons de l'accepter, notre tâche consiste à créer une application qui utilise l'objet `DonneesMeteo` pour actualiser ces trois affichages : conditions actuelles, statistiques et prévisions.

À l'intérieur de la classe *DonneesMeteo*

Le lendemain matin, les fichiers source de *DonneesMeteo* arrivent comme promis. Nous jetons un coup d'œil au code qui semble relativement simple :



Ces trois méthodes retournent les mesures les plus récentes : température, humidité et pression atmosphérique.

Nous n'avons pas besoin de savoir COMMENT ces variables sont affectées ; l'objet *DonneesMeteo* sait comment obtenir de la station les informations à jour.

Les développeurs de l'objet *DonneesMeteo* nous ont laissé un indice à propos de ce que nous devons ajouter...

```
/*
 * Cette méthode est appelée
 * chaque fois que les mesures
 * ont été mises à jour
 */
public void actualiserMesures() {
    // Votre code ici
}
```

DonneesMeteo.java

Souvenez-vous que Conditions actuelles n'est que l'UN des trois affichages possibles.



Dispositif d'affichage

Notre tâche consiste à implémenter la méthode `actualiserMesures()` pour qu'elle mette à jour les trois affichages : conditions actuelles, statistiques et prévisions.

Récapitulons...

Les spécifications de MétéoExpress n'étaient pas des plus claires et nous devons plus ou moins deviner ce qu'il faut faire. Alors, que savons-nous jusqu'à maintenant ?



- La classe `DonneesMeteo` a des méthodes d'accès pour trois valeurs de mesures : température, hygrométrie et pression atmosphérique.

`getTemperature()`

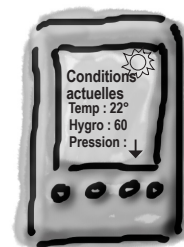
`getHumidite()`

`getPression()`

- La méthode `actualiserMesures()` est appelée chaque fois qu'une nouvelle mesure est disponible. (Nous ne savons pas comment cette méthode est appelée, et peu nous importe ; nous savons simplement qu'elle l'est.

`actualiserMesures()`

- Nous devons implémenter trois affichages qui utilisent les données météorologiques : un affichage des *conditions actuelles*, un affichage des *statistiques* et un affichage des *prévisions*. Ils doivent être mis à jour chaque fois que `DonneesMeteo` acquiert de nouvelles données.



Affichage Un



Affichage Deux



Affichage Trois

- Le système doit être extensible : d'autres développeurs pourront créer des affichages personnalisés et les utilisateurs pourront ajouter ou retirer autant d'éléments qu'ils le souhaitent à l'application. Actuellement, nous ne connaissons que les *trois* types d'affichage initiaux (conditions actuelles, statistiques et prévisions).



Affichages futurs

Premier essai pifométrique de station météo

Voici une première possibilité d'implémentation : nous suivons l'indication des développeurs de MétéoExpress et nous ajoutons notre code à la méthode `actualiserMesures()` :

```
public class DonneesMeteo {  
  
    // déclaration des variables d'instance  
  
    public void actualiserMesures() {  
  
        float temp = getTemperature();  
        float humidite = getHumidite();  
        float pressure = getPression();  
  
        affichageConditions.actualiser(temp, humidite, pression);  
        affichageStats.actualiser(temp, humidite, pression);  
        affichagePrevisions.actualiser(temp, humidite, pression);  
    }  
    // autres méthodes de DonneesMeteo  
}
```

Obtenir les mesures les plus récentes en appelant les méthodes `get` de `DonneesMeteo` (déjà implémentées).

Actualiser les affichages...

Appeler chaque élément pour mettre à jour son affichage en lui transmettant les mesures les plus récentes.



À vos crayons

D'après notre première implémentation, quels énoncés suivants sont vrais ?
(Plusieurs réponses possibles.)

- | | |
|-------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <input type="checkbox"/> A. Nous codons des implémentations concrètes, non des interfaces. | <input type="checkbox"/> D. Les éléments d'affichage n'implémentent pas une interface commune. |
| <input type="checkbox"/> B. Nous devons modifier le code pour chaque nouvel élément d'affichage. | <input type="checkbox"/> E. Nous n'avons pas encapsulé les parties qui varient. |
| <input type="checkbox"/> C. Nous n'avons aucun moyen d'ajouter (ou de supprimer) des éléments d'affichage au moment de l'exécution. | <input type="checkbox"/> F. Nous violons l'encapsulation de la classe <code>DonneesMeteo</code> . |

Qu' est ce qui cloche dans notre implémentation ?

Repensez à tous les concepts et les principes du chapitre 1...

```
public void actualiserMesures() {
    float temp = getTemperature();
    float humidite = getHumidite();
    float pression = getPression();
```

Point de variation : nous devons l'encapsuler.

```
    affichageConditions.actualiser(temp, humidite, pression);
    affichageStats.actualiser(temp, humidite, pression);
    affichagePrevisions.actualiser(temp, humidite, pression);
}
```

En codant des implémentations concrètes, nous n'avons aucun moyen d'ajouter ni de supprimer des éléments sans modifier le programme.

Au moins, nous semblons utiliser une interface commune pour communiquer avec les affichages... Ils ont tous une méthode `actualiser()` qui lit les valeurs de temp, humidite et pression.

Hum, je sais que je suis nouveau ici, mais vu qu'on est dans le chapitre sur le pattern Observateur, on pourrait peut-être commencer à l'utiliser ?



Jetez un coup d'œil au pattern Observateur, puis revenez en arrière et imaginez comment nous pourrions l'exploiter pour notre application météorologique.

Faites connaissance avec le pattern Observateur

Vous savez comment fonctionne un abonnement à un journal ou à un magazine :

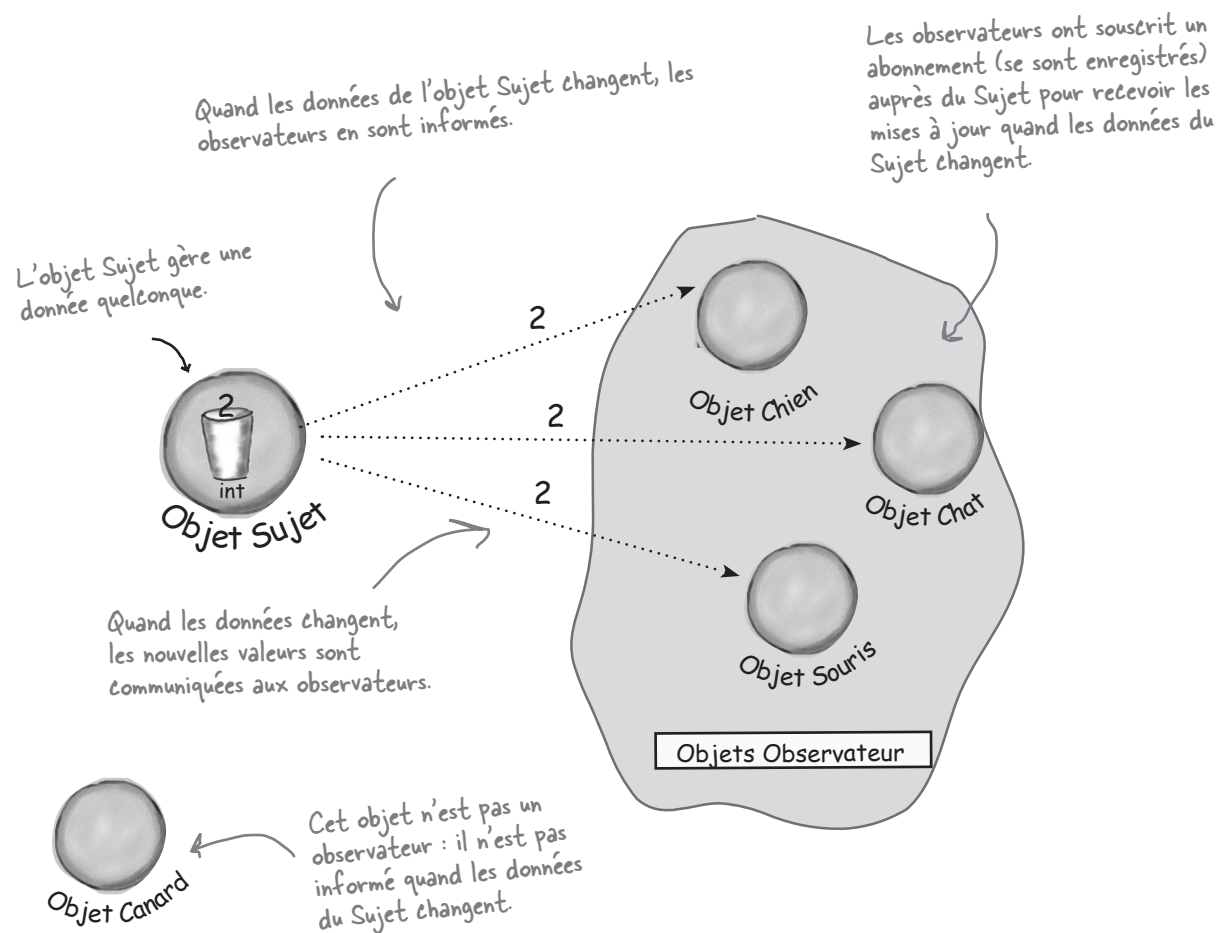
- 1 Un éditeur se lance dans les affaires et commence à diffuser des journaux.
- 2 Vous souscrivez un abonnement. Chaque fois qu'il y a une nouvelle édition, vous la recevez. Tant que vous êtes abonné, vous recevez de nouveaux journaux.
- 3 Quand vous ne voulez plus de journaux, vous résiliez votre abonnement. On cesse alors de vous les livrer.
- 4 Tant que l'éditeur reste en activité, les particuliers, les hôtels, les compagnies aériennes, etc., ne cessent de s'abonner et de se désabonner.



Diffusion + Souscription = pattern Observateur

Si vous comprenez ce qu'est l'abonnement à un journal, vous avez compris l'essentiel du pattern Observateur, sauf que nous appelons l'éditeur le **SUJET et les abonnés les **OBSERVATEURS**.**

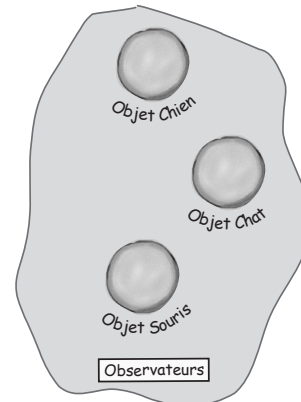
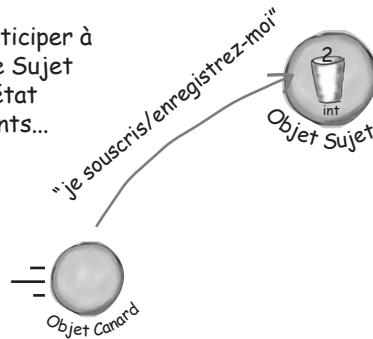
Regardons de plus près :



Une journée dans la vie du pattern Observateur

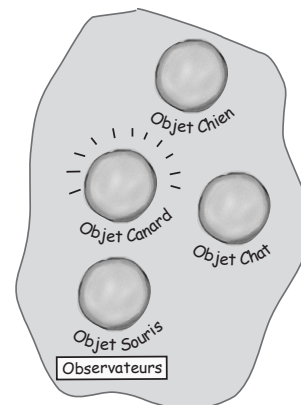
Un objet Canard arrive et dit au Sujet qu'il veut devenir observateur.

Canard veut absolument participer à l'action : tous ces ints que le Sujet envoie chaque fois que son état change ont l'air bien alléchants...



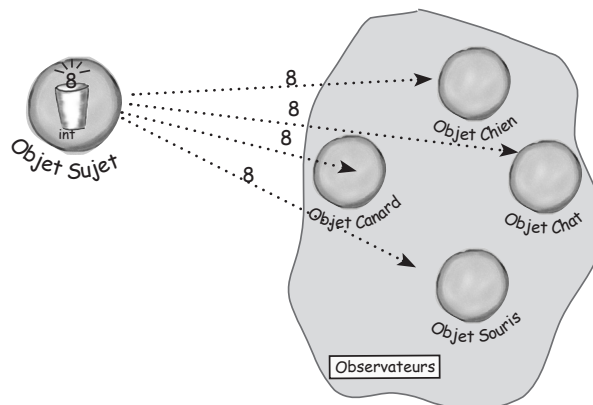
L'objet Canard est maintenant un observateur officiel.

Canard est remonté à bloc... il est sur la liste et il attend avec la plus grande impatience la prochaine notification pour recevoir un int.



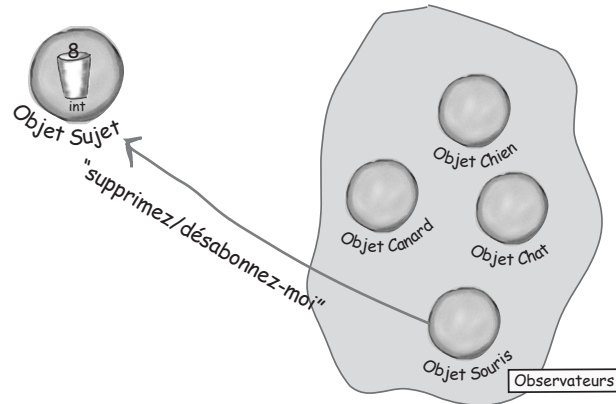
Le Sujet a une nouvelle valeur !

Maintenant, Canard et tous les autres observateurs sont informés que le Sujet a changé.



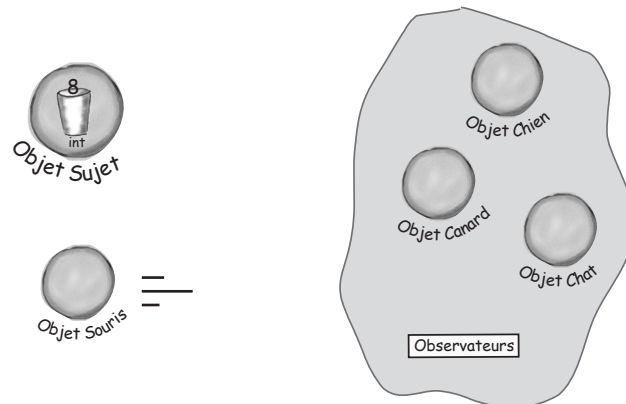
L'objet Souris demande à être supprimé de la liste des observateurs.

L'objet Souris reçoit des ints depuis des lustres et il en a assez. Il décide donc qu'il est temps de cesser d'être observateur.



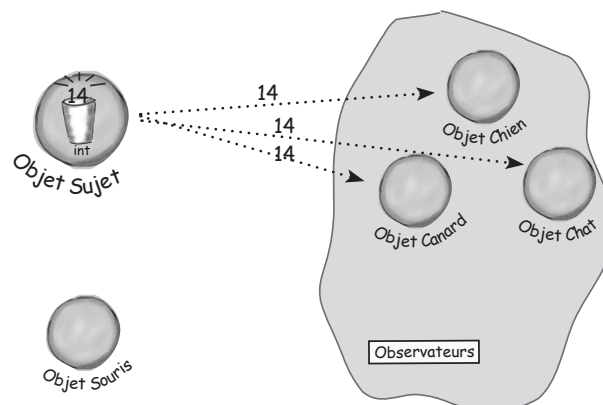
Sortie de Souris !

Le Sujet accuse réception de la requête de Souris et la supprime de la liste des observateurs.



Le Sujet a un nouvel int.

Tous les observateurs en reçoivent notification, sauf Souris qui n'est plus abonné. Ne le dites à personne, mais tous ces ints lui manquent secrètement... Peut-être redemandera-t-il un jour à être un observateur.





Comédie express : un sujet d'observation

Dans le sketch d'aujourd'hui, deux développeurs d'après la bulle technologique rencontrent en personne un chasseur de têtes...

1

Bonjour, c'est Léo. Je cherche un poste de développeur Java. J'ai cinq ans d'expérience et...

Développeur n°1

2

Oui, oui, comme tout le monde, bébé. Je te mets sur ma liste de développeurs Java. Ne m'appelle pas, c'est moi qui t'appellerai !

Chasseur de têtes/Sujet

3

Bonjour, Maud à l'appareil. J'ai écrit un tas de systèmes EJB. Je suis intéressée par tout poste en développement Java.

Développeur n°2

4

Je t'ajoute à la liste, tu seras au courant en même temps que tout le monde.

Sujet

- 5** Pendant ce temps, la vie continue pour Léo et Maud. Si un poste en Java se présente, ils en seront informés. Après tout, ils sont observateurs.



Maud trouve son propre job !



Deux semaines plus tard...



Maud aime la vie. Elle ne fait plus partie des observateurs.

Elle apprécie également la prime confortable qu'elle a touchée pour avoir évité à l'entreprise de payer un chasseur de têtes.

Mais qu'est devenu notre cher Léo ? Nous avons entendu dire qu'il avait battu le chasseur de têtes à son propre jeu. Non seulement il est toujours observateur, mais il a également sa propre liste, et il informe ses propres observateurs. Léo est à la fois sujet et observateur.



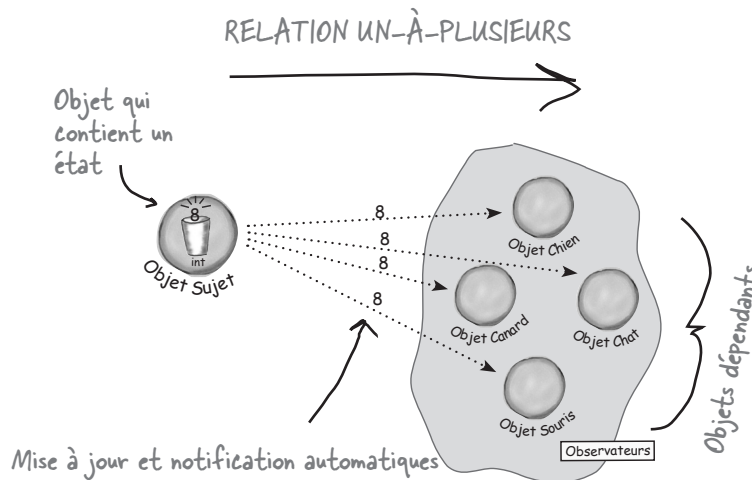
Le pattern Observateur : définition

Quand vous essayez de représenter le pattern Observateur, un service d'abonnement à un journal est une métaphore qui permet de bien visualiser le processus.

Mais, dans le monde réel, le pattern Observateur est généralement défini comme ceci :

Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Relions cette définition à la façon dont nous avons décrit le pattern :



Sujet et observateurs définissent la relation un-à-plusieurs.

Les observateurs sont dépendants du sujet : quand l'état du sujet change, les observateurs en sont informés. Selon le type de notification, l'observateur peut également être mis à jour avec les nouvelles valeurs.

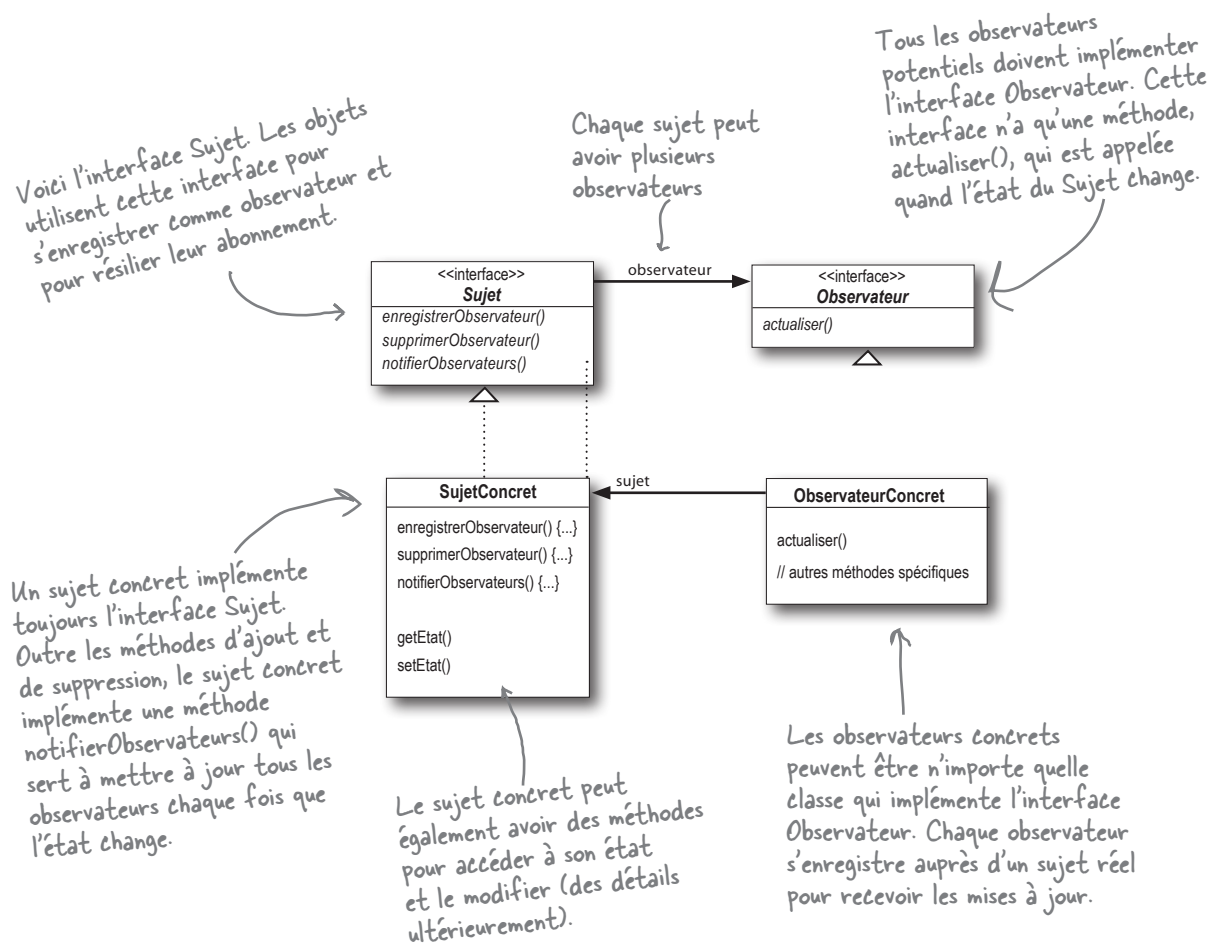
Comme vous le verrez, il y a plusieurs façons d'implémenter le pattern Observateur mais la plupart tournent autour d'une conception qui comprend des interfaces Sujet et Observateur.

Voyons un peu...

Le pattern Observateur définit une relation un-à-plusieurs entre des objets.

Quand l'état de l'un des objets change, tous les objets dépendants en sont informés.

Le pattern Observateur : le diagramme de classes



il n'y a pas de questions stupides

Q: Quel est le rapport avec la relation un-à-plusieurs ?

R: Dans le pattern Observateur, le Sujet est l'objet qui contient l'état et qui le contrôle. Il y a donc UN sujet avec un état. En revanche, les observateurs utilisent l'état, même s'ils ne le possèdent pas. Il y a PLUSIEURS observateurs qui comptent sur le sujet pour les informer de ses changements d'état. Il y a donc une relation entre UN sujet et PLUSIEURS observateurs.

Q: Et que vient faire la dépendance là-dedans ?

R: Comme le sujet est le seul propriétaire de cette donnée, les observateurs en dépendent pour qu'ils soient actualisés quand elle change. C'est une conception objet plus saine que si l'on permettait à plusieurs objets de contrôler la même donnée.

Le pouvoir du Couplage faible

Lorsque deux objets sont faiblement couplés, ils peuvent interagir sans pratiquement se connaître.

Le pattern Observateur permet une conception dans laquelle le couplage entre sujets et observateurs est faible.

Pourquoi ?

Le sujet ne sait qu'une chose à propos de l'observateur : il implémente une certaine interface (l'interface Observateur). Il n'a pas besoin de connaître, ni la classe concrète de l'observateur, ni ce qu'il fait ni quoi que ce soit d'autre.

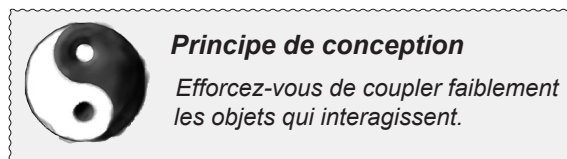
Nous pouvons ajouter des observateurs à tout moment. Comme le sujet dépend uniquement d'une liste d'objets qui implémentent l'interface Observateur, nous pouvons ajouter des observateurs à volonté. En fait, nous pouvons remplacer n'importe quel observateur par un autre au moment de l'exécution : le sujet continuera à ronronner comme si de rien n'était. De même, nous pouvons supprimer des observateurs n'importe quand.

Nous n'avons jamais besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs. Disons que se présente une nouvelle classe concrète qui a besoin d'être un observateur. Nous n'avons pas besoin d'ajouter quoi que ce soit au sujet pour gérer ce nouveau type. Il suffit d'implémenter l'interface Observateur dans la nouvelle classe et de l'enregistrer en tant qu'observateur. Le sujet ne s'en soucie aucunement : il continuera à diffuser des notifications à tous les objets qui implémentent l'interface Observateur.

Nous pouvons réutiliser les objets et les sujets indépendamment les uns des autres. Si nous avons un autre emploi d'un sujet ou d'un observateur, nous pouvons les réutiliser sans problème parce qu'ils sont faiblement couplés.

Les modifications des sujets n'affectent pas les observateurs et inversement. Comme ils sont faiblement couplés, nous sommes libres de les modifier à notre guise tant que les objets continuent à remplir leurs obligations : implémenter les interfaces.

Combien de modifications différentes pouvez-vous identifier ?



Les conceptions faiblement couplées nous permettent de construire des systèmes OO souples, capables de faire face aux changements parce qu'ils minimisent l'interdépendance entre les objets.



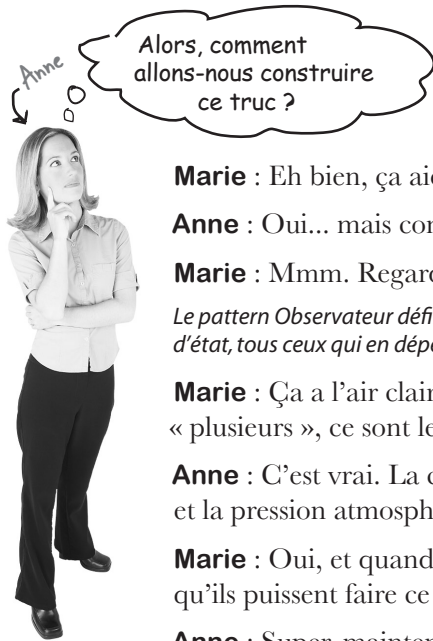
À vos crayons

Avant de continuer, essayez d'esquisser les classes nécessaires pour implémenter la Station Météo, notamment la classe `DonneesMeteo` et ses différents affichages. Vérifiez que votre diagramme montre la façon dont tous les composants s'assemblent et dont un autre développeur pourrait implémenter son propre affichage.

S'il vous faut un peu d'aide, lisez la page suivante : vos collègues sont déjà en train de parler de la conception de la Station Météo.

Conversation dans un box

Revenons au projet de Station Météo. Vos collègues ont déjà commencé à réfléchir au problème...



Marie : Eh bien, ça aide de savoir que nous utilisons le pattern Observateur.

Anne : Oui... mais comment va-t-on l'appliquer ?

Marie : Mmm. Regardons de nouveau la définition :

Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Marie : Ça a l'air clair quand on y pense. Notre classe `DonneesMeteo` est le « un », et le « plusieurs », ce sont les éléments qui affichent les différentes mesures.

Anne : C'est vrai. La classe `DonneesMeteo` a évidemment un état... la température, l'humidité et la pression atmosphérique, et bien sûr ces données changent.

Marie : Oui, et quand ces données changent, il faut notifier tous les éléments d'affichage pour qu'ils puissent faire ce qu'ils doivent faire avec les mesures.

Anne : Super, maintenant je crois que je vois comment on peut appliquer le pattern Observateur à notre problème.

Marie : Mais il y a encore deux ou trois choses que je ne suis pas sûre d'avoir comprises.

Anne : Par exemple ?

Marie : D'abord, comment faire pour que les éléments d'affichage obtiennent les mesures ?

Anne : Eh bien, en regardant de nouveau le diagramme du pattern Observateur, si nous faisons de l'objet `DonneesMeteo` le sujet et que les éléments d'affichage sont les observateurs, alors les affichages vont s'enregistrer eux-mêmes auprès de l'objet `DonneesMeteo` pour obtenir les informations dont ils ont besoin, non ?

Marie : Oui... Et une fois que la Station Météo est au courant de l'existence d'un élément d'affichage, elle peut simplement appeler une méthode pour lui transmettre les mesures.

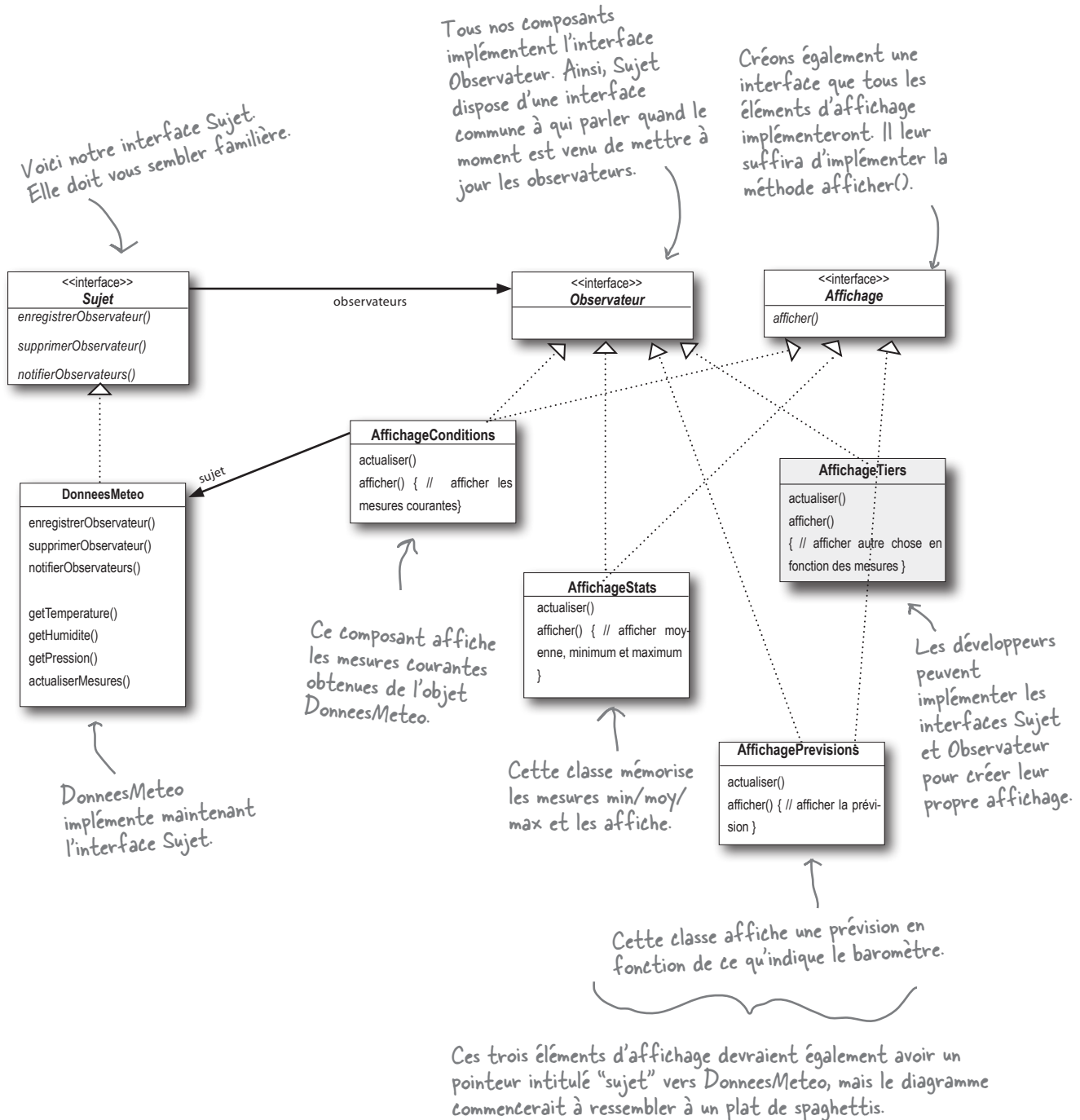
Anne : Il faut se souvenir que chaque affichage peut être différent... et c'est là que je crois qu'intervient une interface commune. Même si chaque composant est d'un type différent, ils doivent tous implémenter la même interface pour que l'objet `DonneesMeteo` sache comment leur transmettre les mesures.

Marie : Je vois ce que tu veux dire. Chaque affichage aura par exemple une méthode `actualiser()` que `DonneesMeteo` va appeler.

Anne : Et `actualiser()` est définie dans une interface commune que tous les éléments implémentent...

Concevoir la Station Météo

Comparez ce diagramme avec le vôtre.



Implémenter la Station Météo

Nous allons commencer notre implémentation en utilisant le diagramme de classes et en suivant les indications de Marie et d'Anne (page 55). Vous verrez bientôt dans ce chapitre que Java dispose de ses propres classes pour prendre en charge le pattern Observateur, mais nous allons quand même mettre les mains dans le cambouis et créer les nôtres pour l'instant. Si vous pouvez souvent utiliser les classes Java standard, construire les vôtres vous apporte plus de souplesse dans de nombreux cas (et ce n'est pas si difficile). Commençons donc par les interfaces :

```
public interface Sujet {
    public void enregistrerObservateur(Observateur o);
    public void supprimerObservateur(Observateur o);
    public void notifierObservateurs();
}
```

Ces deux méthodes acceptent un Observateur en argument ; autrement dit, l'Observateur à enregistrer ou à supprimer.

Cette méthode est appelée pour notifier tous les observateurs que l'état de Sujet a changé.

```
public interface Observateur {
    public void actualiser(float temp, float humidite, float pression);
}
```

Les valeurs de l'état que les Observateurs obtiennent du Sujet quand une mesure change

```
public interface Affichage {
    public void afficher();
}
```

L'interface Affichage ne contient qu'une méthode, afficher(), que nous appellerons quand un élément devra être affiché.

L'interface Observateur étant implémentée par tous les observateurs, ils doivent tous implémenter la méthode actualiser(). Ici, nous appliquons l'idée de Marie et Anne et nous transmettons les mesures aux observateurs.



MUSCLEZ VOS NEURONES

Marie et Anne ont pensé que transmettre directement les mesures aux observateurs était la méthode la plus simple pour mettre à jour l'état. Pensez-vous que cela soit judicieux ? Indication : est-ce un point de l'application susceptible de changer dans le futur ? Si c'est le cas, le changement sera-t-il bien encapsulé ou entraînera-t-il des modifications dans de nombreuses parties du code ?

Voyez-vous d'autres façons de résoudre le problème de la transmission de l'état mis à jour aux observateurs ?

Ne vous inquiétez pas : nous reviendrons sur cette décision de conception lorsque nous aurons terminé l'implémentation initiale.

Implémenter l'interface Sujet dans DonneesMeteo

Vous souvenez-vous de notre premier essai d'implémentation de la classe DonneesMeteo au début de ce chapitre ? Peut-être voulez-vous vous rafraîchir la mémoire ? Maintenant, il est temps d'appliquer le pattern Observateur...

SOUVENEZ-VOUS : les listings ne contiennent pas d'instructions import et package. Téléchargez le code source complet sur le site web du livre. Vous trouverez l'URL page xxxiii de l'Intro.

DonneesMeteo implémente maintenant l'interface Sujet.

```

public class DonneesMeteo implements Subject {
    private ArrayList observateurs;
    private float temperature;
    private float humidite;
    private float pression;

    public DonneesMeteo() {
        observateurs = new ArrayList();
    }

    public void enregistrerObservateur(Observateur o) {
        observateurs.add(o);
    }

    public void supprimerObservateur(Observateur o) {
        int i = observateurs.indexOf(o);
        if (i >= 0) {
            observateurs.remove(i);
        }
    }

    public void notifierObservateurs() {
        for (int i = 0; i < observateurs.size(); i++) {
            Observer observer = (Observer)observateurs.get(i);
            observer.update(temperature, humidite, pression);
        }
    }

    public void actualiserMesures() {
        notifierObservateurs();
    }

    public void setMesures(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        this.pression = pression;
        actualiserMesures();
    }

    // autres méthodes de DonneesMeteo
}

```

Nous avons ajouté une ArrayList pour contenir les observateurs et nous la créons dans le constructeur.

Quand un observateur s'enregistre, nous l'ajoutons simplement à la fin de la liste.

De même, quand un observateur veut se « désenregistrer » nous le supprimons de la liste.

Voici le plus intéressant. C'est là que nous informons les observateurs de l'état du sujet. Comme ce sont tous des objets Observateur, nous savons qu'ils implémentent tous actualiser() et nous savons donc comment les en notifier.

Nous notifions les observateurs quand nous recevons de la Station Météo des mesures mises à jour.

Bon, nous voulions accompagner chaque exemplaire de ce livre d'une jolie petite station météo mais l'éditeur a refusé. Au lieu de lire les données sur une vraie station, nous allons donc utiliser cette méthode pour tester nos affichages. Ou bien, pour le plaisir, vous pouvez écrire un programme qui récupère des relevés sur le web.

Ici, nous implémentons l'interface Sujet.

Maintenant, construisons les affichages

Maintenant que nous avons amélioré la classe `DonneesMeteo`, il est temps de construire les affichages. MétéoExpress en a commandé trois : un pour les conditions actuelles, un pour les statistiques et un pour les prévisions. Jetons un coup d'œil au premier ; une fois que vous aurez une bonne idée de son fonctionnement, regardez le code des autres dans les fichiers que vous avez téléchargés. Vous verrez qu'il est très similaire.

Cet affichage implémente Observateur pour pouvoir obtenir les changements de l'objet DonneesMeteo.

Il implémente également Affichage car notre API va avoir besoin de tous les éléments pour implémenter cette interface.

```
public class AffichageConditions implements Observateur, Affichage {
    private float temperature;
    private float humidite;
    private Sujet donneesMeteo;

    public AffichageConditions(Sujet donneesMeteo) {
        this.donneesMeteo = donneesMeteo;
        donneesMeteo.enregistrerObservateur(this);
    }

    public void actualiser(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        afficher();
    }

    public void afficher() {
        System.out.println("Conditions actuelles : " + temperature
            + " degrés C et " + humidite + " % d'humidité");
    }
}
```

Nous transmettons au constructeur l'objet DonneesMeteo (le Sujet) et nous l'utilisons pour enregistrer l'affichage en tant qu'observateur.

Quand actualiser() est appelée, nous sauvegardons température et humidité et nous appelons afficher().

La méthode afficher() affiche simplement les mesures de température et d'humidité les plus récentes.

Il n'y a pas de Questions Stupides

Q: Est-ce qu'`actualiser()` est le meilleur endroit pour appeler `afficher()` ?

R: Dans cet exemple simple, cela a un sens d'appeler `afficher()` quand les valeurs changent. Mais vous avez raison : il y a de bien meilleures

façons de concevoir l'affichage des données. Nous les verrons quand nous aborderons le pattern Modèle-Vue-Contrôleur.

Q: Pourquoi avez-vous mémorisé une référence au Sujet ? On dirait que vous ne l'utilisez plus après le constructeur ?

R: Exact, mais si nous voulons plus tard nous supprimer nous-même de la liste des observateurs, ce sera pratique d'avoir déjà une référence au Sujet.

Mettre en route la Station Météo



1 Écrivons d'abord un programme pour tester.

La Station Météo est prête à fonctionner : il ne manque plus qu'un peu de code pour coller tous les morceaux. Voici notre première tentative. Nous y reviendrons plus loin dans l'ouvrage et ferons en sorte que tous les composants soient facilement « enfichables » *via* un fichier de configuration. Pour l'instant, voyons le fonctionnement global :

```
public class StationMeteo {
    public static void main(String[] args) {
        DonneesMeteo donneesMeteo = new DonneesMeteo();

        AffichageConditions affichageCond = new AffichageConditions(donneesMeteo);

        AffichageStats affichageStat = new AffichageStats(donneesMeteo);
        AffichagePrevisions affichagePrev = new AffichagePrevisions(donneesMeteo);

        donneesMeteo.setMesures(26, 65, 1020);
        donneesMeteo.setMesures(28, 70, 1012);
        donneesMeteo.setMesures(22, 90, 1012);
    }
}
```

Si vous ne voulez pas télécharger le code, vous pouvez mettre ces deux lignes en commentaire et l'exécuter.

Créons d'abord l'objet DonneesMeteo.

Créer les trois affichages et leur transmettre l'objet DonneesMeteo.

Simuler les nouvelles mesures.

2 Exécutez le code et admirez la magie du pattern Observateur

```
Fichier Edition Fenêtre Aide Tempête
%java StationMeteo
Conditions actuelles : 26 degrés C et 65.0 % d'humidité
Température Moy/Max/Min = 26.0/26.0/26.0
Prévision : Amélioration en cours !
Conditions actuelles : 28 degrés C et 70.0 % d'humidité
Température Moy/Max/Min = 27.0/28.0/26.0
Prévision : Le temps se rafraîchit
Conditions actuelles : 22 degrés C et 90.0 % d'humidité
Température Moy/Max/Min = 25.0/28.0/22.0
Prévision : Dépression bien installée
%
```




À vos crayons

Le PDG de MétéoExpress, Jean-Loup Ragan, vient de vous appeler : ils ne peuvent pas commercialiser leur station sans un affichage de l'humidex. Voici les détails :

L'humidex est un indice qui combine la température et l'humidité afin de déterminer la température apparente (la chaleur réellement ressentie). Pour le calculer, on prend la température, T, et l'humidité relative (HR) et on applique cette formule :

$$\begin{aligned} \text{humidex} = & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * HR - 1.00254 * 10^{-1} * T \\ & * HR + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * HR^2 + 3.45372 * 10^{-4} \\ & * T^2 * HR - 8.14971 * 10^{-4} * T * HR^2 + 1.02102 * 10^{-5} * T^2 * HR^2 - \\ & 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * HR^3 + 1.42721 * 10^{-6} * T^3 * HR \\ & + 1.97483 * 10^{-7} * T * HR^3 - 2.18429 * 10^{-8} * T^3 * HR^2 + 8.43296 * \\ & 10^{-10} * T^2 * HR^3 - 4.81975 * 10^{-11} * T^3 * HR^3 \end{aligned}$$

Allez-y, tapez-la !

Vous pouvez télécharger humidex.txt sur le site de ce livre

Non, c'est une blague. Ne vous inquiétez pas, vous n'avez pas besoin de taper cette formule ; il suffit de créer votre propre fichier AffichageHumidex.java et d'y copier celle que vous trouverez dans humidex.txt. Attention, cette formule ne fonctionne qu'avec des degrés Fahrenheit. Vous devrez insérer avant une ligne pour la conversion en degrés Celsius : $t = (t - 32) * 5 / 9$;

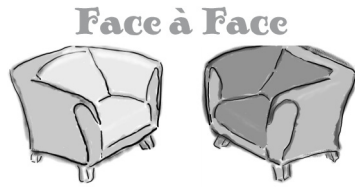
Vous pouvez télécharger humidex.txt sur le site de ce livre

Comment ça marche ? Ouvrez Météorologie la tête la première ou essayez de demander à quelqu'un chez Météo France (ou encore essayez une recherche sur Google). Quand vous aurez terminé, le résultat devrait ressembler à celui-ci :

Voilà ce qui a changé dans l'affichage

Fichier Édition Fenêtre Aide ArcEnCiel

```
%java StationMeteo
Conditions actuelles : 26 degrés C et 65.0 % d'humidité
Température Moy/Max/Min = 26.0/26.0/26.0
Prévision : Amélioration en cours !
L'humidex est de 28,3
Conditions actuelles : 28 degrés C et 70.0 % d'humidité
Température Moy/Max/Min = 27.0/28.0/26.0
Prévision : Le temps se rafraîchit
L'humidex est de 30,5
Conditions actuelles : 22 degrés C et 90.0 % d'humidité
Température Moy/Max/Min = 25.0/28.0/22.0
Prévision : Dépression bien installée
L'humidex est de 28,7
%
```

Au programme ce soir : **Un Sujet et un Observateur s'affrontent sur la bonne façon de transmettre les informations sur l'état à l'Observateur.**

Sujet

Je suis heureux que nous ayons enfin l'occasion de bavarder en personne.

Eh bien, je fais mon travail, non ? Je vous tiens en permanence au courant... Ce n'est pas parce que je ne sais pas vraiment qui vous êtes que je ne me soucie pas de vous. Et d'ailleurs, je sais ce qui est le plus important — vous implémentez l'interface Observateur.

Ah oui ? Par exemple ?

Oh ! Excusez-moi. Il faut que je transmette mon état avec mes notifications pour que vous autres feignants d'observateurs sachez ce qui s'est passé !

Eh bien... Je suppose que cela pourrait marcher. Mais il faudrait que je m'ouvre encore un peu plus pour que vous puissiez obtenir cet état. Ce serait quelque peu dangereux. Je ne peux pas vous laisser entrer et fouiner partout.

Observateur

Vraiment ? Je ne savais pas qu'on avait tant d'importance pour vous, nous autres observateurs.

Bon, bon, mais ce n'est qu'une petite partie de ce que je suis. De toute façon, j'en sais beaucoup à votre sujet...

Eh bien comme vous nous transmettez toujours vos changements d'état, nous savons toujours ce qui se passe en vous. C'est parfois un peu gênant...

Attendez une minute ! D'abord nous ne sommes pas feignants. C'est juste que nous avons autre chose à faire entre vos précieuses notifications, Monsieur le Sujet. Et deuxièmement, pourquoi vous ne nous laissez pas nous adresser à vous pour connaître votre état au lieu de l'envoyer à tout le monde ?

Sujet

Oui, je pourrais vous laisser **tirer** mon état. Mais est-ce que ce ne serait pas moins pratique pour vous ? Si vous devez me contacter chaque fois que vous voulez quelque chose, il vous faudra une quantité d'appels de méthodes pour obtenir tout l'état dont vous avez besoin. C'est pourquoi je préfère **pousser**... En procédant ainsi, vous avez tout ce qu'il vous faut dans une seule notification.

Oui, je vois l'intérêt des deux techniques. J'ai remarqué qu'il existe un pattern Observateur intégré à Java et qu'il permet de tirer ou de pousser.

Parfait... Je vais peut être voir un bon exemple qui ira dans votre sens et je changerai d'avis.

Observateur

Pourquoi ne pas simplement écrire des méthodes get publiques qui nous permettraient de retirer l'état dont nous avons besoin ?

Ne vous poussez pas du col comme ça ! Il y a tellement de types d'observateurs différents que vous n'avez aucun moyen d'anticiper ce dont nous avons besoin. Laissez-nous venir vers vous pour obtenir votre état. Comme ça, si certains d'entre nous n'ont besoin que d'une petite partie de l'état, nous ne sommes pas forcés de l'avoir en entier. Cela faciliterait aussi les modifications. Imaginons que vous évoluez et que vous ajoutez quelque chose à l'état. Si nous tirons, il devient inutile de modifier les appels des méthodes de mise à jour sur chaque observateur. Il suffit de vous modifier pour permettre à d'autres méthodes get d'accéder à l'état supplémentaire.

Vraiment ? Je suppose que nous allons le voir bientôt....

Nous, nous mettre d'accord sur quelque chose ? On peut toujours espérer.

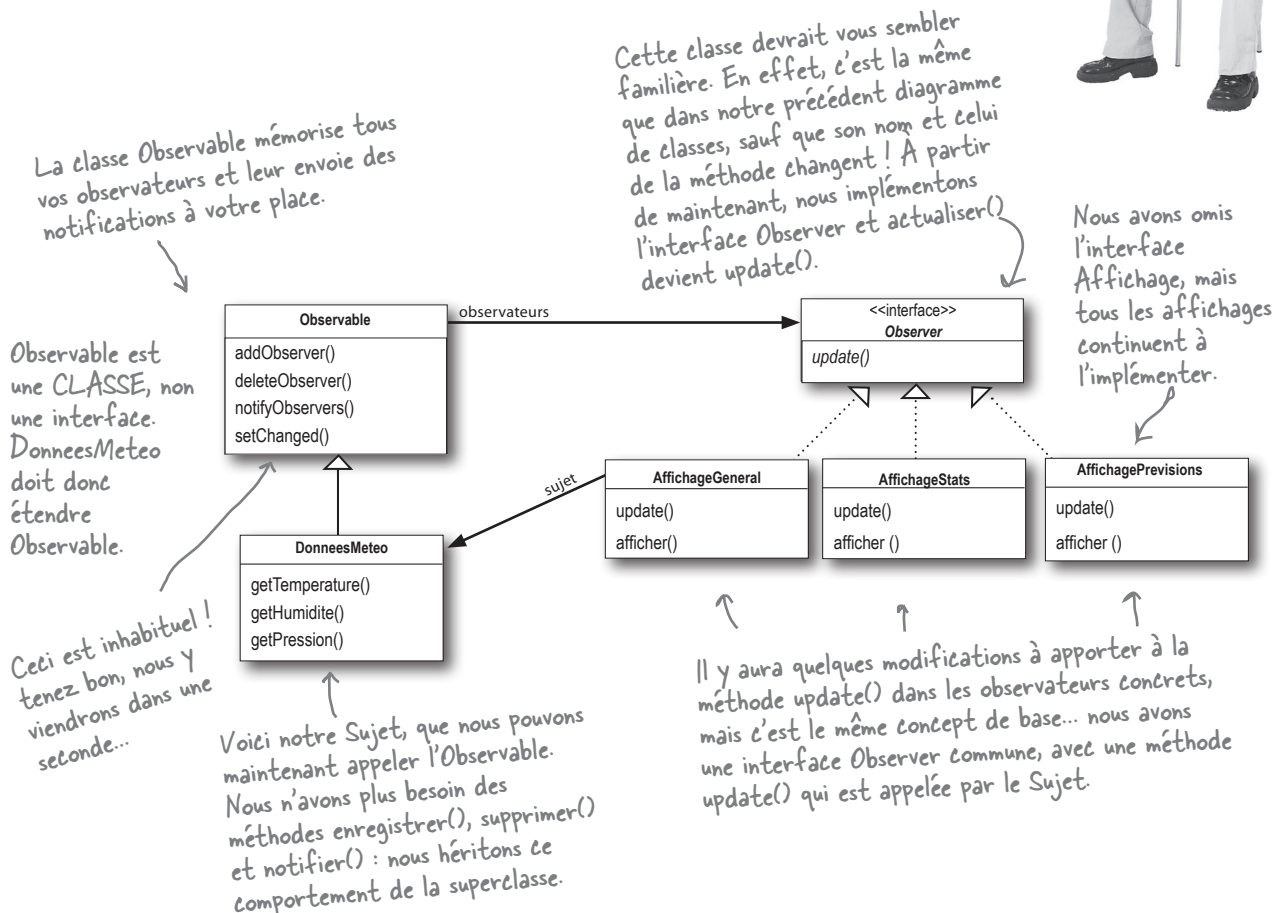
Utiliser le pattern Observateur de Java

Jusqu'ici, nous avons écrit notre propre code pour appliquer le pattern Observateur, mais Java le prend en charge nativement dans plusieurs de ses API.

Le cas le plus général est constitué de l'interface Observer et de la classe Observable du package java.util. Elles sont très similaires à nos interfaces Sujet et Observateur, mais elles vous apportent un certain nombre de fonctionnalités « toutes prêtes ». Comme vous allez le voir, vous pouvez choisir de pousser ou de tirer pour mettre à jour vos observateurs.

Pour avoir une vue d'ensemble de java.util.Observer et de java.util.Observable, étudions cette conception retravaillée de la StationMeteo :

Avec le support intégré de Java, il suffit d'étendre Observable et de lui indiquer quand notifier les observateurs. L'API fait le reste à votre place.



Comment fonctionne le pattern Observateur de Java

Le pattern Observateur intégré à Java fonctionne un peu différemment de l'implémentation que nous avons choisie pour notre Station Météo. La différence la plus évidente réside dans le fait que `DonneesMeteo` (notre sujet) étend maintenant la classe `Observable` et hérite (entre autres) des méthodes `addObserver()`, `deleteObserver()` et `notifyObservers()`. Voici comment nous utilisons la version de Java :

Pour qu'un objet devienne Observateur...

Comme d'habitude, implémenter l'Interface observateur (cette fois `java.util.Observer`) et appeler la méthode `addObserver()` de l'objet `Observable`. De même, pour supprimer un observateur, il suffit d'appeler `deleteObserver()`.

Pour que l'Observable envoie des notifications...

Tout d'abord, il faut qu'il devienne `Observable` en étendant la superclasse `java.util.Observable`. Puis l'on procède en deux étapes :

- ❶ Vous devez d'abord appeler la méthode `setChanged()` pour signifier que l'état de votre objet a changé
- ❷ Puis vous appelez l'une des deux méthodes `notifyObservers()` :

soit `notifyObservers()` **soit** `notifyObservers(Object arg)`

Dans cette version, un objet donnée arbitraire est transmis à chaque observateur lors de la notification.

Pour qu'un observateur reçoive des notifications...

Il implémente la méthode `update()` comme auparavant, mais la signature de la méthode diffère légèrement :

`update(Observable o, Object arg)`

Le Sujet qui a envoyé la notification est passé ici en argument.

L'objet donnée qui a été transmis à `notifyObservers()`, ou null si aucun objet n'a été spécifié.

objet donnée

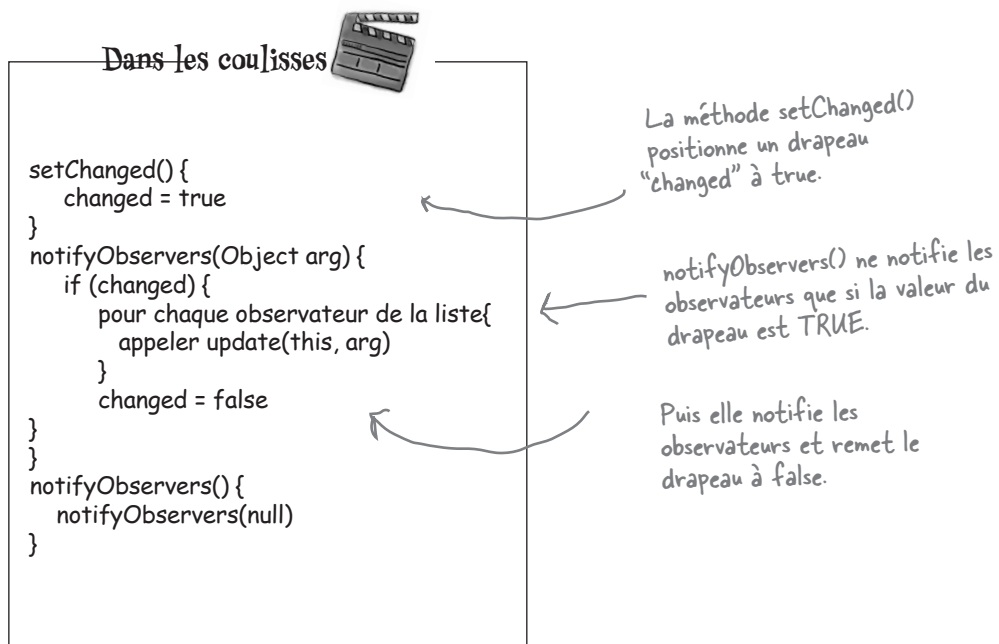
Si vous voulez « pousser » les données vers les observateurs, vous pouvez les transmettre sous forme d'objet donnée à la méthode `notifyObservers(arg)`. Si non, l'Observateur doit « tirer » les données de l'objet `Observable` qui lui a été transmis. Comment ? Retraversons le code de la Station Météo et vous verrez.

Attendez, avant de commencer, pourquoi a-t-on besoin de cette méthode `setChanged()` ? Elle n'existait pas auparavant.



La méthode `setChanged()` sert à signifier que l'état a changé et que `notifyObservers()`, quand elle est appelée, doit mettre à jour les observateurs. Si `notifyObservers()` est appelée sans qu'on ait d'abord appelé `setChanged()`, les observateurs ne seront PAS notifiés. Jetons un œil dans les coulisses d'`Observable` pour voir comment cela fonctionne :

Pseudocode de la classe `Observable`.



Pourquoi est-ce nécessaire ? La méthode `setChanged()` est conçue pour vous autoriser plus de souplesse dans la façon dont vous mettez à jour les observateurs en vous permettant d'optimiser les notifications. Par exemple, dans le cas de notre station météo, imaginez que nos capteurs soient si sensibles que les températures affichées varient constamment de quelques dixièmes de degré. L'objet `DonneesMeteo` pourrait alors émettre des notifications en permanence. Mais si nous voulons émettre des notifications moins fréquentes, nous n'appellerons `setChanged()` que lorsque la température aura augmenté ou diminué d'un demi degré.

Vous n'utiliserez peut-être pas très souvent cette fonctionnalité, mais elle existe en cas de besoin. Dans tous les cas, vous devez appeler `setChanged()` pour que les notifications fonctionnent. Vous pouvez également utiliser la méthode `clearChanged()`, qui réinitialise le drapeau « changed » à faux et la méthode `hasChanged()` qui vous indique la valeur courante du drapeau.

Retravailler la Station Météo avec le pattern intégré

Récrivons d'abord `DonneesMeteo` pour qu'elle étende `java.util.Observable`

1 Assurons-nous d'importer les bonnes classes `Observer / Observable`.

```
import java.util.Observable;
import java.util.Observer;
```

2 Maintenant, nous étendons `Observable`.

```
public class DonneesMeteo extends Observable {
    private float temperature;
    private float humidite;
    private float pression;
```

3 Nous n'avons plus besoin de mémoriser nos observateurs ni de gérer leur ajout et leur suppression : la superclasse s'en charge. Nous avons donc supprimé les méthodes correspondantes.

4 Notre constructeur n'a pas besoin de créer une structure de données pour contenir les observateurs.

```
    public DonneesMeteo() { }
```

5 Nous appelons maintenant `setChanged()` pour indiquer que l'état a changé, avant d'appeler `notifyObservers()`.

```
    public void actualiserMesures() {
        //setChanged();
        notifyObservers(); *
    }
```

6 Ces méthodes ne sont pas nouvelles, mais comme nous allons "tirer" nous avons pensé à vous rappeler leur existence. Les observateurs les utiliseront pour accéder à l'état de l'objet `DonneesMeteo`.

```
    public void setMesures(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        this.pression = pression;
        actualiserMesures();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidite() {
        return humidite;
    }

    public float getPression() {
        return pression;
    }
}
```

*Remarquez que nous n'envoyons pas d'objet donnée avec l'appel de `notifyObservers()`. Autrement dit, nous appliquons le modèle TIRER.

Réécrivons maintenant AffichageConditions

1 Nous importons de nouveau les bonnes classes `Observer` / `Observable`.

2 Nous implémentons maintenant l'interface `java.util.Observer`

```
import java.util.Observable;
import java.util.Observer;

public class AffichageConditions implements Observer, Affichage {
    Observable observable;
    private float temperature;
    private float humidite;

    public AffichageConditions(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof DonneesMeteo) {
            DonneesMeteo donneesMeteo = (DonneesMeteo)obs;
            this.temperature = donneesMeteo.getTemperature();
            this.humidite = donneesMeteo.getHumidite();
            afficher();
        }
    }

    public void afficher() {
        System.out.println("Conditions actuelles : " + temperature
            + " degrés C et " + humidite + " % d'humidité");
    }
}
```

3 Notre constructeur accepte maintenant un `Observable` et nous utilisons « `this` » pour enregistrer l'objet en tant qu'observateur.

4 Nous avons modifié la méthode `update()` pour qu'elle accepte en argument à la fois un `Observable` et des données optionnelles.

5 Dans `update()`, nous nous assurons d'abord que l'observable est de type `DonneesMeteo`, puis nous appelons ses méthodes `get` pour obtenir la température et l'humidité. Puis nous appelons `afficher()`.



Le frigo



La classe AffichagePrevisions a été découpée en fragments, et ceux-ci ont été affichés dans le désordre sur la porte du frigo. Pouvez-vous les réorganiser pour que le programme fonctionne ? Certaines accolades sont tombées par terre et elles sont trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant qu'il faut !



```
public AffichagePrevisions(Observable
observable) {
```

```
    afficher();
```

```
    observable.addObserver(this);
```

```
    if (observable instanceof DonneesMeteo) {
```

```
        public class AffichagePrevisions implements
        Observer, Affichage {
```

```
            public void afficher() {
                // code pour afficher
            }
        }
```

```
        dernierePression = pressionActuelle;
        pressionActuelle = donneesMeteo.getPression();
```

```
        private float pressionActuelle = 1012;
        private float dernierePression;
```

```
        DonneesMeteo donneesMeteo =
        (DonneesMeteo) observable;
```

```
        public void update(Observable observable,
        Object arg) {
```

```
            import java.util.Observable;
            import java.util.Observer;
```


Exécuter le nouveau code

Par sécurité, exécutons le nouveau code...

```
Fichier Édition fenêtre Aide DevoirALaMaison
%java StationMeteo
Prévision : Amélioration en cours !
Température Moy/Max/Min = 26.0/26.0/26.0
Conditions actuelles : 26 degrés C et 65.0 % d'humidité
Prévision : Le temps se rafraîchit
Température Moy/Max/Min = 27.0/28.0/26.0
Conditions actuelles : 28 degrés C et 70.0 % d'humidité
Prévision : Dépression bien installée
Température Moy/Max/Min = 25.0/28.0/22.0
Conditions actuelles : 22 degrés C et 90.0 % d'humidité
%
```

Mmm, avez-vous remarqué une différence ? Regardez encore...

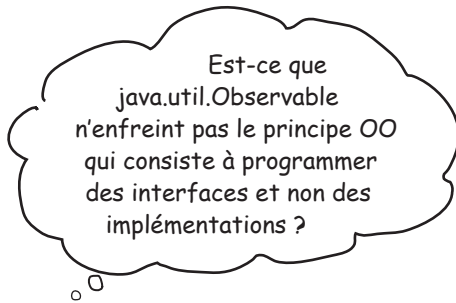
Vous voyez les mêmes calculs, mais l'ordre du texte a mystérieusement changé.
Que s'est-il passé ? Réfléchissez une minute avant de poursuivre votre lecture...

Ne jamais dépendre de l'ordre d'évaluation des notifications

Dans `java.util.Observable`, la méthode `notifyObservers()` est implémentée de telle façon que les observateurs sont notifiés dans un *autre* ordre que dans votre propre implémentation. Qui a raison ?

Ni l'un ni l'autre ; nous avons simplement choisi de procéder différemment.

Ce qui serait incorrect, en revanche, ce serait d'écrire un programme qui *dépende* d'un ordre de notification spécifique. Pourquoi ? Parce que si vous devez modifier les implémentations d'`Observable/Observer`, cet ordre pourrait changer et votre application produirait des résultats erronés. Ce n'est *pas* précisément ce que l'on considère comme un couplage faible.



La face cachée de java.util.Observable

Oui, bien vu. Comme vous l'avez remarqué, Observable est une *classe*, non une *interface*. Pis encore, elle *n'implémente* même pas une interface. Malheureusement, l'implémentation de java.util.Observable présente un certain nombre de problèmes qui limitent son utilité et ses possibilités de réutilisation. Non qu'elle soit inutile, mais elle présente de gros défauts auxquels il convient de faire attention.

Observable est une classe

Nos principes vous ont déjà appris que c'était là une mauvaise idée, mais quel est exactement le problème ?

Tout d'abord, comme Observable est une *classe*, vous devez la *sous-classer*. Autrement dit, vous ne pouvez pas ajouter le comportement d'Observable à une classe existante qui étend déjà une autre superclasse. Cela limite son potentiel de réutilisation (la principale raison a priori pour laquelle nous utilisons des patterns).

Deuxièmement, comme il n'y a pas d'interface Observable, vous ne pouvez même pas créer vous-même une implémentation qui fonctionne correctement avec l'API Observer Java intégrée. Pas plus que vous ne pouvez remplacer l'implémentation de java.util par une autre (par exemple une nouvelle implémentation multithread).

Observable protège les méthodes cruciales

Si vous observez l'API Observable, vous constatez que la méthode `setChanged()` est protégée. Et alors ? Eh bien, cela signifie que vous ne pouvez pas appeler `setChanged()` tant que vous n'avez pas sous-classé Observable. Autrement dit, vous ne pouvez même pas créer une instance de la classe Observable et composer avec vos propres objets : vous êtes obligé de sous-classer. Cette façon de procéder viole un deuxième principe de conception...*préférer la composition à l'héritage*.

Que faire ?

Observable *peut* correspondre à vos besoins si vous pouvez étendre java.util.Observable. D'un autre côté, vous aurez peut-être besoin de développer votre propre implémentation Comme nous l'avons fait au début de ce chapitre. Dans les deux cas, vous connaissez bien le pattern Observateur et vous êtes bien placé pour travailler avec une API qui l'utilise.

Autres endroits où se trouve le pattern Observateur dans le JDK

L'implémentation de Observer/Observable dans java.util n'est pas le seul endroit où vous trouverez le pattern Observateur dans le JDK : les JavaBeans et l'API Swing possèdent également leurs propres implémentations du pattern. À ce stade, vous connaissez suffisamment bien Observateur pour explorer tout seul ces API, mais nous allons voir rapidement un exemple simple avec Swing, juste pour le plaisir.

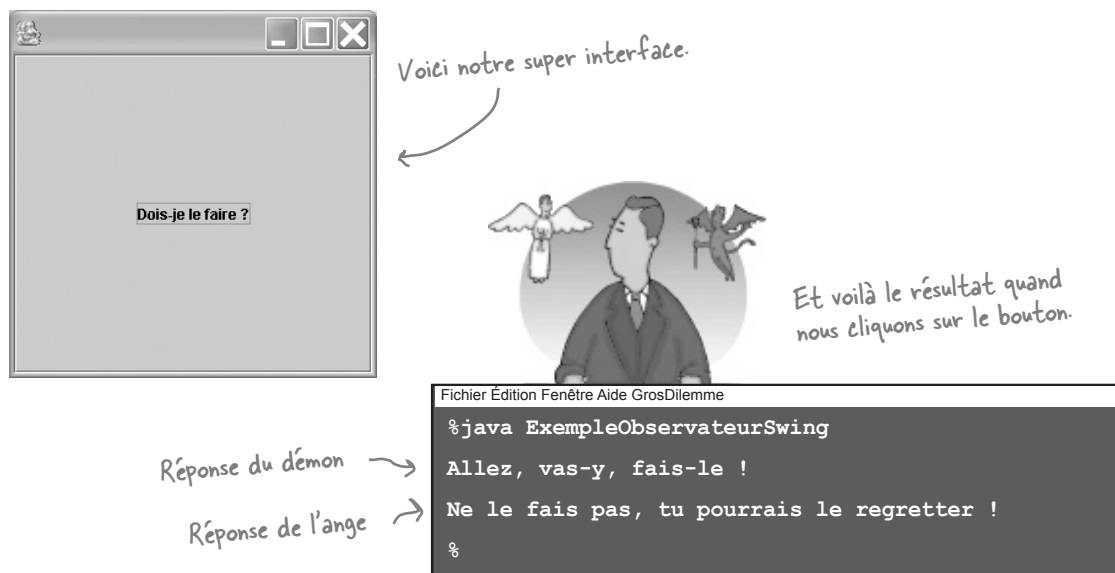
Un peu de contexte...

Examinons un composant simple de l'API Swing, le JButton. Si vous regardez rapidement la superclasse de JButton, AbstractButton, vous verrez qu'elle a un tas de méthodes «add» et «remove». Ces méthodes vous permettent d'ajouter et de supprimer des observateurs, ou, comme on les appelle dans la terminologie Swing, des auditeurs, afin d'être à l'écoute des différents types d'événements qui se produisent dans le composant Swing. Par exemple, un ActionListener vous permet d'«écouter» tous les types d'actions qui peuvent être appliquées à un bouton, comme les clics. Vous trouverez différents types d'auditeurs dans toute l'API Swing.

Si le pattern Observateur dans les JavaBeans vous intéresse, regardez l'interface PropertyChangeListener.

Une petite application qui change la vie

Bon, notre application est extrêmement simple. Nous avons un bouton qui dit «Dois-je le faire ?». Quand vous cliquez sur ce bouton, les auditeurs (observateurs) répondent à la question à leur guise. Nous implémentons ces deux auditeurs, nommés AuditeurAnge et AuditeurDémon. Voici comment l'application se comporte :



Et le code...

Cette application indispensable nécessite très peu de code. Il suffit de créer un objet JButton, de l'ajouter à un JFrame et de définir nos auditeurs. Pour ces derniers, nous allons utiliser des classes internes, technique très courante en programmation Swing. Si vous n'êtes pas au point sur les classes internes ou sur Swing, vous pouvez revoir les chapitres sur les interfaces graphiques dans Java Tête la première.

```
public class ExempleObservateurSwing {
    JFrame cadre;

    public static void main(String[] args) {
        ExempleObservateurSwing exemple = new ExempleObservateurSwing ();
        exemple.go();
    }

    public void go() {
        cadre = new JFrame();
        JButton bouton = new JButton("Dois-je le faire ?");
        bouton.addActionListener(new AuditeurAnge());
        bouton.addActionListener(new AuditeurDemon());
        cadre.getContentPane().add(BorderLayout.CENTER, bouton);
        // Définir les propriétés du cadre ici
    }

    class AuditeurAnge implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Ne le fais pas, tu pourrais le regretter!");
        }
    }

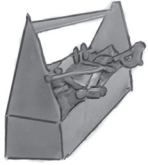
    class AuditeurDemon implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Allez, vas-y, fais-le ");
        }
    }
}
```

Simple application Swing qui se borne à créer un cadre et à y insérer un bouton.

Créer les objets ange et démon (les observateurs) qui écoutent les événements du bouton.

Voici les définitions de classes des observateurs. Ce sont des classes internes (mais ce n'est pas obligatoire).

Au lieu d'update(), la méthode actionPerformed() est appelée quand l'état du sujet (en l'occurrence le bouton) change.



Votre boîte à outils de concepteur

Félicitations, vous êtes bientôt à la fin du chapitre 2. Vous venez d'ajouter quelques éléments à votre boîte à outils OO...

Bases de l'OO

Principes OO

Encapsulez ce qui varie.
Préférez la composition à l'héritage.
Programmez des interfaces, non des implémentations.
Efforcez-vous de coupler faiblement les objets qui interagissent.

Voici notre nouveau principe. Souvenez-vous que les conceptions faiblement couplées sont plus beaucoup plus souples et plus résistantes aux changements.

Patterns OO

Observateur – définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Nouveau pattern pour communiquer un état à un ensemble d'objets de manière faiblement couplée. Nous n'avons pas vu tout le pattern Observateur – attendez un peu que nous parlions de MVC !

POINTS D'IMPACT

- Le pattern Observateur définit entre des objets une relation de type un-à-plusieurs.
- Les Sujets, alias Observables, mettent à jour les Observateurs via une interface commune.
- Les Observateurs sont faiblement couplés, au sens où l'Observable ne sait rien d'eux, en dehors du fait qu'ils implémentent l'interface Observer.
- Vous pouvez « pousser » ou « tirer » les données de l'Observable (tirer est considéré comme plus « correct »).
- Ne faites pas dépendre les observateurs de l'ordre des notifications.
- Java a plusieurs implémentations du pattern Observateur, notamment la classe généraliste Observable dans le package java.util.
- Attention aux problèmes potentiels de l'implémentation de java.util.Observable.
- N'ayez pas peur de créer votre propre implémentation d'Observable si nécessaire.
- Swing utilise intensivement le pattern Observateur, à l'instar de nombreux frameworks de création d'IHM.
- Vous trouverez le pattern à de nombreux autres endroits, en particulier les JavaBeans et RMI.



Principes de conception

Pour chaque principe de conception, dites comment le pattern Observateur l'applique.

Principe de conception

Identifiez les aspects de votre application qui varient et séparez-les de ce qui demeure inchangé.

Principe de conception

Programmez des interfaces, non des implémentations.

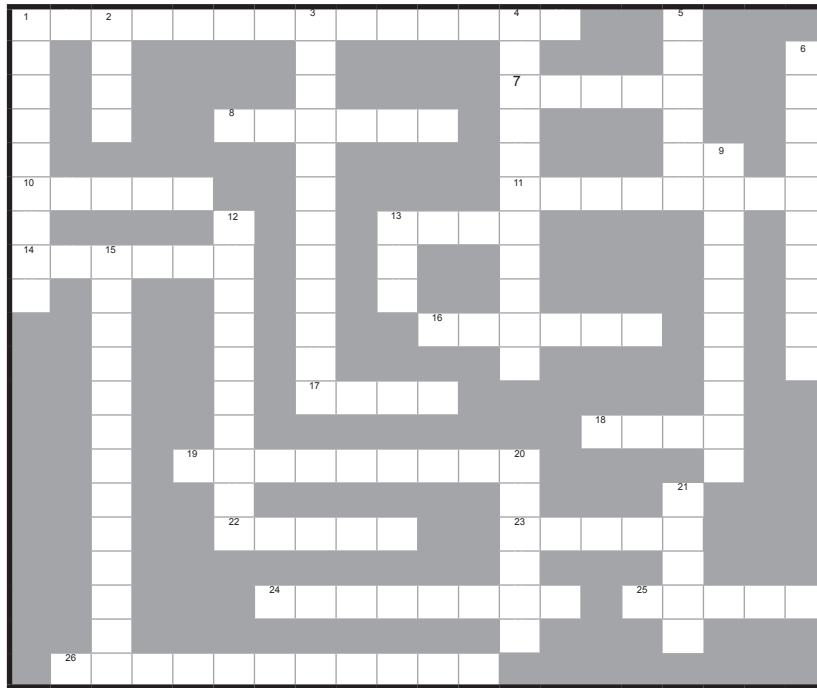
Principe de conception

Préférez la composition à l'héritage.

Cette question est difficile. Indice : pensez à la façon dont sujets et observateurs interagissent.



Il est temps de redonner quelque chose à faire à votre cerveau droit !
Cette fois, tous les mots de la solution sont dans le chapitre 2.



Horizontalement

1. Programmez pour une _____, non pour une implémentation.
7. Package Java plein d'observateurs.
8. Plus le couplage est _____, mieux c'est.
10. Ce type est vraiment primitif.
11. Peut être locale, globale...
13. Tout le code de ce livre est écrit en _____.
14. Observable est une _____, non une interface.
17. Inapplicable ou inconnu.
18. Contraire de push.
19. On l'implémente.
22. Peut être réel ou non.
23. Ne comptez pas dessus pour les notifications.
24. Peut être relative.
25. Indique le temps qu'il fait.
26. L'un des paramètres mesurés la station.

Verticalement

1. Programmez pour une interface, non pour une _____
2. Envers de pull.
3. Les observateurs reçoivent une _____ des changements.
4. Le héros de ce chapitre.
5. L'un des écouteurs de la page 72.
6. La classe DonneesMeteo _____ l'interface Sujet.
9. Gère les observateurs à votre place.
12. Les observateurs sont _____ du sujet.
13. Elle exécute du bytecode.
15. On mesure la pression _____.
16. Génératrice d'événements.
20. Un auditeur _____.
21. Léo et Maud rencontrent un chasseur de _____.



Solutions des exercices

À vos crayons

D'après notre première implémentation, quels énoncés suivants sont vrais ?
(Plusieurs réponses possibles.)

- | | |
|------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <input checked="" type="checkbox"/> A. Nous codons des implémentations concrètes, non des interfaces. | <input type="checkbox"/> D. Les éléments d'affichage n'implémentent pas une interface commune. |
| <input checked="" type="checkbox"/> B. Nous devons modifier le code pour chaque nouvel élément d'affichage. | <input checked="" type="checkbox"/> E. Nous n'avons pas encapsulé les parties qui varient. |
| <input checked="" type="checkbox"/> C. Nous n'avons aucun moyen d'ajouter (ou de supprimer) des éléments d'affichage au moment de l'exécution. | <input type="checkbox"/> F. Nous violons l'encapsulation de la classe <code>DonneesMeteo</code> . |



Principes de conception

Principe de conception

Identifiez les aspects de votre application qui varient et séparez-les de ce qui demeure inchangé.

Les éléments qui varient dans le pattern Observateur sont l'état du Sujet et le nombre et le type des Observateurs. Ce pattern vous permet de faire varier les objets qui dépendent de l'état du Sujet, sans devoir modifier de Sujet. C'est ce qu'on appelle de la planification !

Principe de conception

Programmez des interfaces, non des implémentations

Le Sujet et l'Observateur utilisent tous deux des interfaces. Le Sujet mémorise les objets implémentant l'interface Observateur, tandis que les observateurs s'enregistrent auprès de l'interface Sujet et reçoivent ses notifications. Comme nous l'avons vu, cette technique permet de conserver une conception élégante et faiblement couplée.

Principe de conception

Préférez la composition à l'héritage.

Le pattern Observateur utilise la composition pour composer un nombre quelconque d'Observateurs avec leur Sujet. Ces relations ne sont pas définies par une quelconque hiérarchie d'héritage. Non, elles sont définies par la composition au moment de l'exécution !



Le frigo



Solutions des exercices

```
import java.util.Observable;
import java.util.Observer;

public class AffichagePrevisions implements
Observer, Affichage {

    private float pressionActuelle = 1012;
    private float dernierePression;

    public AffichagePrevisions(Observable
observable) {

        DonneesMeteo donneesMeteo
        =(DonneesMeteo) observable;

        observable.addObserver(this);

    }

    public void update(Observable observable,
Object arg) {

        if (observable instanceof DonneesMeteo) {

            dernierePression = pressionActuelle;
            pressionActuelle = donneesMeteo.getPression();

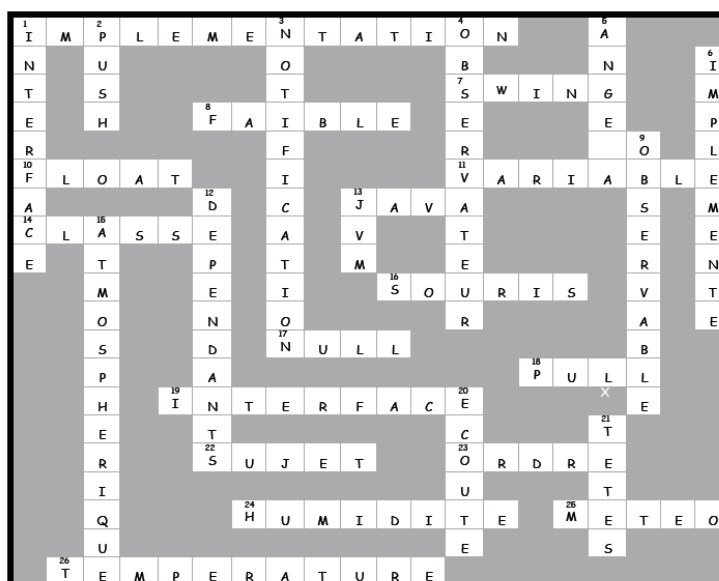
            afficher();

        }

    }

    public void afficher() {
        // code pour afficher
    }

}
```



3 Le pattern Décorateur

✧ *Décorer les objets* ✧



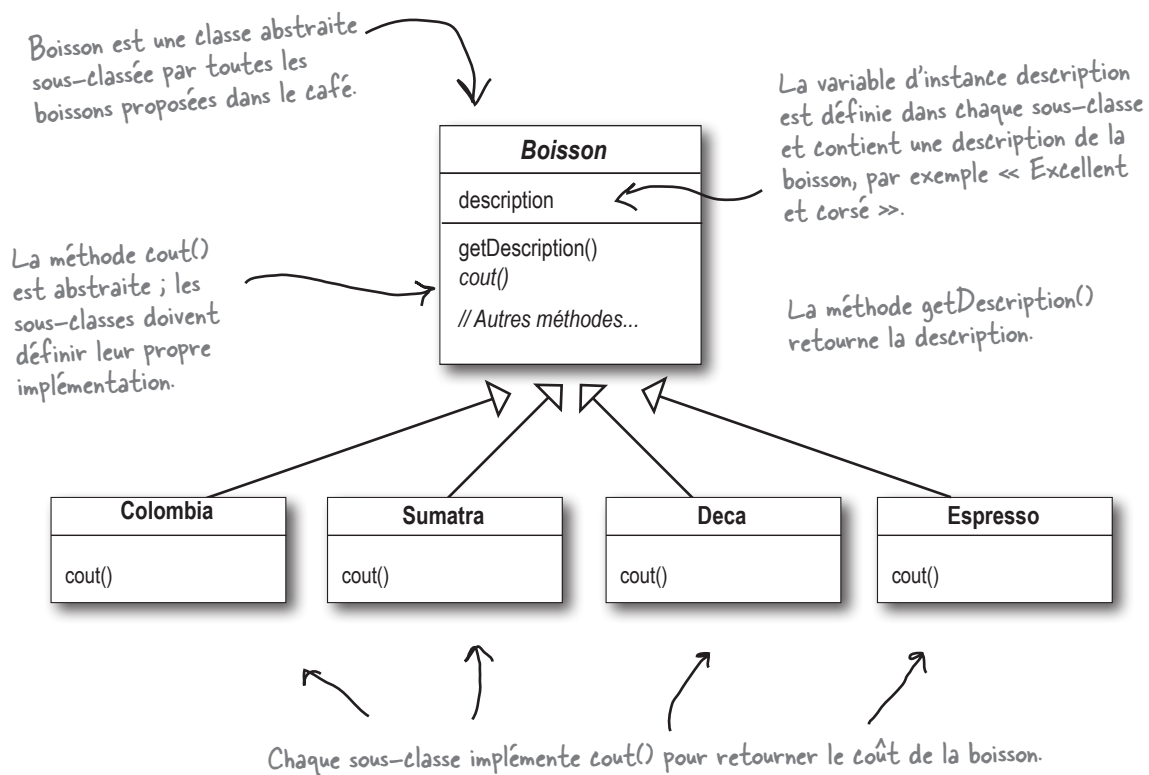
Ce chapitre pourrait s'appeler « Les bons concepteurs se méfient de l'héritage ». Nous allons revoir la façon dont on abuse généralement de l'héritage et vous allez apprendre comment « décorer » vos classes au moment de l'exécution en utilisant une forme de composition. Pourquoi ? Une fois que vous connaîtrez les techniques de décoration, vous pourrez affecter à vos objets (ou à ceux de quelqu'un d'autre) de nouvelles responsabilités *sans jamais modifier le code des classes sous-jacentes*.

Bienvenue chez Starbuzz Coffee

Starbuzz Coffee s'est fait un nom en devenant la plus importante chaîne de « salons de café ».

Si vous en voyez un au coin de la rue, tournez la tête : vous en verrez un autre. Comme ils se sont développés très rapidement, ils se précipitent pour mettre à jour leurs systèmes de prise de commandes afin de l'adapter à leur offre.

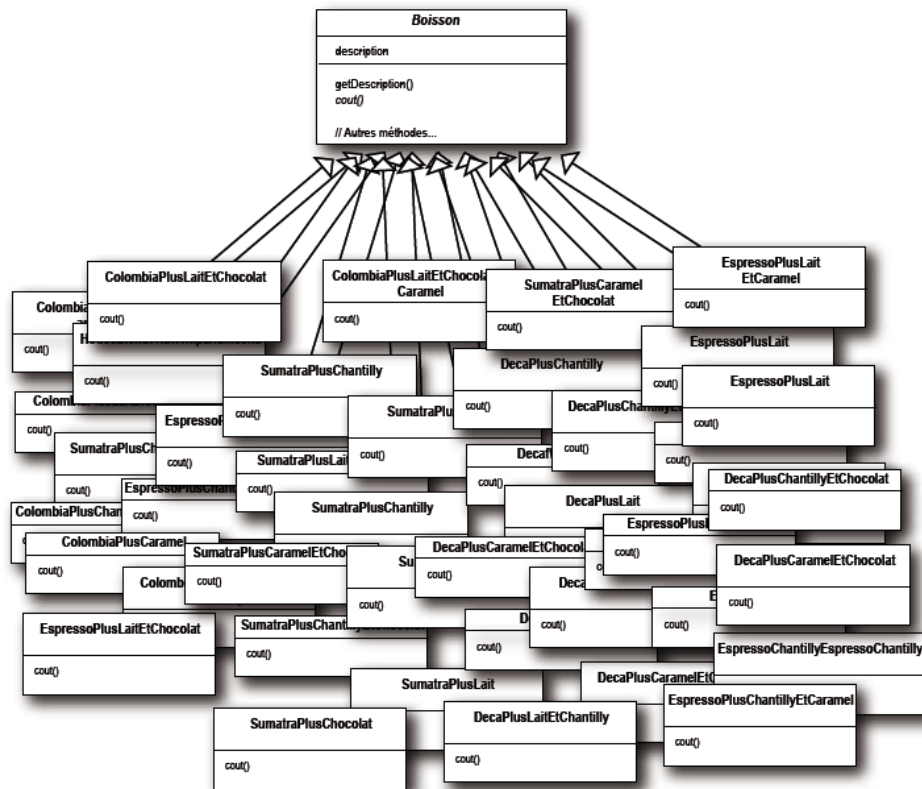
Quand ils ont commencé, ils ont conçu leurs classes comme ceci...



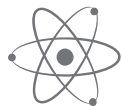
Les salons de café de ce type emploient le vocabulaire italien du café : espresso, capuccino, etc. C'est aussi pourquoi les serveurs et serveuses de la page 94 sont des baristas.

En plus de votre café, vous pouvez également demander plusieurs ingrédients comme de la mousse de lait, du caramel, du chocolat, du sirop de vanille ou de noisette et couronner le tout de crème Chantilly. Starbuzz facturant chacun de ces suppléments, ils ont vraiment besoin de les intégrer à leur système de commande.

Voici leur premier essai...



↑
Chaque méthode `cout()` calcule le coût du café plus celui des autres ingrédients de la commande.



MUSCLES VOS NEURONES

De toute évidence, Starbuzz s'est fourré tout seul dans un vrai cauchemar. Que se passe-t-il quand le prix du lait augmente ? Que font-ils quand ils proposent un nouveau supplément de vanille ?

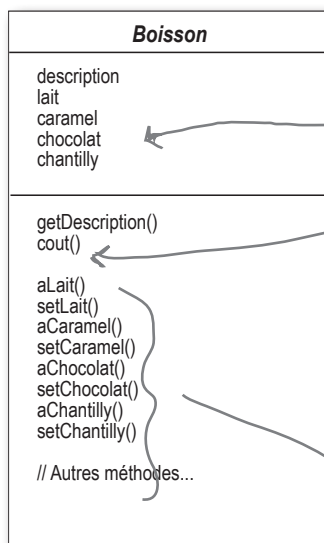
Au-delà du problème de maintenance, ils enfreignent l'un des principes de conception que nous avons déjà vus. Lequel ?

Indication : C'est même une infraction majeure !

C'est stupide. Pourquoi avons-nous besoin de toutes ces classes ? Pourquoi ne pas utiliser des variables d'instance et l'héritage dans la superclasse pour mémoriser les ingrédients ?



Eh bien, essayons. Commençons par la classe de base Boisson et ajoutons des variables d'instance pour représenter si chaque boisson contient ou non du lait, du caramel, du chocolat et/ou de la chantilly...



Nouvelles valeurs booléennes pour chaque ingrédient.

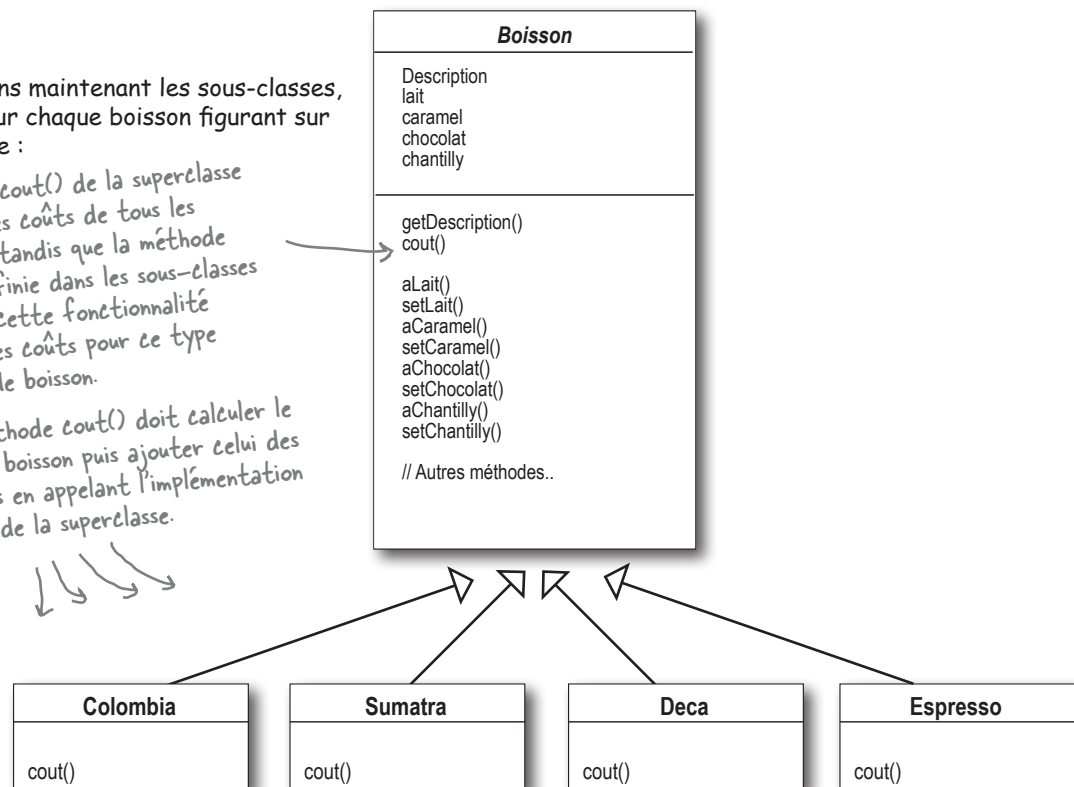
Maintenant, nous implémentons `cout()` dans Boisson (au lieu qu'elle demeure abstraite), pour qu'elle puisse calculer les coûts associés aux ingrédients pour une instance de boisson donnée. Les sous-classes redéfiniront toujours `cout()`, mais elles appelleront également la super-version pour pouvoir calculer le coût total de la boisson de base plus celui des suppléments.

Ces méthodes lisent et modifient les valeurs booléennes des ingrédients.

Ajoutons maintenant les sous-classes, une pour chaque boisson figurant sur la carte :

La méthode `cout()` de la superclasse va calculer les coûts de tous les ingrédients, tandis que la méthode `cout()` redéfinie dans les sous-classes va étendre cette fonctionnalité et inclure les coûts pour ce type spécifique de boisson.

Chaque méthode `cout()` doit calculer le coût de la boisson puis ajouter celui des ingrédients en appelant l'implémentation de `cout()` de la superclasse.



À vos crayons

Écrivez les méthodes `cout()` pour les classes suivantes (du pseudo Java suffit) :

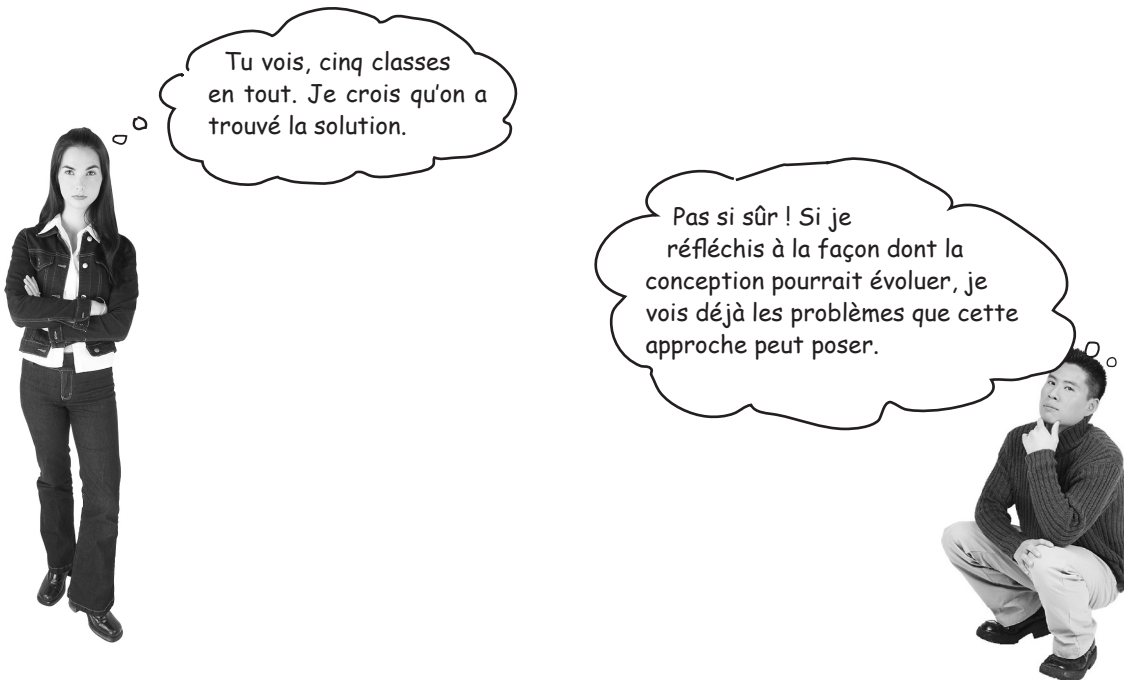
```
public class Boisson {
    public double cout() {
```

```
    }
}
```

```
public class Sumatra extends Boisson {
```

```
    public Sumatra() {
        description = "Délicieux et corsé";
    }
    public double cout() {
```

```
    }
}
```

À vos crayons

Quelles sont les exigences ou les autres facteurs qui pourraient changer et avoir un impact sur cette conception ?

L'augmentation du prix des ingrédients nous obligera à modifier le code existant.

De nouveaux ingrédients nous forceront à ajouter de nouvelles méthodes et à modifier la méthode `cout()` dans la superclasse.

Nous pouvons avoir de nouvelles boissons. Pour certaines de ces boissons (thé glacé?), les ingrédients ne seront peut-être plus adaptés, et pourtant la sous-classe `Thé` continuera à hériter de la méthode `aChantilly()`.

Que se passe-t-il si le client veut un double supplément de chocolat ?

Comme nous l'avons vu au chapitre 1, c'est une très mauvaise idée !

À vous :



Maître et disciple...

Maître : Il y a longtemps que nous ne nous sommes pas vus, petit scarabée. Est-ce que tu as profondément médité sur l'héritage ?

Disciple : Oui, maître. J'ai appris que l'héritage était puissant, mais que ce n'était pas la meilleure voie pour obtenir des conceptions souples et faciles à maintenir.

Maître : Ah, je vois que tu as fait des progrès. Mais dis-moi, scarabée, comment parviendras-tu à la réutilisation si ce n'est par l'héritage ?

Disciple : Maître, j'ai appris qu'il y avait d'autres moyens d'« hériter » d'un comportement au moment de l'exécution, grâce à la composition et à la délégation.

Maître : Continue, scarabée...

Disciple : Quand j'hérite d'un comportement en sous-classant, ce comportement est défini statiquement au moment de la compilation. De plus, toutes les sous-classes doivent hériter du même comportement. Mais si je peux étendre le comportement d'un objet par la composition, je peux le faire dynamiquement lors de l'exécution.

Maître : Très bien, scarabée, tu commences à voir le pouvoir de la composition.

Disciple : Oui. Cette technique me permet d'ajouter plusieurs nouvelles responsabilités aux objets, y compris des responsabilités auxquelles le concepteur de la superclasse n'avait même pas pensé. Et je n'ai pas besoin de toucher à son code !

Maître : Et qu'as-tu appris sur l'effet de la composition sur la maintenance de ton code ?

Disciple : J'y arrive. En composant dynamiquement des objets, je peux ajouter de nouvelles fonctionnalités en écrivant du code au lieu de modifier le code existant. Comme je ne touche pas au code existant, les risques d'introduire des bogues ou de provoquer des effets de bord inattendus sont significativement réduits.

Maître : Très bien. C'est assez pour aujourd'hui, scarabée. J'aimerais que tu ailles méditer encore un peu sur ce sujet... Souviens-toi : le code doit être fermé (au changement) comme la fleur de lotus le soir, et cependant ouvert (à l'extension) comme la fleur de lotus le matin.

Le principe Ouvert-Fermé

Scarabée est en train de découvrir l'un des plus importants principes de conception :



Principe de conception

Les classes doivent être ouvertes à l'extension, mais fermées à la modification.

Bienvenue ; nous sommes
ouverts. N'hésitez pas à étendre nos classes
avec n'importe quel comportement de votre choix. Si
vos besoins ou vos exigences évoluent (et nous savons
que ce sera le cas), il vous suffira de créer vos propres
extensions

Désolés, nous sommes *fermés*.
Oui, nous avons passé des
heures à écrire ce code
correctement et à éliminer les
bogues. Nous ne pouvons donc
pas vous permettre de
modifier le code existant. Il doit rester fermé
aux modifications. Si ça ne vous plaît pas,
adressez-vous au directeur.



Notre but est de permettre d'étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant. Qu'obtenons-nous si nous y parvenons ? Des conceptions résistantes au changement et suffisamment souples pour accepter de nouvelles fonctionnalités répondant à l'évolution des besoins.

Il n'y a pas de questions stupides

Q: Ouvert à l'extension et fermé à la modification ? Cela semble très contradictoire. Comment pouvons-nous avoir les deux ?

R: Excellente question. Il est vrai que c'est contradictoire à première vue. Après tout, moins une chose est modifiable, plus elle est difficile à étendre, non ? Mais vous allez voir qu'il existe en développement OO tout un tas de moyens fûtés pour étendre un système même si on ne peut pas modifier le code sous-jacent. Pensez au pattern Observateur (au chapitre 2)... En ajoutant des observateurs, on peut étendre le sujet à tout moment, sans ajouter de code à celui-ci. Vous allez bientôt découvrir qu'il existe d'autres moyens d'étendre un comportement grâce à un certain nombre d'autres techniques OO.

Q: OK, je comprends Observable. Mais comment faire d'une façon générale pour concevoir quelque chose d'extensible et pourtant fermé aux modifications ?

R: De nombreux patterns nous offrent des modèles de conception éprouvés qui protègent votre code des modifications tout en fournissant des moyens d'extension. Dans ce chapitre, vous allez voir un bon exemple d'emploi du pattern Décorateur pour appliquer le principe Ouvert-Fermé.

Q: Comment faire pour que toutes les parties de ma conception appliquent le principe Ouvert-Fermé ?

R: C'est généralement impossible. Créer des conceptions OO souples et ouvertes à l'extension sans modifier le code existant demande du temps et des efforts. D'habitude, nous ne pouvons pas nous offrir ce luxe pour chaque partie de nos applications (et ce serait probablement peu rentable). L'application du principe Ouvert-Fermé introduit généralement de nouveaux niveaux d'abstraction, ce qui rend le code plus complexe. Vous devez vous concentrer sur les zones qui sont le plus susceptibles de changer et c'est là qu'il faut appliquer ce principe.

Q: Comment savoir quels sont les points de variation les plus importants ?

R: C'est en partie une question d'expérience en conception de systèmes OO, et cela demande également de connaître le domaine dans lequel vous travaillez. L'étude d'autres exemples vous aidera à comprendre comment identifier ces points d'évolution dans votre propre contexte.

Malgré la contradiction apparente, il existe des techniques qui permettent d'étendre le code sans le modifier directement.

Soyez attentif lorsque vous choisissez les points du code qui doivent être étendus. Appliquer le principe Ouvert-Fermé PARTOUT est inutile, peu rentable et susceptible de déboucher sur un code complexe et difficile à comprendre.

Faites la connaissance du *pattern Décorateur*

Bien. Nous avons vu que représenter notre boisson plus le schéma de tarification des ingrédients au moyen de l'héritage n'a pas très bien fonctionné – nous obtenons, une explosion de classes, une conception rigide, ou nous ajoutons à la classe de base une fonctionnalité inappropriée pour certaines des sous-classes.

Voici donc ce que nous allons faire. Nous allons commencer par une boisson et nous allons la « décorer » avec des ingrédients au moment de l'exécution. Si par exemple le client veut un Sumatra avec Chocolat et Chantilly, nous allons :

Bon, ça suffit « le club des concepteurs objet ». Nous avons de vrais problèmes ici ! Vous vous souvenez de nous ? Starbuzz Coffee ? Pensez-vous que vous pourriez appliquer certains de ces principes de conception pour nous aider vraiment ?



- ❶ **Prendre un objet Sumatra.**
- ❷ **Le décorer avec un objet Chocolat.**
- ❸ **Le décorer avec un objet Chantilly.**
- ❹ **Appeler la méthode `cout()` et nous appuyer sur la délégation pour ajouter les coûts des ingrédients.**

Parfait. Mais comment fait-on pour « décorer » un objet, et que vient faire ici la délégation ?

Indice : représentez-vous les objets décorateurs comme des « enveloppes ». Voyons comment cela fonctionne...

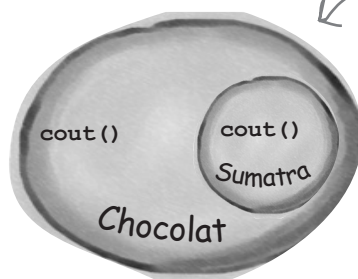
Construire une boisson avec des décorateurs

1 Commençons par notre objet Sumatra.



Souvenez-vous que Sumatra hérite de Boisson et a une méthode `cout()` qui calcule le prix de la boisson.

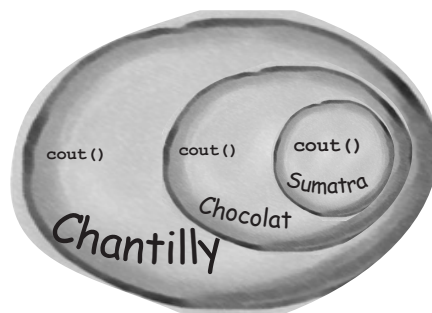
2 Le client veut du chocolat. Nous créons donc un objet Chocolat et nous enveloppons le Sumatra dedans.



L'objet Chocolat est un décorateur. Son type reflète l'objet qu'il décore, en l'occurrence une Boisson. (Par « reflète », nous voulons dire qu'il est du même type.)

Chocolat a donc également une méthode `cout()`. Grâce au polymorphisme, nous pouvons traiter n'importe quelle Boisson enveloppée de Chocolat comme une Boisson (parce que Chocolat est un sous-type de Boisson).

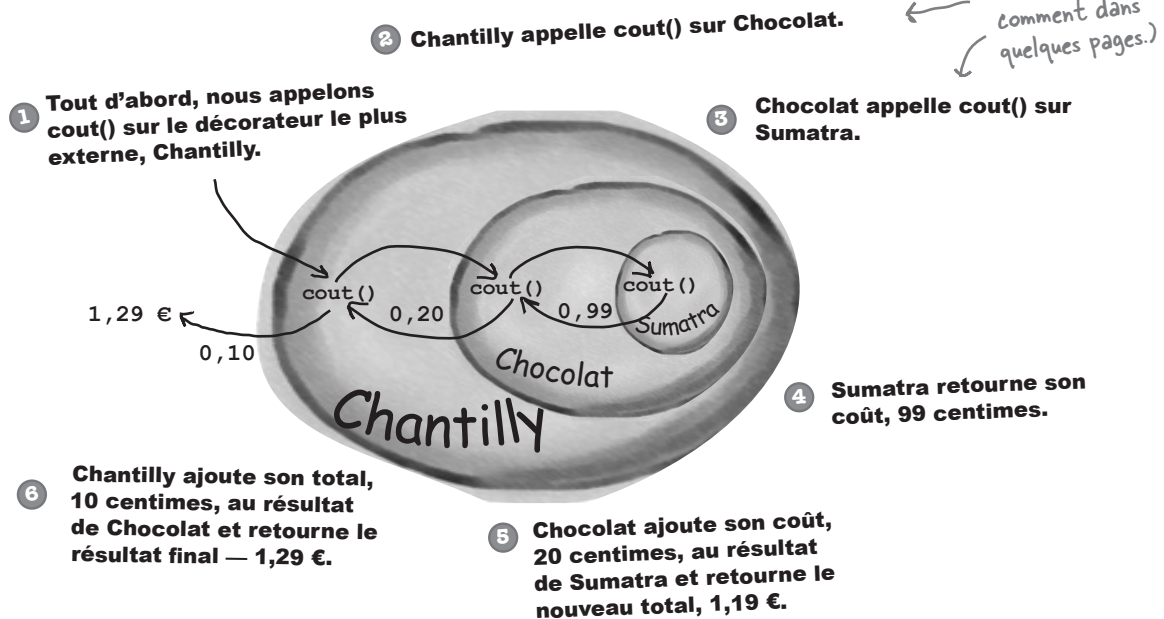
3 Le client veut aussi de la Chantilly. Nous créons un décorateur Chantilly et nous enveloppons Chocolat dedans.



Chantilly est un décorateur. Il reflète également le type de Sumatra et possède une méthode `cout()`.

Ainsi, un Sumatra enveloppé de Chocolat et de Chantilly est toujours une Boisson. Nous pouvons lui faire tout ce que nous ferions avec un Sumatra, notamment appeler sa méthode `cout()`.

- ④ Il est temps de calculer le coût pour le client. Pour ce faire, nous appelons `cout()` sur le décorateur le plus externe, Chantilly, et Chantilly va déléguer le calcul du coût à l'objet qu'il décore. Une fois qu'il aura le coût, il ajoutera le coût de Chantilly.



Bien. Voici ce que nous savons jusqu'à maintenant...

- Les décorateurs ont le même supertype que les objets qu'ils décorent.
- Vous pouvez utiliser un ou plusieurs décorateurs pour envelopper un objet.
- Comme le décorateur a le même supertype que l'objet qu'il décore, nous pouvons transmettre un objet décoré à la place de l'objet original (enveloppé).
- Le décorateur ajoute son propre comportement soit avant soit après avoir délégué le reste du travail à l'objet qu'il décore.
- Les objets pouvant être décorés à tout moment, nous pouvons les décorer dynamiquement au moment de l'exécution avec autant de décorateurs que nous en avons envie.

Point-clé !

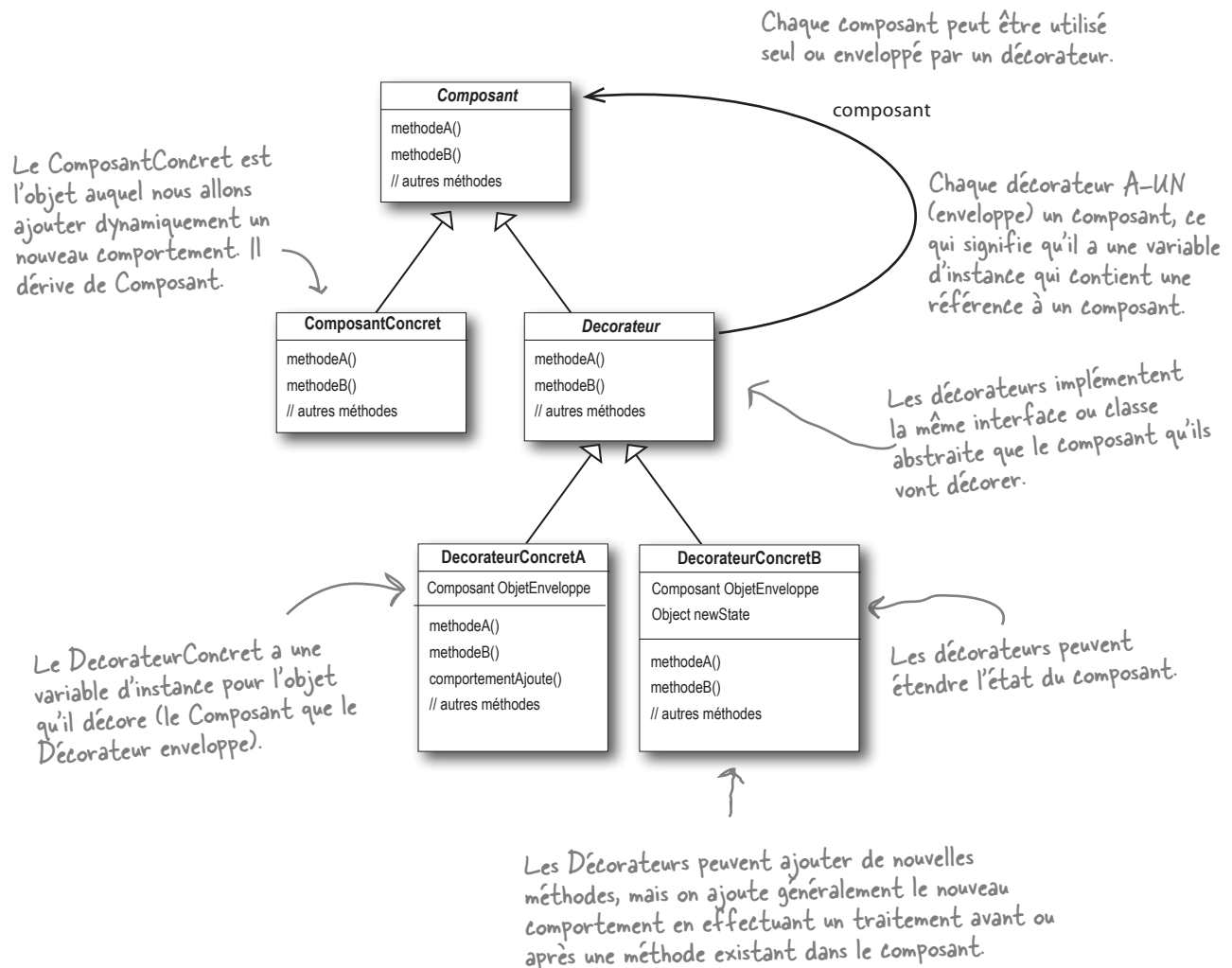
Voyons maintenant comment tout ceci fonctionne réellement en étudiant la définition du pattern *Décorateur* et en écrivant un peu de code.

Le pattern Décorateur : définition

Jetons d'abord un coup d'œil à la définition du pattern Décorateur :

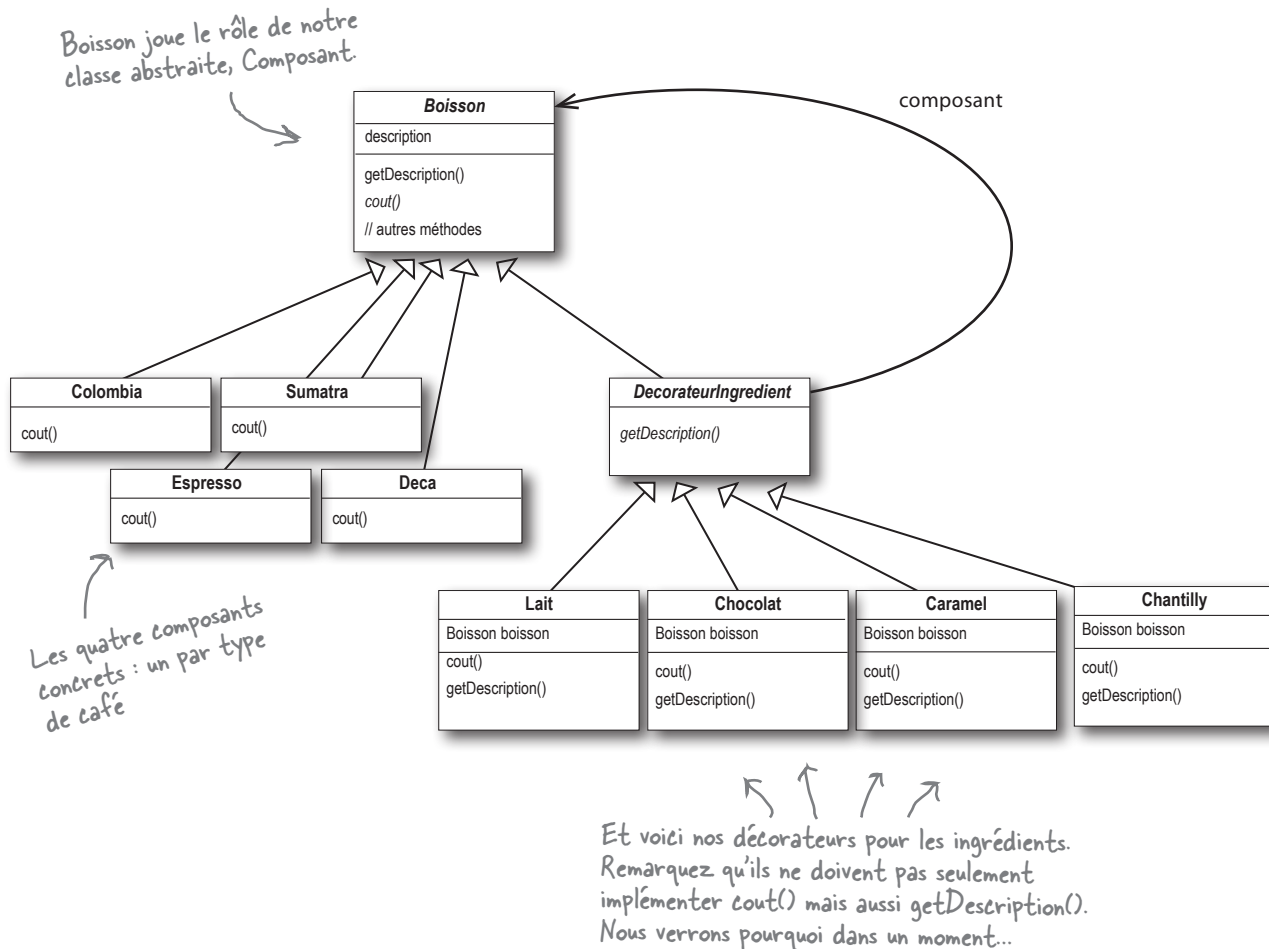
Le pattern Décorateur attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.

Si cet énoncé décrit bien le rôle du pattern Décorateur, elle ne nous donne pas beaucoup d'indications sur la façon de l'appliquer à notre propre implémentation. Regardons le diagramme de classes, un peu plus révélateur (à la page suivante, nous verrons la même structure appliquée au problème des boissons).



Décorons nos Boissons

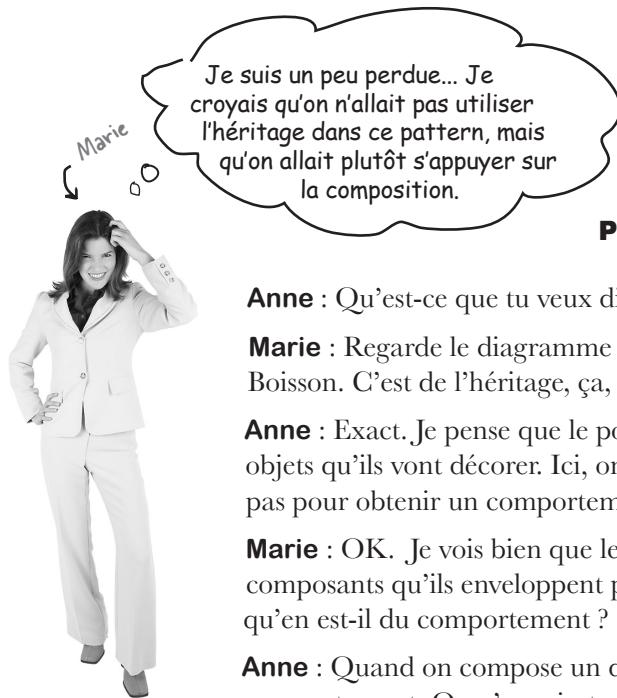
Parfait. C'est dans ce cadre que nous allons travailler sur les boissons de Starbuzz...



MUSCLEZ VOS NEURONES

Avant d'aller plus loin, réfléchissez à la façon dont vous implémenteriez la méthode `cout()` pour le café puis les ingrédients. Réfléchissez également à l'implémentation de la méthode `getDescription()` des ingrédients.

Conversation dans un box



Petite mise au point : héritage vs. composition

Anne : Qu'est-ce que tu veux dire ?

Marie : Regarde le diagramme de classes. Le DecorateurIngredient étend toujours la classe Boisson. C'est de l'héritage, ça, non ?

Anne : Exact. Je pense que le point vital, c'est que les décorateurs ont le même type que les objets qu'ils vont décorer. Ici, on utilise l'héritage pour obtenir la concordance de type, mais pas pour obtenir un comportement.

Marie : OK. Je vois bien que les décorateurs ont besoin de la même « interface » que les composants qu'ils enveloppent parce qu'ils doivent prendre la place du composant. Mais qu'en est-il du comportement ?

Anne : Quand on compose un décorateur avec un composant, on ajoute un nouveau comportement. On n'acquiert pas un nouveau comportement en l'héritant d'une superclasse mais en composant des objets ensemble.

Marie : Oui, on sous-classe la classe abstraite Boisson pour avoir le type correct, pas pour hériter de son comportement. Le comportement provient de la composition des décorateurs avec les composants de base ainsi que les autres décorateurs.

Anne : C'est cela.

Marie : Oooh, je vois. Et comme nous utilisons la composition, nous avons beaucoup plus de souplesse pour la façon de mélanger et d'assortir les ingrédients et les boissons. Génial.

Anne : Oui. Si on s'appuie sur l'héritage, le comportement ne peut être déterminé que statiquement au moment de la compilation. Autrement dit, nous n'avons pas d'autre comportement que celui que la superclasse nous donne ou que nous redéfinissons. Avec la composition, nous pouvons mélanger les ingrédients à notre guise... au moment de l'exécution.

Marie : Si j'ai bien compris, on peut implémenter de nouveaux décorateurs n'importe quand pour ajouter de nouveaux comportements. Si on s'appuyait sur l'héritage, il faudrait modifier le code existant chaque fois qu'on veut un nouveau comportement.

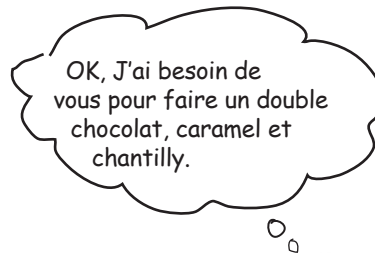
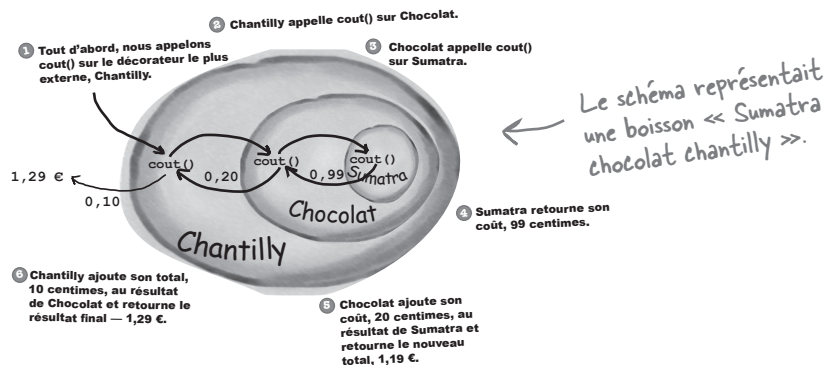
Anne : Exactement.

Marie : Encore une question. Si on n'a besoin de rien d'autre que d'hériter le type du composant, Pourquoi ne pas utiliser une interface au lieu d'une classe abstraite pour la classe Boisson ?

Anne : Souviens-toi. Quand on a reçu ce code, Starbuzz avait déjà une classe abstraite Boisson. Traditionnellement, le pattern Décorateur spécifie un composant abstrait, mais en Java, de toute évidence, on pourrait utiliser une interface. Mais on essaie toujours de ne pas modifier le code existant. Si la classe abstraite fonctionne bien, inutile d'y toucher.

Nouvelle formation des baristas

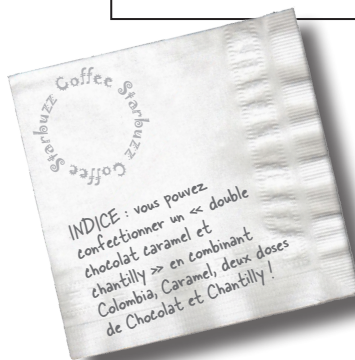
Représentez-vous ce qui se passe quand un client demande une boisson « double chocolat caramel et chantilly ». Reportez-vous à la carte pour les prix corrects, et tracez votre schéma dans le format que nous avons déjà utilisé (page 90) :



À vos crayons

Tracez votre schéma ici

Starbuzz Coffee	
Cafés	
Colombia	0,89
Sumatra	0,99
Deca	1,05
Espresso	1,99
Suppléments	
Lait	0,10
Chocolat	0,20
Caramel	0,15
Chantilly	0,10



Écrire le code de Starbuzz



Il est temps de traduire cette conception en code.

Commençons par la classe **Boisson**, la conception d'origine de Starbuzz qu'il est inutile de modifier. Jetons-y un coup d'œil :

```
public abstract class Boisson {
    String description = "Boisson inconnue";
    public String getDescription() {
        return description;
    }
    public abstract double cout();
}
```

Boisson est une classe abstraite qui possède deux méthodes : `getDescription()` et `cout()`.

`getDescription()` a déjà été implémentée pour nous, mais nous devons implémenter `cout()` dans les sous-classes.

Boisson est une classe très simple. Implémentons également la classe abstraite pour les ingrédients (décorateurs) :

```
public abstract class DecorateurIngredient extends Boisson {
    public abstract String getDescription();
}
```

D'abord, comme elle doit être interchangeable avec une `Boisson`, nous étendons la classe `Boisson`.

Nous allons aussi faire en sorte que les ingrédients (décorateurs) réimplémentent tous la méthode `getDescription()`. Nous allons aussi voir cela dans une seconde...

Coder les boissons

Maintenant que nous nous sommes débarrassés de nos classes de base, implémentons quelques boissons. Nous allons commencer par un Espresso. Souvenez-vous : il nous faut définir une description de la boisson spécifique et aussi implémenter la méthode `cout()`.

```
public class Espresso extends Boisson {
    public Espresso() {
        description = "Espresso";
    }
    public double cout() {
        return 1.99;
    }
}
```

Nous étendons d'abord la classe `Boisson`, puisqu'il s'agit d'une boisson.

Nous gérons la description dans le constructeur de la classe. Souvenez-vous que la variable d'instance `description` est héritée de `Boisson`.

Enfin, nous calculons le coût d'un Espresso. Maintenant que nous n'avons plus besoin d'ajouter les ingrédients dans cette classe, il suffit de retourner le prix d'un Espresso : 1,99 €.

```
public class Colombia extends Boisson {
    public Colombia() {
        description = "Pur Colombia";
    }

    public double cout() {
        return .89;
    }
}
```

Bien. Voici une autre Boisson. Il suffit de spécifier la description appropriée, « Pur Colombia », puis de retourner le prix correct : 89 centimes.

Starbuzz Coffee	
<u>Cafés</u>	
Colombia	0,89
Sumatra	0,99
Deca	1,05
Espresso	1,99
<u>Suppléments</u>	
Lait	0,10
Chocolat	0,20
Caramel	0,15
Chantilly	0,10

Vous pouvez créer les deux autres classes Boissons (Sumatra et Deca) exactement de la même façon.

Coder les ingrédients

Si vous regardez de nouveau le digramme de classe du pattern Décorateur, vous verrez que nous avons maintenant écrit notre composant abstrait (Boisson), notre décorateur abstrait (DecorateurIngredient) et que nous avons un composant concret (Colombia). Il est temps d'implémenter les décorateurs concrets. Voici Chocolat :

Chocolat est un décorateur : nous étendons DecorateurIngredient.

Souvenez-vous que DecorateurIngredient étend Boisson.

Nous allons instancier Chocolat avec une référence à une Boisson en utilisant :

```
public class Chocolat extends DecorateurIngredient {
    Boisson boisson;

    public Chocolat(Boisson boisson) {
        this.boisson = boisson;
    }

    public String getDescription() {
        return boisson.getDescription() + ", Chocolat";
    }

    public double cout() {
        return .20 + boisson.cout();
    }
}
```

(1) Une variable d'instance pour contenir la boisson que nous enveloppons.

(2) Un moyen pour affecter à cette variable d'instance l'objet que nous enveloppons. Ici, nous allons transmettre la boisson que nous enveloppons au constructeur du décorateur.

La description ne doit pas comprendre seulement la boisson – disons « Sumatra » – mais aussi chaque ingrédient qui décore la boisson, par exemple, « Sumatra, Chocolat ». Nous allons donc déléguer à l'objet que nous décorons pour l'obtenir, puis ajouter « Chocolat » à la fin de cette description.

Nous devons maintenant calculer le coût de notre boisson avec Chocolat. Nous déléguons d'abord l'appel à l'objet que nous décorons pour qu'il calcule son coût. Puis nous ajoutons le coût de Chocolat au résultat.

À la page suivante, nous allons instancier réellement la boisson et l'envelopper dans tous ses ingrédients (décorateurs), mais d'abord...

À vos crayons

Écrivez et compilez le code des autres ingrédients, Caramel et Chantilly. Vous en aurez besoin pour terminer et tester l'application.

Le café est servi

Félicitations. Il est temps de nous asseoir, de commander quelques cafés et de nous émerveiller de la souplesse de la conception que vous avez créée en appliquant le pattern Décorateur.

Voici un peu de code de test ^{*} pour commander nos boissons :

```
public class StarbuzzCoffee {
```

```
    public static void main(String args[]) {
        Boisson boisson = new Espresso();
        System.out.println(boisson.getDescription()
            + " €" + boisson.cout());
```

```
        Boisson boisson2 = new Sumatra();
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chantilly(boisson2);
```

```
        System.out.println(boisson2.getDescription()
            + " €" + boisson2.cout());
```

```
        Boisson boisson3 = new Colombia();
```

```
        boisson3 = new Caramel(boisson3);
```

```
        boisson3 = new Chocolat(boisson3);
```

```
        boisson3 = new Chantilly(boisson3);
```

```
        System.out.println(boisson3.getDescription()
            + " €" + boisson3.cout());
```

```
    }
```

```
}
```

Commander un espresso, pas d'ingrédients et afficher sa description et son coût.

Créer un objet Sumatra.

L'envelopper dans un Chocolat.

L'envelopper dans un second Chocolat.

L'envelopper de Chantilly.

Enfin nous servir un Colombia avec Caramel, Chocolat et Chantilly.

* Nous allons voir une bien meilleure façon de créer des objets décorés quand nous aborderons les patterns Fabrication et Monteur. Notez que Monteur est abordé dans l'annexe.

Maintenant, voyons le résultat :

```
Fichier Édition Fenêtre Aide NuageDeLait
% java StarbuzzCoffee
Espresso €1.99
Sumatra, Chocolat, Chocolat, Chantilly €1.49
Colombia, Caramel, Chocolat, Chantilly €1.34
%
```


il n'y a pas de Questions Stupides

Q: Il y a quelque chose qui m'inquiète un peu. On pourrait avoir du code qui teste un composant concret donné – disons, Colombia – et qui applique par exemple une remise. Une fois que j'ai enveloppé Colombia dans les décorateurs, cela ne fonctionnera plus.

R: Tout à fait. Si vous avez du code qui s'appuie sur le type du composant concret, les décorateurs endommagent ce code. Tant que vous n'utilisez que le type du composant abstrait, l'emploi des décorateurs reste transparent. Mais dès que vous utilisez le type des composants concrets, vous devez repenser la conception de votre application et votre usage des décorateurs.

Q: Est-ce qu'il ne pourrait pas arriver que le client d'une boisson finisse avec un décorateur qui ne serait pas le décorateur le plus externe ? Par exemple si j'avais un Sumatra avec Chocolat, Caramel et Chantilly, ce serait facile d'écrire du code qui se termine par une référence à Caramel au lieu de Chantilly, ce qui signifie qu'il n'inclurait pas Chantilly dans la commande.

R: On peut dire à coup sûr que le Pattern Décorateur oblige à gérer plus d'objets, et qu'il y a donc un risque accru que des erreurs de codage introduisent le type de problème auquel vous faites allusion. Mais on crée généralement des décorateurs en utilisant d'autres patterns, comme Fabrication et Monteur. Une fois que nous aurons abordé ces patterns, vous verrez que la création du composant concret avec son décorateur est « bien encapsulée » et évite ce genre de bogue.

Q: Les décorateurs peuvent-ils connaître les autres décorateurs de la chaîne ? Disons que j'aimerais que ma méthode getDescription() affiche « Chantilly, Double Chocolat » au lieu de « Chocolat, Chantilly, Chocolat ». Cela impliquerait que mon décorateur le plus externe connaisse tous les autres décorateurs qu'il enveloppe.

R: Les décorateurs sont conçus pour ajouter un comportement à l'objet qu'ils enveloppent. Si vous devez accéder à plusieurs couches de la chaîne de décorateurs, vous commencez à détourner le pattern de son véritable objectif. Néanmoins, ce n'est pas impossible. On peut imaginer un décorateur AffichageAmélioré qui analyse la description finale et affiche « Chocolat, Chantilly, Chocolat » sous la forme « Chantilly, Double Chocolat ». Notez que getDescription() pourrait retourner une ArrayList de descriptions pour faciliter le processus.

À vos crayons

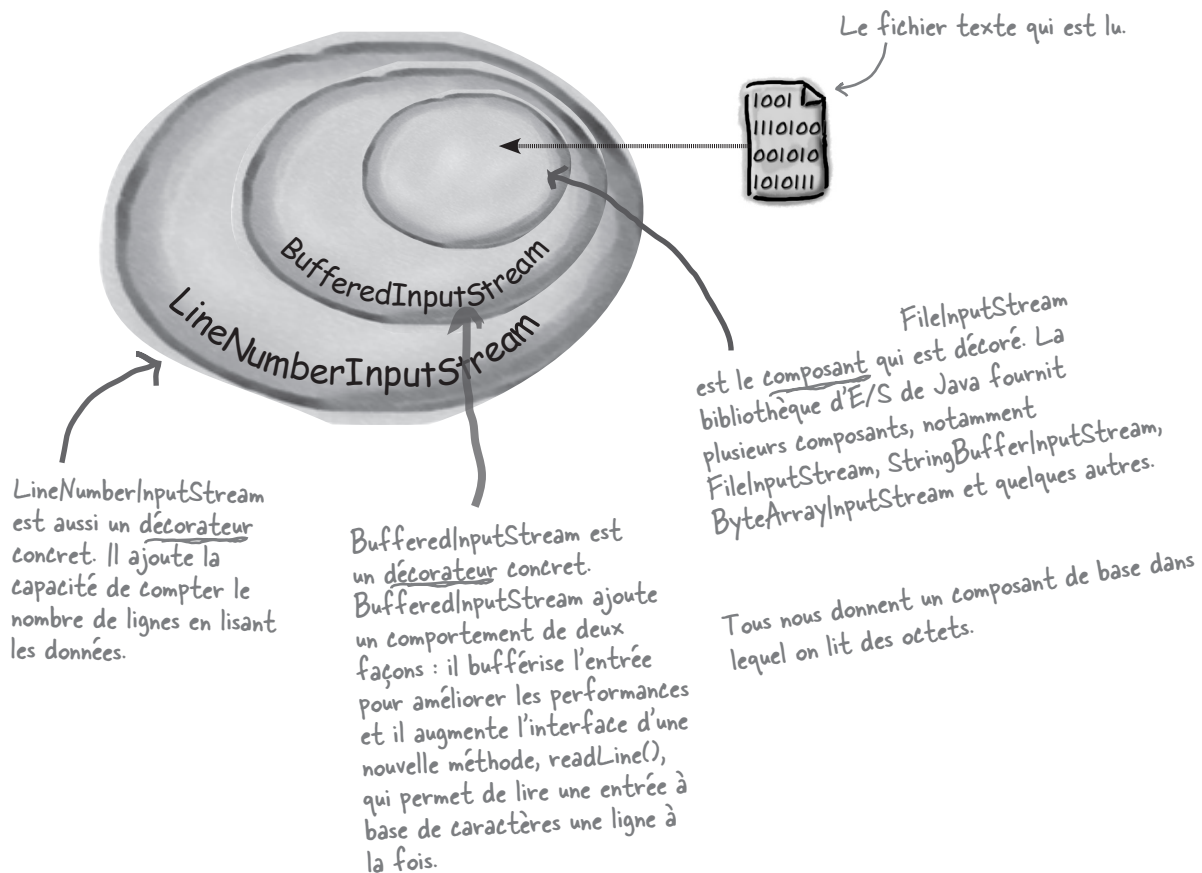


Nos amis de chez Starbuzz ont ajouté des tailles à leur carte. Vous pouvez maintenant commander un café en taille normale, grande et venti (traduction : petit, moyen et grand). Comme Starbuzz a vu cela comme une partie intrinsèque de la classe Café, ils ont ajouté deux méthodes à la classe Boisson : setTaille() et getTaille(). Ils voudraient également que le prix des ingrédients varie selon la taille. Par exemple, le Caramel coûterait respectivement 10, 15 et 20 centimes pour un petit café, un moyen et un grand.

Comment modifieriez-vous les classes décorateurs pour gérer cette demande de changement ?

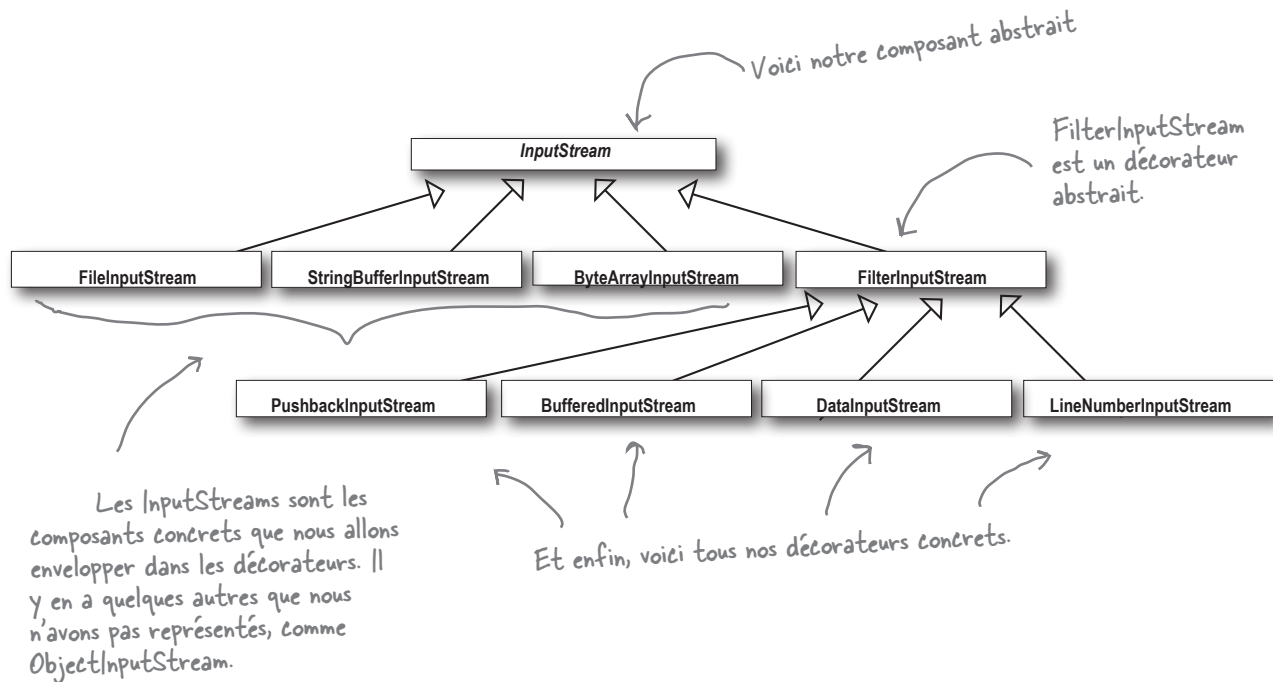
Décorateurs du monde réel : les E/S Java

Le nombre de classes du package java.io est... *étourdissant*. Si vous avez poussé un soupir la première fois (et même la deuxième et la troisième) que vous avez regardé cette API, rassurez-vous, vous n'êtes pas le seul. Mais maintenant que vous connaissez le pattern Décorateur, les classes d'E/S devraient être plus faciles à comprendre, puisque le package java.io est largement basé sur Décorateur. Voici un ensemble typique d'objets qui utilisent des décorateurs pour ajouter des fonctionnalités à la lecture de données dans un fichier :



`BufferedInputStream` et **`LineNumberInputStream`** dérivent tous deux de **`FilterInputStream`**, la classe qui joue le rôle de décorateur abstrait.

Décoration des classes de java.io



Vous constatez que ce n'est pas si différent de la conception de Starbuzz. Vous devez maintenant être à même d'étudier la documentation de l'API java.io et de composer des décorateurs avec les différents flots d'entrée.

Et vous verrez que les flots de sortie sont conçus de la même manière. Et vous avez probablement déjà découvert que les flots Reader/Writer (pour les données de type caractère) reflètent étroitement la conception des classes de flots, (avec quelques différences et quelques hiatus, mais de façon suffisamment proche pour comprendre ce qui se passe).

Mais les E/S Java mettent également en évidence les *inconvénients* du pattern Décorateur : l'application de celui-ci aboutit souvent à un grand nombre de petites classes épuisantes pour un développeur qui essaie d'utiliser une API basée sur Décorateur. Mais maintenant que vous savez comment ce pattern fonctionne, vous pouvez remettre les choses en perspective : quand vous utiliserez l'API de quelqu'un d'autre et qu'elle fera un usage intensif de Décorateur, vous serez capable de comprendre comment ses classes sont organisées et d'utiliser des « enveloppes » pour obtenir le comportement que vous recherchez.

Écrire votre propre décorateur d'E/S Java

Bien. Vous connaissez le pattern Décorateur et vous avez vu le diagramme de classes des E/S. Vous devez être prêt à écrire votre propre décorateur pour les entrées.

Que diriez-vous d'écrire un décorateur qui convertit toutes les majuscules en minuscules dans le flot d'entrée. Autrement dit, si nous lisons « Je connais le Pattern Décorateur, je suis le MAÎTRE DU MONDE ! » votre décorateur écrira « je connais le pattern décorateur, je suis le maître du monde ! ».

N'oubliez pas d'importer java.io... (omis ici)

D'abord étendre FilterInputStream, le décorateur abstrait de tous les InputStreams.

```
public class MinusculeInputStream extends FilterInputStream {
    public MinusculeInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int decal, int lg) throws IOException {
        int resultat = super.read(b, decal, lg);
        for (int i = decal; i < decal+resultat; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return resultat;
    }
}
```

RAPPEL : nous ne fournissons pas les instructions import et package dans les listings. Téléchargez le code source complet sur le site web du livre. Vous trouverez l'URL page xxxi de l'Intro.

Maintenant, nous devons implémenter deux méthodes read(). Elles acceptent un octet (ou un tableau d'octets) et convertissent chaque octet (qui représente un caractère) en minuscule si le caractère en question est une majuscule.



Tester votre nouveau décorateur

Écrivons rapidement un peu de code pour tester notre décorateur :

```
public class TestEntree {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in = new MinusculeInputStream(
                new BufferedInputStream(
                    new FileInputStream("test.txt")));
            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Créer le `FileInputStream` et le décorer, d'abord avec un `BufferedInputStream`, puis avec notre tout nouveau filtre, `MinusculeInputStream`.

Il suffit d'utiliser le flot pour lire les caractères jusqu'à la fin du fichier, puis d'afficher.

Je connais le Pattern décorateur, je suis le MAÎTRE DU MONDE !

fichier test.txt

Vous devez créer ce fichier.

Faisons le tourner :

```
Fichier Édition Fenêtre Aide LoiDuDécorateur
% java TestEntree
je connais le pattern décorateur, je suis le maître du monde !
%
```




Interview

Cette semaine :
les confessions d'un Décorateur

DPTLP : Bienvenue, Pattern Décorateur. Certains disent que vous n'avez pas trop le moral ces temps-ci ?

Décorateur : Oui, je sais que le monde me voit comme un design pattern prestigieux, mais, vous savez, j'ai mon lot de problèmes comme tout un chacun.

DPTLP : Peut-être pouvez-vous nous dire ce qui ne va pas ?

Décorateur : Bien sûr. Eh bien, vous savez que j'ai le pouvoir d'ajouter de la souplesse à une conception, il n'y a pas de doute là-dessus, mais j'ai aussi une face cachée. Voyez-vous, je peux parfois obliger à créer un tas de petites classes, et, à l'occasion, cela donne une conception que les autres peuvent avoir du mal à comprendre.

DPTLP : Pouvez-vous nous donner un exemple ?

Décorateur : Prenez les bibliothèques d'E/S de Java. Elles sont notoirement difficiles à comprendre d'emblée. Mais si les gens voyaient simplement les classes comme un ensemble d'enveloppes autour d'un `InputStream`, la vie serait beaucoup plus facile.

DPTLP : Ça n'a pas l'air si mal. Vous êtes toujours un grand pattern, et la solution n'est rien d'autre qu'un problème de formation, non ?

Décorateur : Ce n'est pas tout, j'en ai peur. J'ai des problèmes de typage : vous voyez, les gens prennent parfois un morceau de code client qui s'appuie sur des types spécifiques et introduisent des décorateurs sans avoir une réflexion d'ensemble. Maintenant, j'ai un gros avantage : vous pouvez généralement insérer des décorateurs de façon transparente et le client n'a jamais besoin de savoir qu'il a à faire avec un décorateur. Mais, comme je vous l'ai dit, il y a des programmes qui dépendent de types spécifiques, et quand vous commencez à introduire des décorateurs, patatrac ! Rien ne va plus.

DPTLP : Eh bien, je pense que tout le monde a compris qu'il faut être prudent quand on insère des décorateurs. Je ne vois là aucune raison de vous mettre martel en tête.

Décorateur : Je sais, j'essaie de me détacher. Mais j'ai un autre problème. L'introduction de décorateurs peut aussi accroître la complexité du code nécessaire pour instancier un composant. Une fois que vous avez des décorateurs, il ne suffit pas d'instancier le composant, il faut encore l'envelopper dans je ne sais combien de décorateurs.

DPTLP : Je vais interviewer les patterns Fabrication et Monteur la semaine prochaine. J'ai entendu dire qu'ils pouvaient aider à résoudre ce problème.

Décorateur : C'est vrai. Je devrais parler avec eux plus souvent.

DPTLP : Bon, nous savons tous que vous êtes un grand pattern qui permet de créer des conceptions souples et de rester fidèle au Principe Ouvert-Fermé, alors relevez la tête et essayez de penser de façon positive !

Décorateur : Je vais faire de mon mieux, merci.

Principes OO

- Encapsulez ce qui varie.
- Préférez la composition à l'héritage.
- Programmez des interfaces, non des implémentations.
- Efforcez-vous de coupler faiblement les objets qui interagissent.
- Les classes doivent être ouvertes à l'extension mais fermées à la modification.

Bases de l'OO

ction
ulation
rphisme
ge

Nous a
le Prin
Fermé
guider
effor
notre
que le
soient
nouvel

Patterns OO

Stratégie

d'algorithmes

rend

l'algorithme

clients

Décorateur - attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique à la dérivation pour étendre les fonctionnalités..

pour créer des

Propriété de Stéphane LARDIC, La Martinière Groupe <slardic@lamartiniere.fr>

Solutions des exercices

```

public class Boisson {
    // déclarer des variable d'instances pour coutLait,
    // coutCaramel, coutChocolat et coutChantilly, et
    // des méthodes get et set pour lait, caramel,
    // chocolat et chantilly.

    public float cout() {
        float coutIngredient = 0.0;
        if (aLait()) {
            coutIngredient += coutLait;
        }
        if (aCaramel()) {
            coutIngredient += coutCaramel;
        }
        if (aChocolat()) {
            coutIngredient += coutChocolat;
        }
        if (aChantilly()) {
            coutIngredient += coutChantilly;
        }
        return coutIngredient;
    }
}

public class Sumatra extends Boisson {

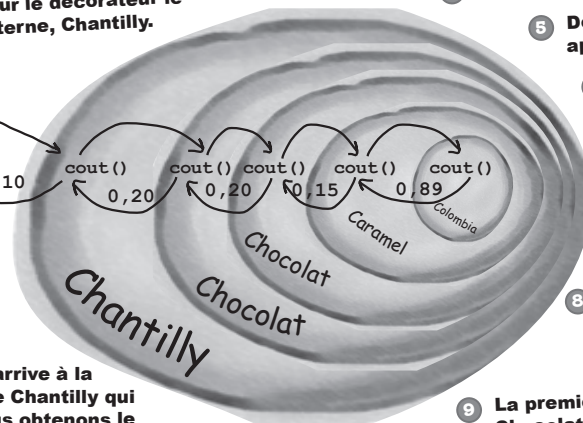
    public Sumatra() {
        description = "Excellent et corsé";
    }

    public float cout() {
        return 1.99 + super.cout();
    }
}

```

Nouvelle formation des baristas

« double chocolat caramel et chantilly »



- 1 Nous appelons d'abord `cout()` sur le décorateur le plus externe, Chantilly.
- 2 Chantilly appelle `cout()` sur Chocolat
- 3 Chocolat appelle `cout()` sur un autre Chocolat
- 4 Puis Chocolat appelle `cout()` sur Caramel.
- 5 Dernier supplément ! Caramel appelle `cout()` sur Colombia.
- 6 La méthode `cout()` de Colombia retourne 0,89 € et elle est dépillée.
- 7 La méthode `cout()` de Caramel ajoute 0,15. Elle retourne le résultat et elle est dépillée.
- 8 La deuxième méthode `cout()` de Chocolat ajoute 0,20. Elle retourne le résultat et elle est dépillée.
- 9 La première méthode `cout()` de Chocolat ajoute 0,20. Elle retourne le résultat et elle est dépillée.
- 10 Enfin, le résultat arrive à la méthode `cout()` de Chantilly qui ajoute 0,10 et nous obtenons le coût final : 1,54 €.

Solutions des exercices

Nos amis de chez Starbuzz ont ajouté des tailles à leur carte. Vous pouvez maintenant commander un café en taille normale, grande et venti (traduction : petit, moyen et grand). Comme Starbuzz a vu cela comme une partie intrinsèque de la classe Café, ils ont ajouté deux méthodes à la classe Boisson : `setTaille()` et `getTaille()`. Ils voudraient également que le prix des ingrédients varie selon la taille. Par exemple, le Caramel coûterait respectivement 10, 15 et 20 centimes pour un petit café, un moyen ou un grand.

Comment modifieriez-vous les classes décorateurs pour gérer cette demande de changement ?

```
public class Caramel extends DecorateurIngredient {
    Boisson boisson;

    public Caramel(Boisson boisson) {
        this.boisson = boisson;
    }

    public int getTaille() {
        return boisson.getTaille();
    }

    public String getDescription() {
        return boisson.getDescription() + ", Caramel";
    }

    public double cout() {
        double cout = boisson.cout();
        if (getTaille() == Boisson.NORMAL) {
            cout += .10;
        } else if (getTaille() == Boisson.GRANDE) {
            cout += .15;
        } else if (getTaille() == Boisson.VENTI) {
            cout += .20;
        }
        return cout;
    }
}
```

Nous devons maintenant propager la méthode `getTaille()` à la boisson enveloppée. Nous devons également transférer cette méthode dans la classe abstraite puisqu'elle sera utilisée dans tous les ingrédients décorateurs.

Ici nous obtenons la taille (qui se propage tout du long jusqu'à la boisson concrète) et nous ajoutons le coût approprié.

4 les patterns fabriques

Un peu de cuisine orientée objet



Apprêtez-vous à confectionner des conceptions OO faiblement couplées. Créer des objets ne se limite pas à utiliser l'opérateur **new**. Vous allez apprendre que l'instanciation est une activité qui ne devrait pas toujours être publique et qui peut souvent entraîner des *problèmes de couplage*. Et vous n'en avez pas vraiment envie, n'est-ce pas ? Découvrez comment les patterns fabriques peuvent vous aider à vous libérer de dépendances embarrassantes.

Bon, j'ai déjà lu trois chapitres et vous n'avez toujours pas répondu à ma question sur **new**. Nous ne sommes pas censés programmer une implémentation mais chaque fois que j'utilise **new**, c'est exactement ce que je fais, non ?



Quand vous voyez « new », pensez « concret ».

Oui, quand vous utilisez **new**, vous instanciez sans aucun doute possible une classe concrète : il s'agit donc bien d'une implémentation, non d'une interface. Et c'est une bonne question : vous avez appris que le fait de lier votre code à une classe concrète peut le rendre plus fragile et plus rigide.

```
Canard canard = new Colvert();
```

Nous voulons utiliser des interfaces pour conserver de la souplesse.

Mais nous devons créer une instance d'une classe concrète !

Quand vous avez tout un ensemble de classes concrètes apparentées, vous êtes souvent obligé d'écrire du code qui ressemble au fragment suivant :

```
Canard canard;  
  
if (dansLaMare) {  
    canard = new Colvert();  
} else if (aLaChasse) {  
    canard = new Leurre();  
} else if (dansLaBaignoire) {  
    canard = new CanardEnPlastique();  
}
```

Nous avons plusieurs classes canards différentes. Ce n'est qu'au moment de l'exécution que nous saurons laquelle nous avons besoin d'instancier.

Nous avons ici plusieurs instanciations de classes concrètes et la décision de la classe à instancier est prise au moment de l'exécution, en fonction d'un certain nombre de conditions.

Quand vous voyez un code comme celui-ci, vous savez que, lorsqu'il faudra apporter des modifications ou des extensions, vous devrez reprendre ce code et examiner ce qu'il faudra ajouter (ou supprimer). Ce type de code se retrouve souvent dans plusieurs parties de l'application, ce qui rend la maintenance et les mises à jour plus difficiles et plus sujettes à l'erreur.

Mais il faut bien créer des objets à un moment ou à un autre, et Java ne nous donne qu'un moyen de créer un objet, non ? Alors comment faire ?



Quel est le problème avec « new » ?

Du point de vue technique, il n'y a pas de problème avec **new**. Après tout, c'est une partie fondamentale de Java. Le vrai coupable, c'est notre vieil ami, le CHANGEMENT, et l'impact qu'il a sur notre utilisation de **new**.

En codant une interface, vous savez que vous vous isolez de la foule de changements qui pourraient survenir dans un système en aval. Pourquoi ? Si votre code s'appuie sur une interface, il fonctionnera avec toute nouvelle classe qui implémente cette interface via le polymorphisme. Mais si vous avez du code qui utilise de nombreuses classes concrètes, vous êtes en train de chercher les ennuis, parce que vous devrez modifier ce code à chaque ajout de nouvelles classes concrètes. Autrement dit, votre code ne sera pas « fermé à la modification ». Pour l'étendre avec de nouveaux types concrets, vous devrez le « rouvrir ».

Que pouvez-vous donc faire ? C'est dans des situations comme celles-ci qu'on peut se rabattre sur les principes de conception OO pour y chercher des indices. Souvenez-vous : notre premier principe concerne le changement et nous conseille d'*identifier les aspects qui varient et de les séparer de ceux qui demeurent les mêmes*.

N'oubliez pas qu'une conception doit être « ouverte à l'extension mais fermée à la modification » ; voir le chapitre 3 pour une petite révision.



MUSCLEZ VOS NEURONES

Comment pourriez-vous prendre toutes les parties de votre application qui instancient des classes concrètes et les encapsuler ou les séparer du reste de l'application ?

Identifier les aspects qui varient

Disons que vous possédez une boutique à Objectville et que vous vendez des pizzas. Pour rester à la pointe de la technologie, vous écririez peut-être un programme comme celui-ci :

```
Pizza commanderPizza() {
    Pizza pizza = new Pizza();

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
}
```



Si nous voulons de la souplesse, il nous faut une interface ou une classe abstraite, mais nous ne pouvons instancier directement ni l'une ni l'autre.

Mais un seul type de pizza ne suffit pas...

Vous ajoutez donc du code qui détermine le type de pizza approprié et qui passe ensuite à sa réalisation :

```
Pizza commanderPizza(String type) {
    Pizza pizza;

    if (type.equals("fromage")) {
        pizza = new PizzaFromage();
    } else if (type.equals("grecque")) {
        pizza = new PizzaGrecque();
    } else if (type.equals("poivrons")) {
        pizza = new PizzaPoivrons();
    }

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
}
```

Maintenant, nous transmettons le type de pizza à commanderPizza().

Selon le type de pizza, nous instancions la bonne classe concrète et nous l'affectons à chaque variable d'instance pizza. Notez que chaque pizza doit implémenter l'interface Pizza.

Une fois que nous avons une Pizza, nous la préparons (vous savez, étaler la pâte, mettre la sauce et ajouter garnitures et fromage). Puis nous la faisons cuire, nous la coupons et nous la mettons dans une boîte

Chaque sous-type de Pizza (PizzaFromage, PizzaPoivrons, etc.) sait comment se préparer lui-même !

Mais la pression du marché vous oblige à ajouter d'autres types de pizza

Vous vous rendez compte que vos concurrents ont ajouté à leur carte deux pizzas très tendance : une pizza aux fruits de mer et une pizza végétarienne. De toute évidence, vous devez suivre le mouvement et les proposer aussi. Et comme vous n'avez pas vendu beaucoup de pizzas grecques ces temps derniers, vous décidez de les retirer de la carte :

Ce code n'est PAS fermé à la modification. Si vous changez votre carte, vous devrez reprendre ce code et le modifier.

```
Pizza commanderPizza(String type) {
    Pizza pizza;

    if (type.equals("fromage")) {
        pizza = new PizzaFromage();
    } else if (type.equals("grecque")) {
        pizza = new PizzaGrecque();
    } else if (type.equals("poivrons")) {
        pizza = new PizzaPoivrons();
    } else if (type.equals("fruitsDeMer")) {
        pizza = new PizzaFruitsDeMer();
    } else if (type.equals("vegetarienne")) {
        pizza = new PizzaVegetarienne();
    }

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
}
```

Voici ce qui varie. Comme le type de pizza change avec le temps, vous n'allez pas cesser de modifier ce code.

Voici ce qui ne devrait pas changer. En majeure partie, la préparation, la cuisson et l'emballage d'une pizza n'ont pas varié depuis des lustres. Ce n'est donc pas ce code qui changera, mais seulement les pizzas sur lesquelles il opère.

De toute évidence, l'obligation de savoir *quelle* classe concrète est instanciée bousille votre méthode `commanderPizza()` et l'empêche d'être fermée à la modification. Mais maintenant que nous savons ce qui varie et ce qui ne varie pas, il est probablement temps de l'encapsuler.

Encapsuler la création des objets

Nous savons maintenant que nous ferions mieux d'extraire la création des objets de la méthode `commanderPizza()`. Mais comment ? Eh bien nous allons prendre ce code et le placer dans un autre objet qui ne s'occupera que d'une seule chose : créer des pizzas.

```
Pizza commanderPizza(String type) {
    Pizza pizza;

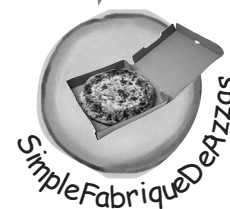
    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
}
```

Tout d'abord, nous extrayons le code qui crée les objets de la méthode `commanderPizza()`.

```
if (type.equals("fromage")) {
    pizza = new PizzaFromage();
} else if (type.equals("poivrons")) {
    pizza = new PizzaPoivrons();
} else if (type.equals("fruitsDeMer")) {
    pizza = new PizzaFruitsDeMer();
} else if (type.equals("vegetarienne")) {
    pizza = new PizzaVegetarienne();
}
```

Qu'allons-nous placer ici ?

Puis nous plaçons ce code dans un objet dont la seule responsabilité est de créer des pizzas. Si un autre objet a besoin qu'une pizza soit créée, c'est à cet objet qu'il faut s'adresser.



Nous avons un nom pour ce nouvel objet : nous l'appelons une Fabrique.

Les Fabriques gèrent les détails de la création des objets. Une fois que nous avons une `SimpleFabriqueDePizzas`, notre méthode `commanderPizza()` devient simplement un client de cet objet. Chaque fois qu'elle aura besoin d'une pizza, elle demandera à la fabrique de pizzas de lui en faire une. Le temps n'est plus où la méthode `commanderPizza()` devait savoir si la pizza était aux poivrons ou aux fruits de mer. Maintenant, une seule chose lui importe : obtenir une pizza qui implémente l'interface `Pizza` afin de pouvoir appeler `preparer()`, `cuire()`, `couper()` et `emballer()`.

Il nous reste encore quelques détails à régler, par exemple par quoi la méthode `commanderPizza()` remplace-t-elle le code qui crée les objets ? Pour le savoir, nous allons implémenter une simple fabrique de pizzas...

Construire une simple fabrique de pizzas

Nous allons commencer par la fabrique elle-même. Qu'allons-nous faire ? Nous allons définir une classe qui encapsule la création des objets pour toutes les pizzas. La voici...

Voici notre nouvelle classe, la SimpleFabriqueDePizzas. Elle n'a qu'une seule chose à faire dans la vie : créer des pizzas pour ses clients.

Nous définissons d'abord une méthode creerPizza() dans la fabrique. C'est la méthode que tous les clients utiliseront pour créer de nouvelles instances.

```
public class SimpleFabriqueDePizzas {
    public Pizza creerPizza(String type) {
        Pizza pizza = null;

        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        } else if (type.equals("poivrons")) {
            pizza = new PizzaPoivrons();
        } else if (type.equals("fruitsDeMer")) {
            pizza = new PizzaFruitsDeMer();
        } else if (type.equals("vegetarienne")) {
            pizza = new PizzaVegetarienne();
        }
        return pizza;
    }
}
```

Voici le code que nous avons extrait de la méthode commanderPizza().

Ce code est toujours paramétré par le type de pizza, tout comme l'était notre méthode commanderPizza() d'origine.

Il n'y a pas de questions stupides

Q: Quel est l'avantage de procéder ainsi ? On dirait qu'on transfère simplement le problème à un autre objet.

R: Il faut vous rappeler une chose : la SimpleFabriqueDePizzas peut avoir plusieurs clients. Nous n'avons vu que la méthode commanderPizza(), mais il pourrait y avoir une classe CartePizzas qui utiliserait la fabrique pour accéder à la description et au prix des pizzas. Nous pourrions également avoir une classe LivraisonADomicile qui gérerait les pizzas différemment de notre classe Pizzeria mais qui serait également un client de la fabrique.

Donc, en encapsulant la création des pizzas dans une seule classe, nous n'avons plus qu'un seul endroit auquel apporter des modifications quand l'implémentation change.

N'oubliez pas que nous sommes également sur le point de supprimer de notre code client les instanciations concrètes !

Q: J'ai vu une conception similaire dans laquelle une fabrique comme celle-ci est définie comme une méthode statique. Quelle est la différence ?

R: Définir une fabrique simple comme une méthode statique est une technique courante, et on la nomme souvent fabrique statique. Pourquoi employer une méthode statique ? Parce que vous n'avez pas besoin d'instancier un objet pour utiliser une méthode de création. Mais souvenez-vous que cette technique a un inconvénient : vous ne pouvez pas sous-classer ni modifier le comportement de la méthode de création.

Retravailler la classe Pizzeria

Il est maintenant temps d'arranger notre code client. Ce que nous voulons, c'est nous reposer sur la fabrique pour qu'elle crée les pizzas à notre place. Voici les modifications :

```

public class Pizzeria {
    SimpleFabriqueDePizzas fabrique;

    public Pizzeria(SimpleFabriqueDePizzas fabrique) {
        this.fabrique = fabrique;
    }

    public Pizza commanderPizza(String type) {
        Pizza pizza;

        pizza = fabrique.creerPizza(type);

        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }
    // autres méthodes
}

```

Nous donnons maintenant à Pizzeria une référence à une SimpleFabriqueDePizzas.

Nous transmettons la fabrique à Pizzeria dans le constructeur.

Et la méthode commanderPizza() utilise la fabrique pour créer ses pizzas en transmettant simplement le type de la commande.

Remarquez que nous avons remplacé l'opérateur new par une méthode de création de l'objet fabrique. Nous n'avons plus d'instanciations concrètes !



MUSCLEZ VOS NEURONES

Nous savons que la composition va nous permettre (entre autres) de modifier dynamiquement le comportement au moment de l'exécution parce que nous pouvons échanger les implémentations. Comment pourrions-nous appliquer cela à Pizzeria ? Quelles implémentations de fabrique pourrions-nous échanger ?

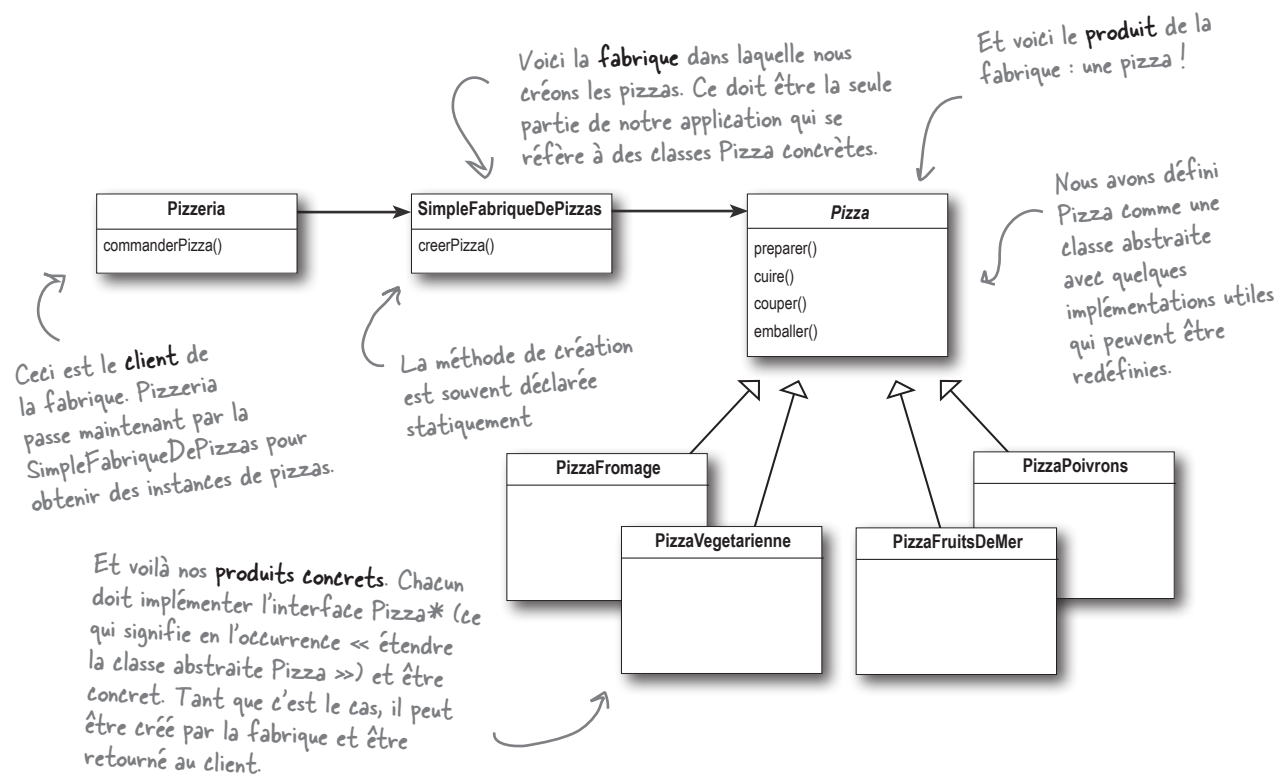
Nous ne savons pas à quoi vous pensez, mais nous pensons à des fabriques de pizzas de style breton, alsacien et provençal (sans oublier la Corse).

Fabrique Simple : définition

La Fabrique Simple n'est pas vraiment un Design Pattern : c'est plutôt un idiome de programmation.

Mais, comme son emploi est courant, nous lui avons décerné une mention honorable de Pattern Tête La Première. Certains informaticiens confondent cet idiome avec le « Pattern Fabrication ». Ainsi, la prochaine fois qu'il y aura un silence embarrassé entre vous et un autre développeur, vous aurez un sujet intéressant pour briser la glace.

Ce n'est pas parce que Fabrique Simple n'est pas un VRAI pattern que nous n'allons pas étudier comment il fonctionne. Jetons un coup d'œil au diagramme de classes de notre nouvelle Pizzeria :



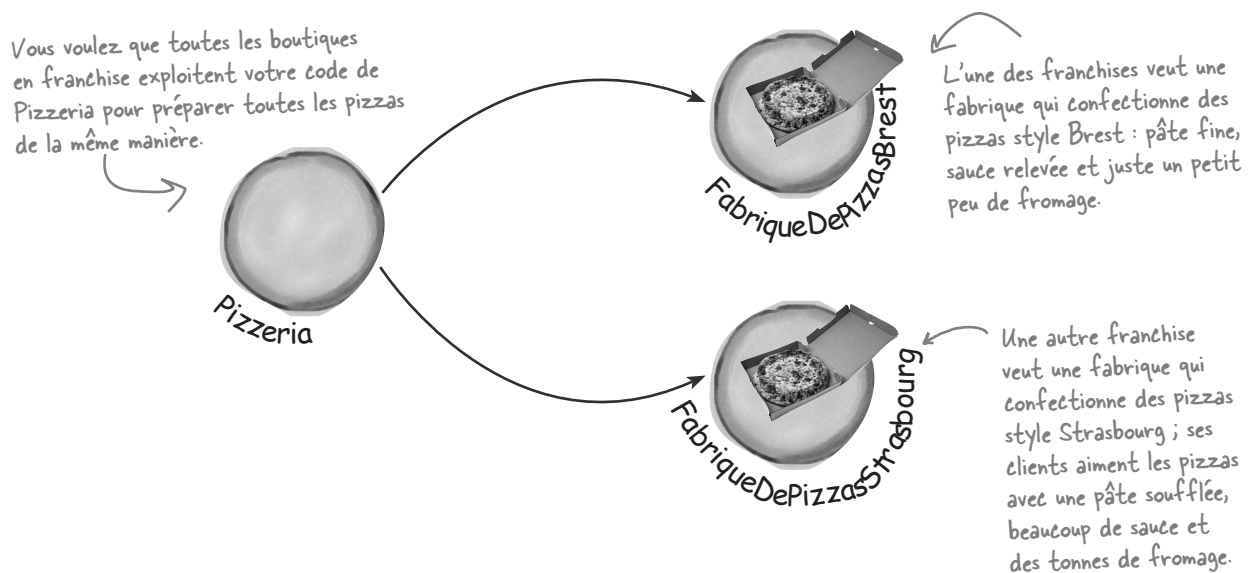
Considérez Fabrique Simple comme un échauffement. Nous allons bientôt explorer deux patterns très robustes qui sont tous deux des fabriques. Mais ne vous inquiétez pas, nous n'en avons pas fini avec la pizza !

*Juste un autre petit rappel : dans le contexte des design patterns, l'expression « implémenter une interface » ne veut PAS toujours dire « écrire une classe qui implémente une interface Java en utilisant le mot-clé « implements » dans sa déclaration ». Dans son acception générale, elle signifie qu'une classe concrète qui implémente une méthode d'un supertype (qui peut être une classe OU une interface) est toujours considérée comme « implémentant l'interface » de ce supertype.

Franchiser les pizzerias

Votre Pizzeria d'Objectville a connu une telle réussite que vous avez battu la concurrence à plates coutures. Maintenant, chacun veut une Pizzeria à proximité de chez lui. En tant que franchiseur, vous voulez vous assurer de la qualité de la production des franchises et vous voulez qu'ils utilisent votre code.

Mais qu'en est-il des différences régionales ? Chaque franchise peut souhaiter proposer différents styles de pizzas (Brest, Strasbourg, et Marseille par exemple), selon son emplacement géographique et les goûts des amateurs de pizza locaux.



Nous avons vu une approche...

Si nous extrayons SimpleFabriqueDePizzas et que nous créons trois fabriques différentes, FabriqueDePizzasBrest, FabriqueDePizzasStrasbourg et FabriqueDePizzasMarseille, il nous suffit de composer Pizzeria avec la fabrique appropriée et la franchise est prête à démarrer. C'est une approche possible.

Voyons à quoi elle ressemblerait...


```
FabriqueDePizzasBrest fabriqueBrest = new FabriqueDePizzasBrest();
Pizzeria boutiqueBrest = new Pizzeria(fabriqueBrest);
boutiqueBrest.commander("Végétarienne");
```

Ici, nous créons une fabrique pour faire des pizzas style Brest.
 Puis nous créons un Pizzeria et nous lui transmettons une référence à la fabrique Brest.
 ... et quand nous faisons des pizzas, nous obtenons des pizzas brestoises.

```
FabriqueDePizzasStrasbourg fabriqueStrasbourg = new FabriqueDePizzasStrasbourg();
Pizzeria boutiqueStrasbourg = new Pizzeria(fabriqueStrasbourg);
boutiqueStrasbourg.commander("Végétarienne");
```

Pareil pour Strasbourg : nous créons une fabrique de pizzas style Strasbourg et nous créons une Pizzeria composée avec une fabrique Strasbourg. Quand nous créons des pizzas, elles ont le goût de celles de Strasbourg.

Mais le contrôle qualité vous semble légèrement insuffisant...

Vous avez donc testé le concept de FabriqueSimple, et vous avez découvert que les franchises utilisaient bien votre fabrique pour créer des pizzas, mais qu'elles commençaient à employer leur propres procédures maison pour le reste du processus : elles avaient un mode de cuisson un peu différent, oubliaient de couper la pizza et achetaient leurs boîtes à d'autres fournisseurs.

Après avoir quelque peu repensé le problème, vous constatez que ce que vous aimeriez réellement faire serait de créer une structure qui lie la boutique et la création des pizzas, tout en permettant de conserver de la souplesse.

Dans notre premier programme, avant la SimpleFabriqueDePizzas, le code de confection des pizzas était bien lié à Pizzeria, mais la souplesse manquait à l'appel. Alors, comment faire pour avoir notre pizza et la manger ?

Comme je fais des pizzas depuis des années, j'ai pensé que je pouvais ajouter mes propres « améliorations » aux procédures de Pizzeria...



Ce n'est pas ce que vous attendez d'un bon franchisé. Vous ne voulez PAS savoir ce qu'il met sur ses pizzas.

Une structure pour la Pizzeria

Il *existe* un moyen de localiser toutes les activités de fabrication des pizzas dans la classe Pizzeria tout en laissant aux franchises la liberté d'avoir leur propre style régional.

Voici ce que nous allons faire : nous allons remettre la méthode `creerPizza()` dans Pizzeria. Mais ce sera cette fois une **méthode abstraite**, puis nous créerons une sous-classe de Pizzeria pour chaque style régional.

Examinons d'abord les modifications de Pizzeria :

Pizzeria est maintenant abstraite (vous allez voir pourquoi ci-dessous).

```
public abstract class Pizzeria {  
  
    public Pizza commanderPizza(String type) {  
        Pizza pizza;  
  
        pizza = creerPizza(type);  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
  
        return pizza;  
    }  
  
    abstract creerPizza(String type);  
}
```

Maintenant, `creerPizza()` est de nouveau un appel à une méthode de Pizzeria et non à un objet fabrique.

Ceci ne change pas...

Et nous avons transféré notre objet fabrique à cette méthode.

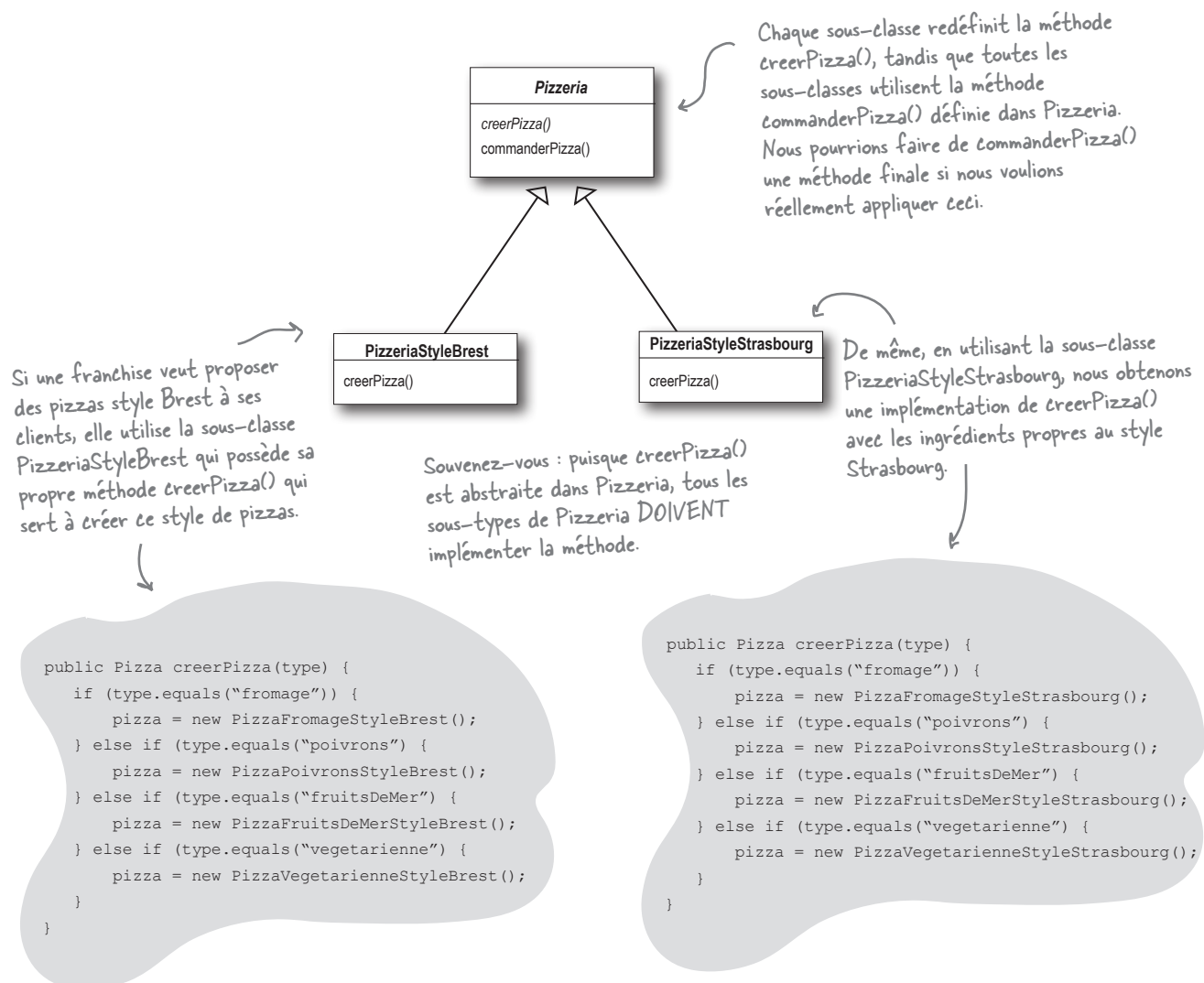
Notre « méthode de fabrication » est maintenant abstraite dans Pizzeria.

Maintenant, notre boutique attend des sous-classes. Nous allons donc avoir une sous-classe pour chaque type régional (PizzeriaBrest, PizzeriaStrasbourg, PizzeriaMarseille) et chaque sous-classe choisira ce qui constitue une pizza. Voyons un peu comment cela va fonctionner.

Laisser les sous-classes décider

Souvenez-vous : Pizzeria a déjà un système de commandes bien au point dans la méthode `commanderPizza()` et vous voulez être certain qu'elle est cohérente entre toutes les franchises.

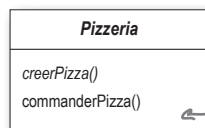
Ce qui varie entre les pizzerias régionales, c'est le style de pizzas qu'elles fabriquent – la pizza de Brest a une pâte fine, celle de Strasbourg est épaisse, etc. – et nous allons remplacer toutes ces variations dans la méthode `creerPizza()` et la rendre responsable de la création du bon type de pizza. Pour ce faire, nous laissons chaque sous-classe de `Pizzeria` définir à quoi va ressembler la méthode `creerPizza()`. Ainsi, nous aurons un certain nombre de sous-classes concrètes de `Pizzeria`, chacune ayant ses propres variantes, mais toutes s'insérant dans la structure de `Pizzeria` et continuant à utiliser notre méthode bien au point : `commanderPizza()`.



Je ne comprends pas. Les sous-classes de Pizzeria ne sont que dessous-classes. Comment peuvent-elles décider quoi que ce soit ? Je ne vois aucune logique de prise de décision dans le code de PizzeriaStyleBrest....

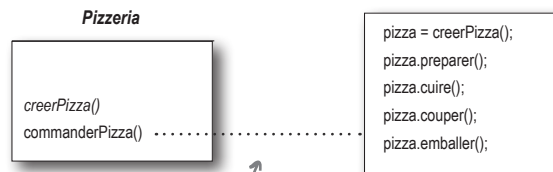


Bon. Réfléchissez du point de vue de la méthode `commanderPizza()` : elle est définie dans la classe abstraite `Pizzeria`, mais les types concrets ne sont créés que dans les sous-classes.



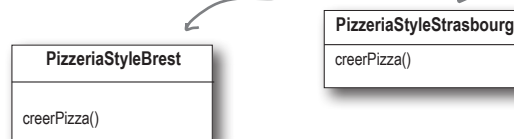
`commanderPizza()` est définie dans la classe abstraite `Pizzeria`, pas dans les sous-classes. La méthode `Pizzeria` n'a donc aucune idée de la sous-classe qui exécute réellement le code et qui fabrique les pizzas.

Maintenant, pour aller un peu plus loin, la méthode `commanderPizza()` fait un tas de choses avec un objet `Pizza` (elle le prépare, le fait cuire, le coupe et le met dans une boîte), mais comme `Pizza` est abstraite, `commanderPizza()` n'a aucune idée des classes concrètes qui sont impliquées. Autrement dit, elle est découplée !



`commanderPizza()` appelle `creerPizza()` pour obtenir un objet `Pizza`. Mais quel genre de pizza va-t-elle recevoir ? La méthode `commanderPizza()` ne peut pas décider : elle ne sait pas comment faire. Alors, qui décide ?

Lorsque `commanderPizza()` appelle `creerPizza()`, l'une de vos sous-classes entre en scène pour créer une pizza. Quelle sorte de pizza va-t-elle faire ? C'est le choix de la boutique à partir de laquelle vous passez commande, `PizzeriaStyleBrest` ou `PizzeriaStyleStrasbourg`, qui va le déterminer..



Est-ce donc une décision en temps réel que les sous-classes prennent ? Non, mais du point de vue de `commanderPizza()`, si vous avez choisi un `PizzeriaStyleBrest`, cette sous-classe va déterminer quelle pizza sera créée. Ce ne sont donc pas les sous-classes qui « décident » réellement – c'est vous qui avez décidé en choisissant votre boutique. En revanchent, elles déterminent bien la sorte de pizza qui sera fabriquée.

Créer une Pizzeria

Être une franchise a ses avantages. Vous profitez de toutes les fonctionnalités de Pizzeria pour rien. Il suffit aux boutiques régionales de sous-classer Pizzeria et de fournir une méthode `creerPizza()` qui implémente leur style de Pizza. Nous allons nous occuper des trois grand styles de pizzas pour les franchisés.

Voici le style Brest :

creerPizza() retourne une Pizza, et la sous-classe a l'entière responsabilité de déterminer la Pizza concrète qu'elle instancie

Comme PizzeriaBrest étend Pizzeria, elle hérite (entre autres) de la méthode commanderPizza().

```
public class PizzeriaBrest extends Pizzeria {
    Pizza creerPizza(String item) {
        if (choix.equals("fromage")) {
            return new PizzaFromageStyleBrest();
        } else if (choix.equals("vegetarienne")) {
            return new PizzaVegetarienneStyleBrest();
        } else if (choix.equals("fruitsDeMer")) {
            return new PizzaFruitsDeMerStyleBrest();
        } else if (choix.equals("poivrons")) {
            return new PizzaPoivronsStyleBrest();
        } else return null;
    }
}
```

Nous devons implémenter `creerPizza()` puisqu'elle est abstraite dans Pizzeria.

Voici l'endroit où nous créons nos classes concrètes. Pour chaque type de Pizza nous créons le style Brest.

** Notez que la méthode `commanderPizza()` de la superclasse ne dispose d'aucun indice sur la Pizza que nous créons : elle sait simplement qu'elle peut la préparer, la faire cuire la découper et l'emballer !*

Une fois nos sous-classes de Pizzeria construites, il va être temps de songer à commander une pizza ou deux. Mais auparavant, pourquoi ne pas essayer de construire les boutiques Style Strasbourg et Style Marseille page suivante ?

À vos crayons

Nous en avons fini avec la classe PizzeriaBrest. Encore deux et nous serons prêts à accorder des franchises !

Écrivez ici les implémentations de PizzeriaStrasbourg et de PizzeriaMarseille :

Déclarer une méthode de fabrique

Rien qu'en apportant une ou deux transformations à Pizzeria, nous sommes passés d'un objet gérant l'instanciation de nos classes concrètes à un ensemble de sous-classes qui assument maintenant cette responsabilité. Voyons cela de plus près :

```
public abstract class Pizzeria {
```

```
    public Pizza commanderPizza(String type) {
        Pizza pizza;
```

```
        pizza = creerPizza(type);
```

```
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
```

```
        return pizza;
```

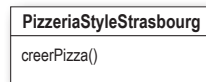
```
    }
```

```
    protected abstract Pizza creerPizza(String type);
```

```
    // autres méthodes
```

```
}
```

Les sous-classes de Pizzeria gèrent l'instanciation des objets à notre place dans la méthode `creerPizza()`.



Toute la responsabilité de l'instanciation des Pizzas a été transférée à une méthode qui se comporte comme une fabrique.



Code à la loupe

Une méthode de fabrique gère la création des objets et l'encapsule dans une sous-classe. Cette technique découple le code client de la superclasse du code de création des objets de la sous-classe.

abstract Produit fabrication(String type)

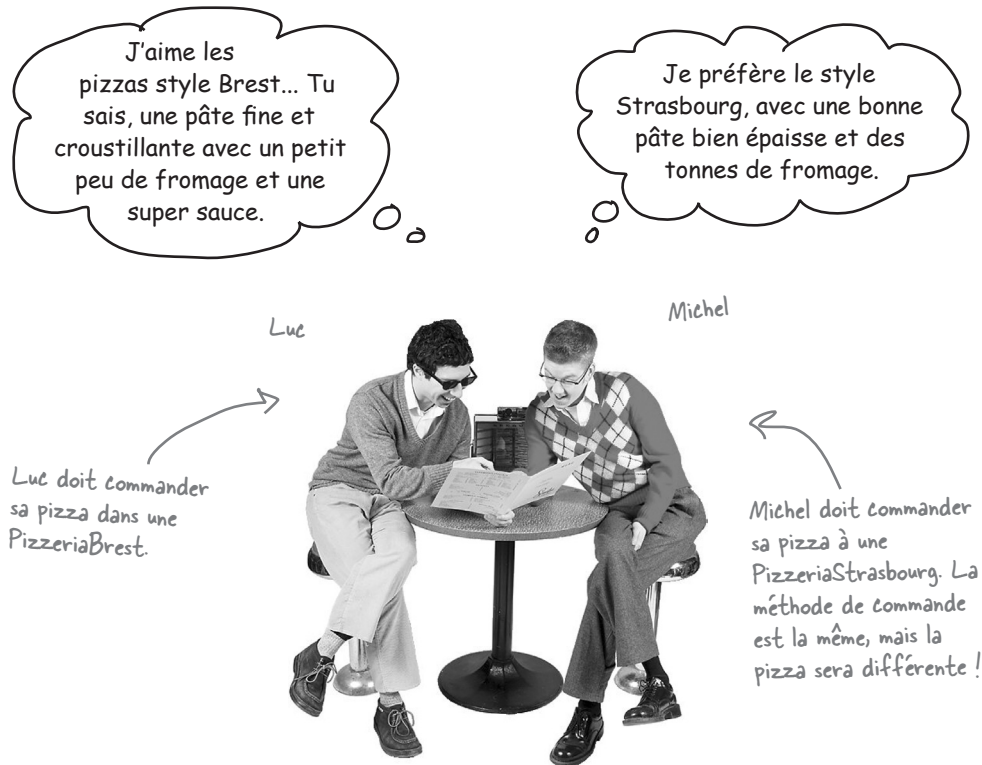
Comme une méthode de fabrication est abstraite, on compte sur les sous-classes pour gérer la création des objets.

Une méthode de fabrication retourne un `Produit` qu'on utilise généralement dans les méthodes définies dans la super-classe.

Une méthode de fabrique peut être paramétrée (ou non) pour choisir entre différentes variantes d'un produit.

Une méthode de fabrique isole le client (le code de la superclasse, tel `commanderPizza()`) : elle lui évite de devoir connaître la sorte de `Produit` concret qui est réellement créée.

Voyons comment tout ceci fonctionne : commandons des pizzas grâce à la méthode de fabrique de pizzas



Mais comment commandent-ils ?

- 1 Tout d'abord, Michel et Luc ont besoin d'une instance de Pizzeria. Michel doit instancier une PizzeriaStrasbourg et Luc une PizzeriaBrest.
- 2 Avec une Pizzeria à leur disposition, Luc et Michel appellent tous deux la méthode commanderPizza() et lui transmettent le type de pizza dont ils ont envie (fromage, poivrons, etc.).
- 3 Pour créer les pizzas, la méthode creerPizza() est appelée. Elle est définie dans les deux sous-classes PizzeriaBrest et PizzeriaStrasbourg. Telles que nous les avons définies, PizzeriaBrest instancie une pizza style Brest et PizzeriaStrasbourg une pizza style Strasbourg. Dans l'un et l'autre cas, la Pizza est retournée à la méthode commanderPizza().
- 4 La méthode commanderPizza() n'a aucune idée du genre de pizza qui a été créé, mais elle sait que c'est une pizza : elle la prépare, la fait cuire, la découpe et l'emballage pour Luc et Michel.

Voyons comment les pizzas sont réellement faites sur commande...

Dans les coulisses



1

Suivons la commande de Luc : tout d'abord, il nous faut une PizzeriaBrest :

```
Pizzeria pizzeriaBrest = new PizzeriaBrest();
```

Crée une instance de PizzeriaBrest.

2

Maintenant que nous avons une boutique, nous pouvons prendre une commande :

```
pizzeriaBrest.commanderPizza("fromage");
```

La méthode commanderPizza() est appelée sur l'objet pizzeriaBrest (la méthode définie dans Pizzeria s'exécute).

3

Puis la méthode commanderPizza() appelle la méthode creerPizza():

```
Pizza pizza = creerPizza("fromage");
```

Souvenez-vous que creerPizza(), la méthode de fabrication, est implémentée dans la sous-classe. En l'occurrence, elle retourne une PizzaFromageBrest.

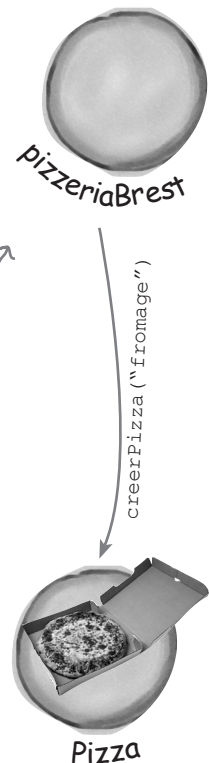
4

Enfin, nous avons une pizza brute et la méthode commanderPizza() finit de la préparer :

```
pizza.preparer();  
pizza.cuire();  
pizza.couper();  
pizza.emballer();
```

La méthode commanderPizza() obtient une Pizza, sans savoir exactement de quelle classe concrète il s'agit.

Toutes ces méthodes sont définies dans la pizza spécifique retournée depuis la méthode de fabrique creerPizza(), définie dans PizzeriaBrest.



Il ne nous manque qu'une chose : la PIZZA !

Notre Pizzeria ne va pas faire un tabac si on n'y trouve pas de pizzas. Nous allons donc en implémenter :



Nous allons commencer par une classe abstraite `Pizza` dont toutes les pizzas concrètes dériveront.

```
public abstract class Pizza {
    String nom;
    String pate;
    String sauce;
    ArrayList garnitures = new ArrayList();

    void preparer() {
        System.out.println("Préparation de " + nom);
        System.out.println("Étalage de la pâte...");
        System.out.println("Ajout de la sauce...");
        System.out.println("Ajout des garnitures: ");
        for (int i = 0; i < garnitures.size(); i++) {
            System.out.println(" " + garnitures.get(i));
        }
    }

    void cuire() {
        System.out.println("Cuisson 25 minutes à 180°");
    }

    void couper() {
        System.out.println("Découpage en parts triangulaires");
    }

    void emballer() {
        System.out.println("Emballage dans une boîte officielle");
    }

    public String getNom() {
        return nom;
    }
}
```

Chaque `Pizza` a un nom, un type de pâte, un type de sauce et un ensemble d'ingrédients.

La classe abstraite fournit des comportements par défaut pour la cuisson, le découpage et l'emballage.

La préparation suit un certain nombre d'étapes ordonnées selon une suite particulière.

RAPPEL : les listings ne contiennent pas d'instructions `import` et `package`. Vous pouvez télécharger le code source complet sur le site du livre, dont l'URL se trouve page xxxi de l'Intro.

Maintenant, il ne nous faut plus que quelques classes concrètes...

Que diriez-vous de définir des pizzas au fromage style Brest et Strasbourg ?

```
public class PizzaFromageStyleBrest extends Pizza {
    public PizzaFromageStyleBrest() {
        nom = "Pizza sauce style brest et fromage";
        pate = "Pâte fine"; sauce = "Sauce Marinara";
        garnitures.add("Parmigiano reggiano râpé");
    }
}
```

La Pizza brestoise a sa propre sauce marinara et une pâte fine.

Et une garniture, du parmigiano reggiano !

```
public class PizzaFromageStyleStrasbourg extends Pizza {

    public PizzaFromageStyleStrasbourg() {
        nom = "Pizza pâte style Strasbourg et fromage";
        pate = "Extra épaisse";
        sauce = "Sauce aux tomates cerise";
        garnitures.add("Lamelles de mozzarella");
    }

    void couper() {
        System.out.println("Découpage en parts carrées");
    }
}
```

La Pizza Strasbourg a une sauce aux tomates cerise et une pâte très épaisse.

La Pizza Strasbourg a des tonnes de mozzarella !

La pizza Strasbourg redéfinit également la méthode couper() afin de découper des parts carrées.

Nous avons assez attendu, il est temps pour quelques pizzas !

```
public class PizzaTestDrive {
```

```
    public static void main(String[] args) {
```

```
        Pizzeria boutiqueBrest = new PizzeriaBrest();
```

```
        Pizzeria boutiqueStrasbourg = new PizzeriaStrasbourg();
```

```
        Pizza pizza = boutiqueBrest.commanderPizza("fromage");
```

```
        System.out.println("Luc a commandé une " + pizza.getNom() + "\n");
```

```
        pizza = boutiqueStrasbourg.commanderPizza("fromage");
```

```
        System.out.println("Michel a commandé une " + pizza.getNom() + "\n");
```

```
    }
```

```
}
```

D'abord, nous créons deux boutiques différentes.

Puis nous en utilisons une pour la commande de Luc.

Et l'autre pour celle de Michel.

Fichier Édition Fenêtre Aide PlusDeFromage

```
% java PizzaTestDrive
```

```
Préparation de Pizza sauce style brest au fromage
```

```
Étalage de la pâte...
```

```
Ajout de la sauce...
```

```
Ajout des garnitures :
```

```
    Parmigiano reggiano râpé
```

```
Cuisson 25 minutes à 180°
```

```
Découpage en parts triangulaires
```

```
Emballage dans une boîte officielle
```

```
Luc a commandé une Pizza sauce style brest et fromage
```

```
Préparation de Pizza pâte Strasbourg et fromage
```

```
Étalage de la pâte...
```

```
Ajout de la sauce...
```

```
Ajout des garnitures :
```

```
    Mozzarella en lamelles
```

```
Cuisson 25 minutes à 180°
```

```
Découpage en parts carrées
```

```
Emballage dans une boîte officielle
```

```
Michel a commandé une Pizza pâte Strasbourg et fromage
```

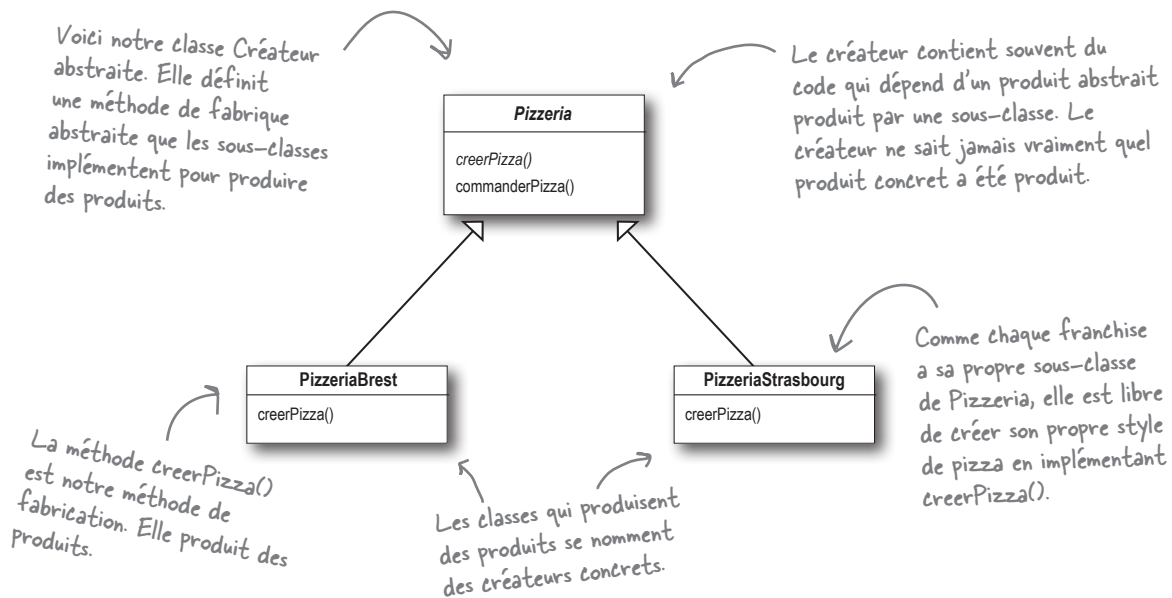
La préparation commencée, les garnitures sont ajoutées, et les deux pizzas sont cuites, découpées et emballées.

Notre superclasse n'a jamais eu besoin de connaître les détails : la sous-classe s'est occupée de tout rien qu'en instanciant la bonne pizza.

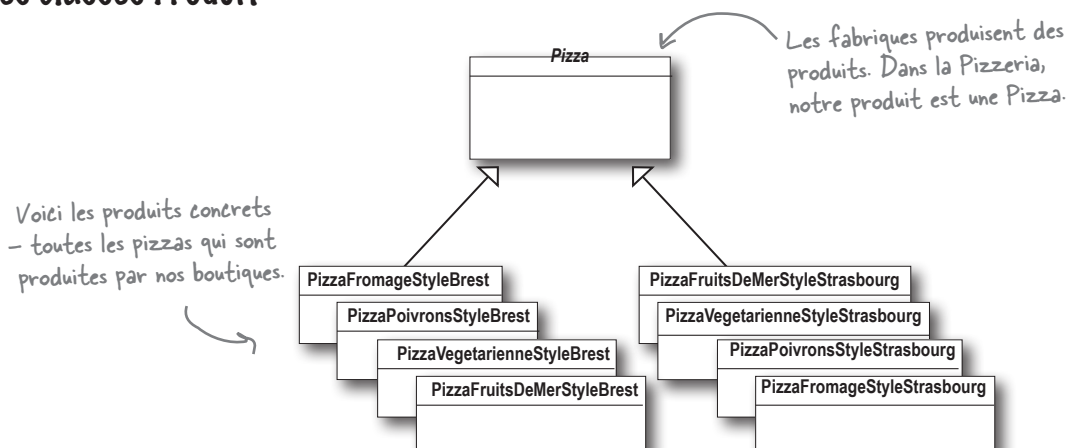
Il est enfin temps de rencontrer le pattern Fabrication

Tous les patterns de fabrique encapsulent la création des objets. Le Design Pattern Fabrication (Factory Method) encapsule la création des objets en laissant aux sous-classes le choix des objets à créer. Observons les diagrammes de classes pour voir qui sont les acteurs de ce pattern :

Les classes Créateur (ou Facteur)



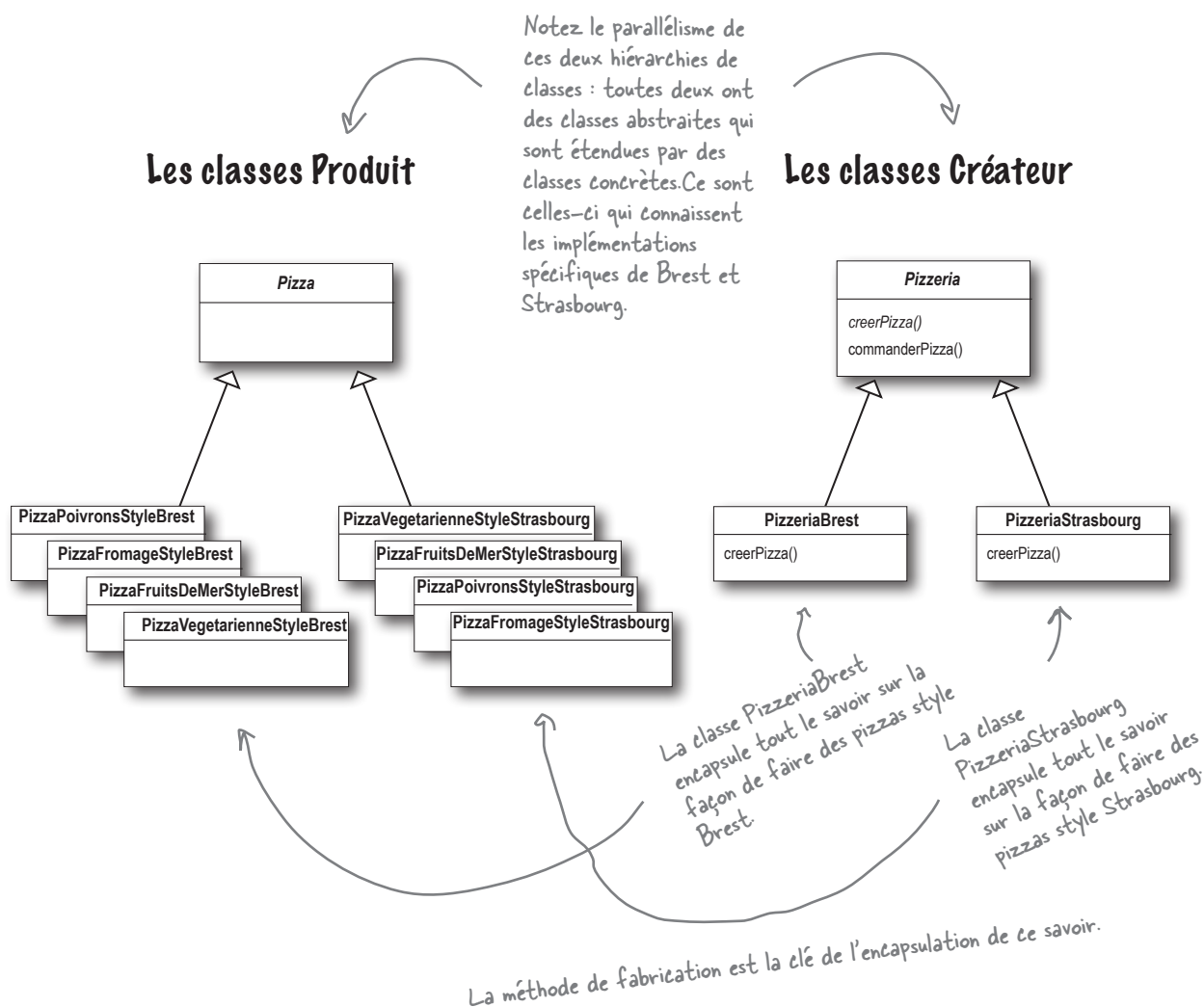
Les classes Produit



Un autre point de vue : des hiérarchies de classes parallèles

Nous avons vu que nous obtenions une structure en combinant la méthode `commanderPizza()` avec une méthode de fabrique. Une autre façon de se représenter la structure fournie pas ce pattern consiste à envisager la façon dont il encapsule la connaissance du produit dans chaque créateur.

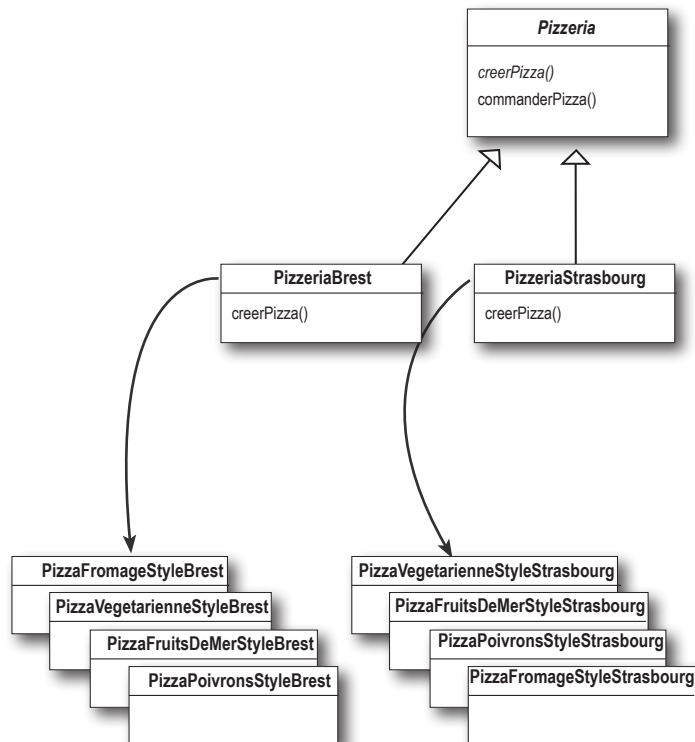
Voyons les deux hiérarchies de classes parallèles et les relations qu'elles entretiennent :





Problème de conception

Il nous faut un autre type de pizza pour ces fous de Marseillais (fous dans le *bon* sens du terme bien sûr). Représentez l'autre ensemble de classes parallèles dont vous avez besoin pour ajouter une région Marseille à notre Pizzeria.



Terminez le diagramme ici...

Maintenant, inscrivez les cinq garnitures les *plus bizarres* que vous pouvez imaginer mettre sur une pizza.

Ensuite, vous serez prêt à faire des affaires avec vos pizzas à Marseille !

Le Pattern Fabrication : définition

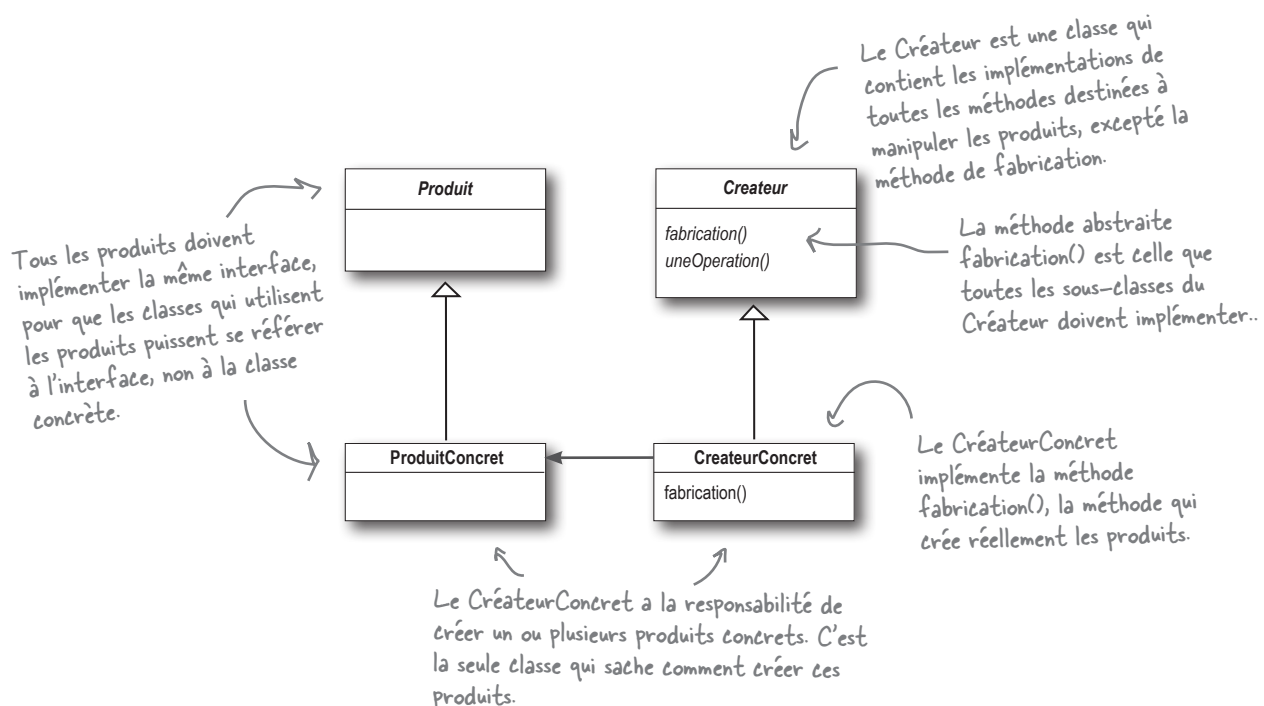
Il est temps de révéler la définition officielle du Pattern Fabrication (*Factory Method*) :

Le pattern Fabrication définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Fabrication permet à une classe de déléguer l'instanciation à des sous-classes.

Comme toute fabrication, le pattern Fabrication nous offre un moyen d'encapsuler l'instanciation de types concrets. Si vous observez le diagramme de classes ci-dessous, vous constaterez que le Créateur abstrait vous fournit une interface dotée d'une méthode permettant de créer des objets, alias « fabrication ». Toutes les autres méthodes implémentées dans le Créateur abstrait sont écrites pour opérer sur les produits créés par la fabrique. Seules les sous-classes implémentent réellement la méthode de fabrique et créent des produits.

Comme dans la définition officielle, vous entendrez souvent des développeurs dire que la Fabrication laisse aux sous-classes le choix des classes à instancier. Ils parlent de « choix » non seulement parce que le pattern permet aux sous-classes elles-mêmes de décider au moment de l'exécution, mais aussi parce que la classe créatrice ne sait rien des produits réels qui seront créés, choix qui appartient entièrement à la sous-classe utilisée.

Vous pourriez leur demander ce que « choix » signifie, mais nous parions que vous connaissez maintenant ce pattern mieux qu'eux !



il n'y a pas de questions stupides

Q: Quel est l'avantage du Pattern Fabrication quand on n'a qu'un CréateurConcret ?

R: Le Pattern Fabrication est utile même si vous n'avez qu'un CréateurConcret parce que vous découpez l'implémentation du produit de son utilisation. Si vous ajoutez d'autres produits ou si vous modifiez l'implémentation d'un produit existant, le Créateur ne sera pas affecté (parce que le Créateur n'est fortement couplé à aucun ProduitConcret).

Q: Serait-il correct de dire que nos boutiques de Brest et de Strasbourg sont implémentées avec une Fabrique Simple ? Cela en a tout l'air.

R: Les deux patterns sont similaires mais utilisés de façon différente. Même si l'implémentation de chaque boutique concrète ressemble beaucoup à une SimpleFabricationDePizzas, n'oubliez pas que les boutiques concrètes étendent une classe dans laquelle la méthode creerPizza() est déclarée abstraite. Il appartient à chaque boutique de définir le comportement de creerPizza(). Dans Fabrication Simple, la fabrication est un autre objet qui est combiné avec la Pizzeria.

Q: Est-ce que la méthode fabrication() et le Créateur sont toujours abstraits ?

R: Non. Vous pouvez définir une méthode de fabrication par défaut afin de créer un produit concret. Ainsi, vous disposez toujours d'un moyen de créer des produits, même s'il n'y a pas de sous-classes du Créateur.

Q: Chaque boutique peut confectionner quatre sortes de pizzas différentes selon le type qui lui est transmis. Tous les créateurs concrets créent-ils plusieurs produits ou arrive-t-il qu'ils n'en fassent qu'un ?

R: Nous avons implémenté ce qu'on appelle une fabrication paramétrée. Comme vous l'avez remarqué, elle peut créer plusieurs objets en fonction de l'argument qu'on lui transmet. Toutefois, il est fréquent qu'une fabrication ne produise qu'un seul objet et qu'elle ne soit pas paramétrée.

Q: Vos types paramétrés ne semblent pas très sûrs. Je ne transmets jamais qu'une chaîne de caractères, après tout ! Que se passerait-il si je demandais une « pizza aux poireaux » ?

R: Sans aucun doute. Cela provoquerait ce qu'on appelle dans le métier une « erreur d'exécution ». Il existe plusieurs autres techniques plus sophistiquées que vous pouvez appliquer pour mettre en œuvre la « sécurité du type » des paramètres, autrement dit pour assurer que les erreurs dans les arguments seront interceptées par le compilateur. Vous pouvez par exemple créer des objets qui représentent les types des paramètres, utiliser des constantes statiques ou, en Java 5, employer des *enums*.

Q: Je ne suis pas sûr d'avoir bien compris la différence entre Fabrique Simple et Fabrication. Ils ont l'air très similaires, excepté que, dans Fabrication, la classe qui retourne la pizza est une sous-classe. Pouvez-vous expliquer ?

R: Vous avez raison : il est vrai que les sous-classes ressemblent beaucoup à Fabrique Simple, mais Fabrique Simple ne sert qu'une fois tandis que Fabrication vous permet de créer une structure qui laisse aux sous-classes le choix de l'implémentation qui sera utilisée. Par exemple, la méthode commanderPizza() de la Fabrication fournit un cadre général pour créer des pizzas qui s'appuie sur une méthode de fabrication pour créer les classes concrètes qui entrent en jeu dans la confection d'une pizza. En sous-classant la classe Pizzeria, vous décidez quels sont les produits concrets qui participent à la création de la pizza que commanderPizza() retourne. Comparez cette technique avec Fabrique Simple, qui vous fournit un moyen d'encapsuler les objets, mais sans la souplesse de Fabrication parce qu'elle ne permet pas de faire varier les objets que vous créez.



Maître et disciple...

Maître : *Petit scarabée, dis-moi comment se passe ta formation.*

Disciple : *Maître, j'ai continué à étudier « l'encapsulation de ce qui varie »...*

Maître : *Continue...*

Disciple : *J'ai compris qu'on pouvait encapsuler le code qui crée les objets. Quand on a du code qui instancie des classes concrètes, c'est un domaine voué en permanence au changement. J'ai appris une technique nommée « fabrications » qui permet d'encapsuler ce comportement d'instanciation.*

Maître : *Et ces « fabrications », que nous apportent-elles ?*

Disciple : *De nombreux avantages. En plaçant tout le processus de création dans une méthode ou dans un objet, j'évite de dupliquer du code et de multiplier les opérations de maintenance. Cela signifie également que les clients ne dépendent que des interfaces et non des classes concrètes nécessaires pour instancier les objets. Comme je l'ai appris, cela me permet de programmer pour une interface, non une implémentation, et cela rend mon code plus souple et plus extensible.*

Maître : *Oui, Scarabée, tu commences à avoir des réflexes objet. As-tu des questions à poser à ton maître aujourd'hui ?*

Disciple : *Maître, je sais qu'en encapsulant la création des objets je code en m'appuyant sur des abstractions et je découple mon code client des implémentations réelles. Mais mon code de fabrication doit toujours utiliser des classes concrètes pour instancier des objets réels. Ne suis-je pas en train de m'égarer ?*

Maître : *Scarabée, la création des objets est une réalité de l'existence ; si nous ne créons pas d'objets, nous n'écrirons jamais un seul programme Java. Mais, grâce à la connaissance de cette réalité, nous pouvons concevoir nos programmes afin d'encapsuler notre code de création, tout comme nous enfermons nos brebis dans un parc, pour les empêcher de s'égarer elles aussi. Une fois que nous avons enfermé ce code, nous pouvons en prendre soin et le protéger. Mais si nous le laissons courir en liberté, nous ne récolterons jamais de laine."*

Disciple : *Maître, je vois la vérité maintenant.*

Maître : *Je savais que tu comprendrais. Va, maintenant et médite sur la dépendance des objets.*

Une Pizzeria très dépendante



À vos crayons

Faisons comme si vous n'aviez jamais entendu parler de fabrication OO. Voici une version de la Pizzeria qui n'utilise pas de fabrication. Faites le compte du nombre d'objets concrets dont cette classe dépend. Si vous ajoutez des pizzas de style Marseille à cette Pizzeria, de combien d'objets dépendra-t-elle alors ?

```
public class PizzeriaDependante {

    public Pizza creerPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("Brest")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleBrest();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleBrest();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleBrest();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleBrest();
            }
        } else if (style.equals("Strasbourg")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleStrasbourg();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleStrasbourg();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleStrasbourg();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleStrasbourg();
            }
        } else
        {
            System.out.println("Erreur : type de pizza invalide");
            return null;
        }
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }
}
```

Gère toutes les pizzas
de style Brest

Gère toutes les
pizzas de style
Strasbourg

Vous pouvez insérer
vos réponses ici :

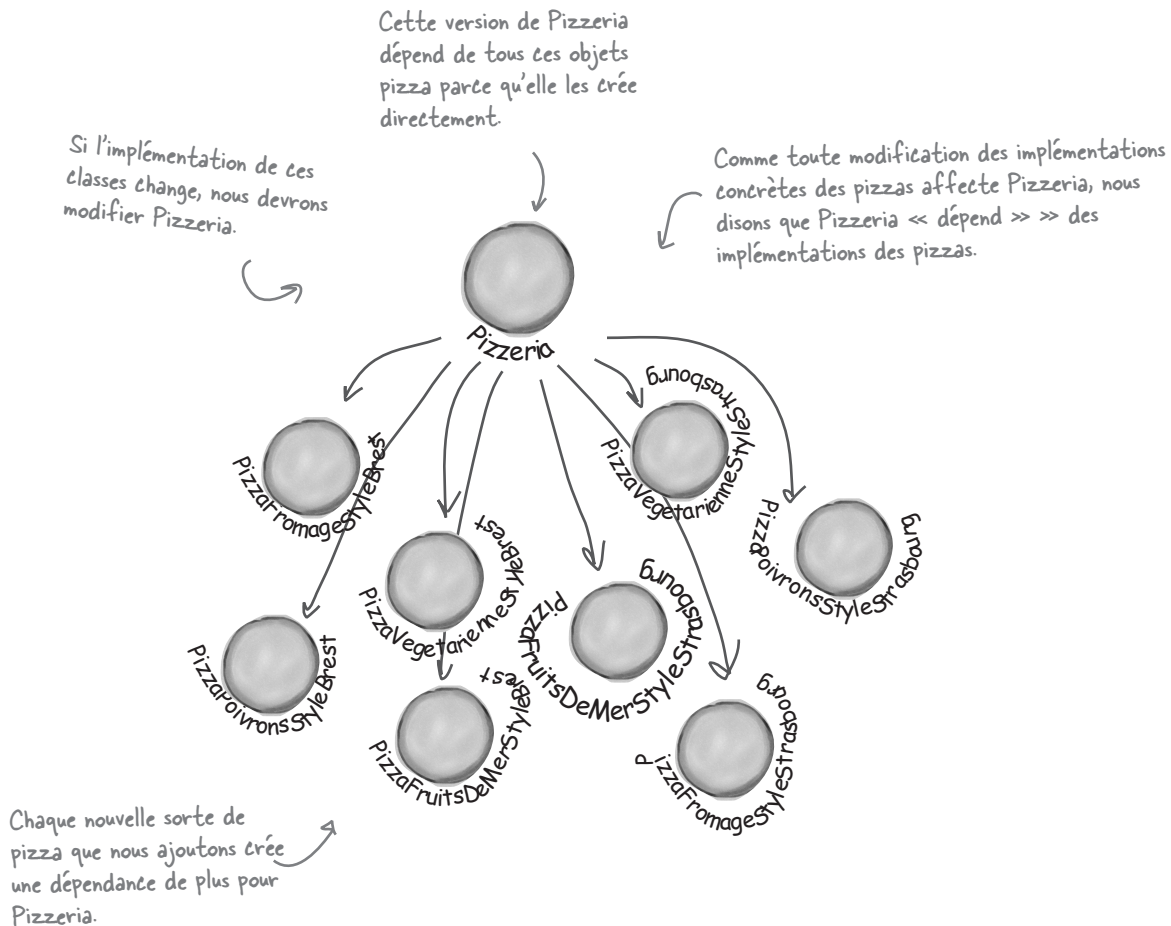
nombre

nombre avec Marseille

Problèmes de dépendance des objets

Quand vous instanciez directement un objet, vous dépendez de sa classe concrète. Jetez un coup d'œil à notre Pizzeria hyper dépendante de la page précédente. Elle crée tous les objets pizza en plein dans la classe Pizzeria au lieu de déléguer à une fabrication.

Si nous traçons un diagramme représentant cette version de Pizzeria et de tous les objets dont elle dépend, voici à quoi il ressemble :

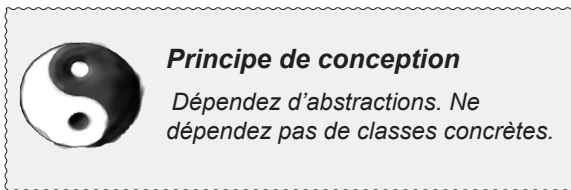


Le principe d'inversion des dépendances

Une notion doit être bien claire : la réduction des dépendances aux classes concrètes dans notre code est une « bonne chose ». En fait, nous avons un principe de conception OO qui formalise cette notion. Elle porte même un beau nom bien ronflant : *Principe d'inversion des dépendances*.

← Encore une expression que vous pouvez employer pour impressionner les cadres ! Le montant de votre augmentation dépassera largement ce que vous aurez coûté ce livre, et vous vous attirerez l'admiration de vos collègues développeurs.

Voici sa définition générale :



À première vue, ce principe ressemble beaucoup à « Programmez pour une interface, non pour une implémentation ». Mais s'il est similaire, le Principe d'inversion des dépendances affirme quelque chose de beaucoup plus fort à propos de l'abstraction. Il suggère que nos composants de haut niveau ne doivent pas dépendre des composants de bas niveau, mais que les deux doivent dépendre d'abstractions.

Mais qu'est-ce que cela peut bien vouloir dire ?

Eh bien commençons par observer de nouveau le diagramme de la page précédente. Pizzeria est notre « composant de haut niveau », les implémentations de pizzas sont nos « composants de bas niveau » et il est clair que Pizzeria dépend des classes de pizza concrètes.

Maintenant, ce principe nous dit que nous devons plutôt écrire notre code afin de dépendre d'abstractions, non de classes concrètes. Cela vaut à la fois pour nos modules de haut niveau et pour nos modules de bas niveau.

Mais comment procéder ? Réfléchissons comment nous appliquerions ce principe à notre implémentation de Pizzeria hyper dépendante...

← Un composant « de haut niveau » est une classe dont le comportement est défini en termes d'autres composants « de bas niveau ».

Par exemple, Pizzeria est un composant de haut niveau parce que son comportement est défini en termes de pizzas – il crée tous les objets pizza différents, les prépare, les fait cuire, les découpe et les emballe, tandis que les pizzas qu'il utilise sont les composants de bas niveau.

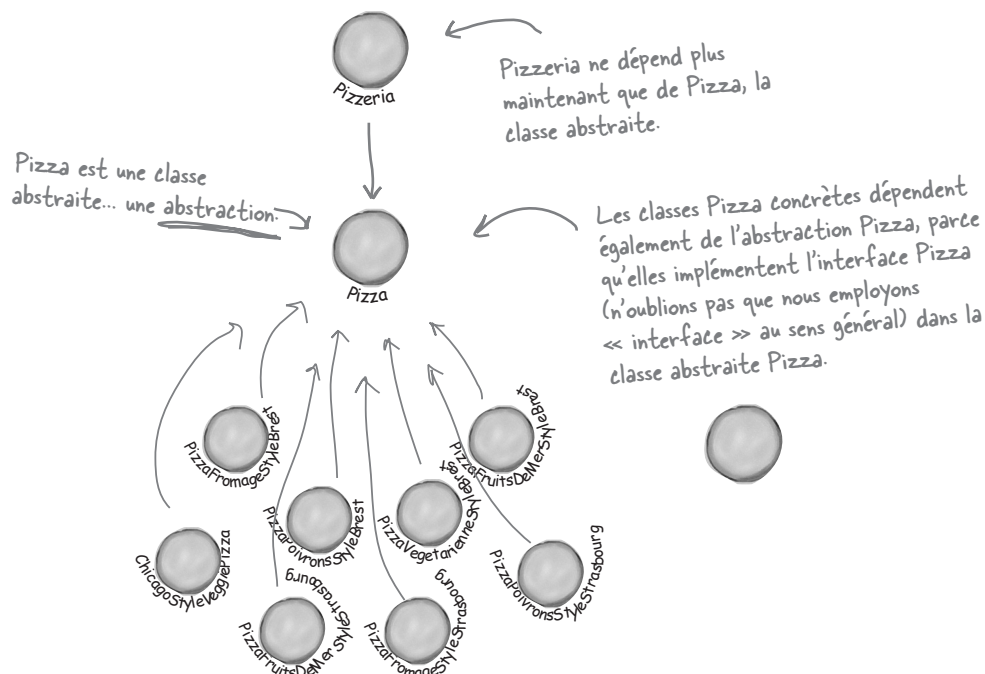
Appliquer le principe

Maintenant, le principal problème de la Pizzeria hyper dépendante est qu'elle dépend de chaque type de pizza parce qu'elle instancie les types concrets dans sa méthode `commanderPizza()`.

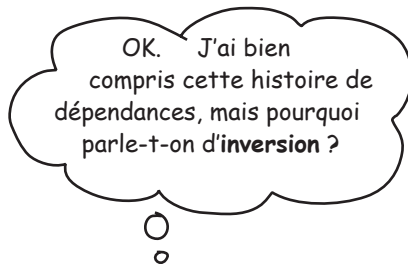
Si nous avons bien créé une abstraction, `Pizza`, nous créons néanmoins les pizzas concrètes dans ce code, ce qui nous empêche d'exploiter réellement les avantages de cette abstraction.

Comment extraire ces instanciations de la méthode `commanderPizza()` ? Eh bien, comme nous le savons, c'est exactement ce que le pattern `Fabrication` nous permet de faire.

Ainsi, après avoir appliqué Fabrication, nous obtenons le diagramme suivant :



Après avoir appliqué Fabrication, vous remarquerez que notre composant de haut niveau, Pizzeria, et nos composants de bas niveau, les pizzas, dépendent tous de Pizza, l'abstraction. Fabrication n'est pas la seule technique qui permet de respecter le Principe d'inversion des dépendances, mais c'est l'une des plus puissantes.

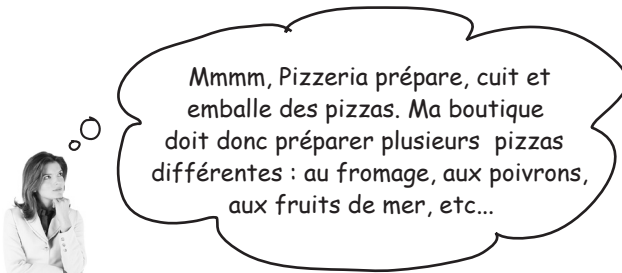


Où est l' « inversion » dans le Principe d'inversion des dépendances ?

Dans l'expression « principe d'inversion des dépendances », le mot « inversion » indique qu'on inverse la façon dont vous pourriez penser votre conception OO. Regardez le diagramme de la page précédente et remarquez que les composants de bas niveau dépendent maintenant d'une abstraction de plus haut niveau. De plus, le composant de haut niveau est également lié à la même abstraction. Ainsi, le graphe de dépendances que nous avons tracé il y a deux pages s'est inversé, et les deux types de modules dépendent maintenant de l'abstraction.

Voyons également quelle est la pensée qui sous-tend le processus de conception classique et comment l'introduction de ce principe peut inverser notre façon d'envisager la conception...

Inverser votre pensée...



Bien. Ainsi, vous devez implémenter une Pizzeria. Quelle est la première pensée qui vous vient à l'esprit ?

OK. Vous commencez par le sommet et vous descendez jusqu'aux classes concrètes. Mais, comme vous l'avez vu, vous ne voulez pas que votre boutique ait connaissance des types concrets de pizzas, parce qu'elle dépendrait alors de ces classes concrètes !

Maintenant, « inversez » votre pensée... Au lieu de commencer par en haut, commencez par les Pizzas et réfléchissez à ce que vous pouvez abstraire.

Très bien ! Vous pensez à l'abstraction *Pizza*. Maintenant, revenez en arrière et réfléchissez de nouveau à la conception de Pizzeria.

Vous brûlez. Mais, pour ce faire, vous devez vous appuyer sur une fabrication pour extraire ces classes concrètes de votre Pizzeria. Cela fait, vos différents types concrets ne dépendront que d'une abstraction et votre boutique également. Nous avons pris une conception dans laquelle la boutique dépendait de classes concrètes et nous avons inversé ces dépendances (ainsi que votre mode de pensée).

Quelques conseils pour vous aider à appliquer le principe...

Les lignes directrices suivantes peuvent vous aider à éviter les conceptions OO qui enfreignent le Principe d'inversion des dépendances :

- Aucune variable ne doit contenir une référence à une classe concrète.
- Aucune classe ne doit dériver d'une classe concrète.
- Aucune classe ne doit redéfinir une méthode implémentée dans une classe de base.

Si vous utilisez l'opérateur `new`, vous faites référence à une classe concrète. Utilisez une fabrication pour contourner le problème !

Si vous sous-classez une classe concrète, vous dépendez de cette classe concrète.

Sous-classez une abstraction, comme une interface ou une classe abstraite.

Si vous redéfinissez une méthode implémentée, votre classe de base n'était pas une abstraction pour commencer. Ces méthodes implémentées dans la classe de base sont connues pour être partagées par toutes les sous-classes.

Mais, attendez... Ces conseils sont impossibles à suivre. Si je les applique tous, je n'écirai jamais un seul programme !

Vous avez entièrement raison ! Comme pour beaucoup de nos principes, il s'agit de lignes directrices vers lesquelles vous devez tendre et non de règles que vous devez appliquer tout le temps. De toute évidence, tout programme Java jamais écrit viole au moins l'un de ces principes !

Mais si vous intériorisez ces conseils et que vous les avez toujours dans un coin de votre esprit quand vous concevez une application, vous saurez que vous enfreignez le principe et vous aurez une bonne raison de le faire. Par exemple, si vous avez une classe qui n'est pas susceptible de changer et que vous le savez, ce ne sera pas la fin du monde si vous instanciez une classe concrète. Réfléchissez : nousinstancions des objets `String` en permanence sans y regarder à deux fois.

Est-ce que nous violons le principe ? Oui. Est-ce que c'est mal ? Non. Pourquoi ? Parce qu'il est très improbable que la classe `String` change.

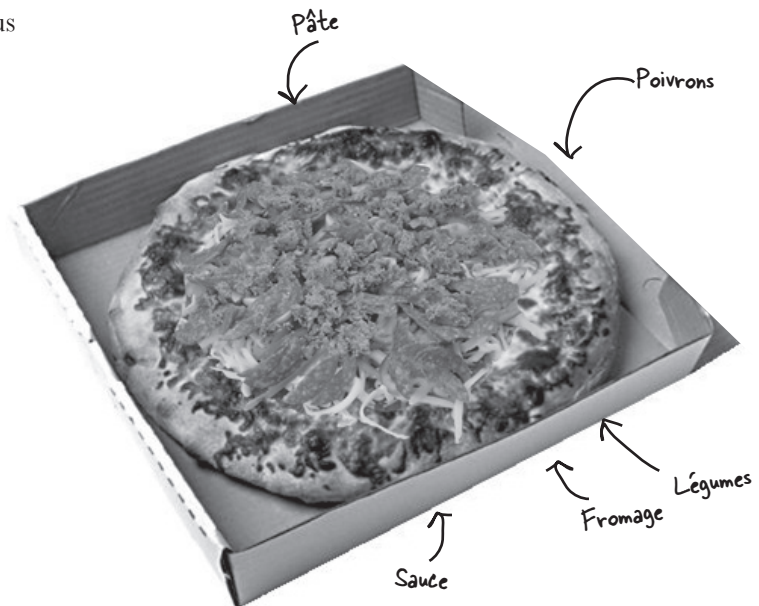
Si en revanche une classe que vous avez écrite est susceptible de changer, Vous disposez de techniques intéressantes, telle Fabrication, pour encapsuler le changement.



Pendant ce temps, de retour à la Pizzeria...

La conception de la Pizzeria prend réellement forme : nous disposons d'une structure souple et nous respectons de bons principes de conception.

Maintenant, la clé du succès de votre boutique d'Objectville a toujours résidé dans le fait qu'elle employait des garnitures fraîches et de qualité. Or, avec la nouvelle structure, vous avez découvert que si vos franchises appliquaient bien vos *procédures*, certaines d'entre elles substituaient des garnitures de qualité inférieure pour diminuer les coûts et augmenter leurs marges. Vous savez que vous devez faire quelque chose, sous peine de voir endommager à long terme la réputation de la marque d'Objectville !

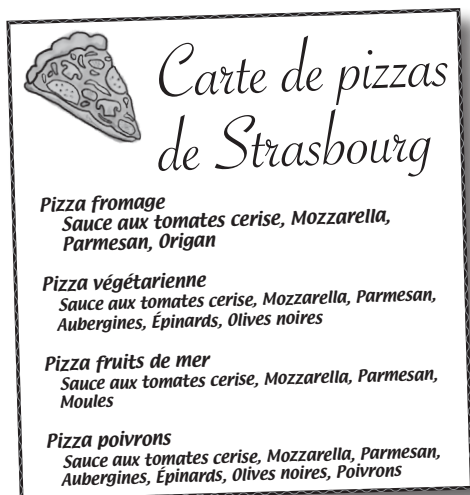


Assurer la cohérence de vos garnitures

Comment allez-vous donc veiller à ce que chaque franchise utilise des garnitures de qualité ? Vous allez construire une fabrique qui les produit et qui les leur expédie !

Mais ce plan pose un problème : les franchises sont situées dans différentes régions et ce qui est une sauce tomate à Brest n'est pas une sauce tomate à Strasbourg.

Ainsi, vous avez un ensemble de garnitures qui doit être livré à Brest et un ensemble *différent* qui est destiné à Strasbourg. Voyons cela de plus près :



Nous avons les mêmes familles de produits (pâte, sauce, fromage, légumes, etc...) mais des implémentations différentes selon la région.

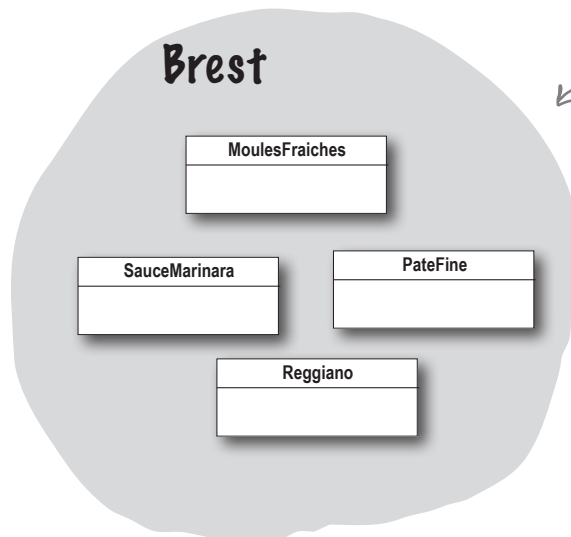


Familles de garnitures...

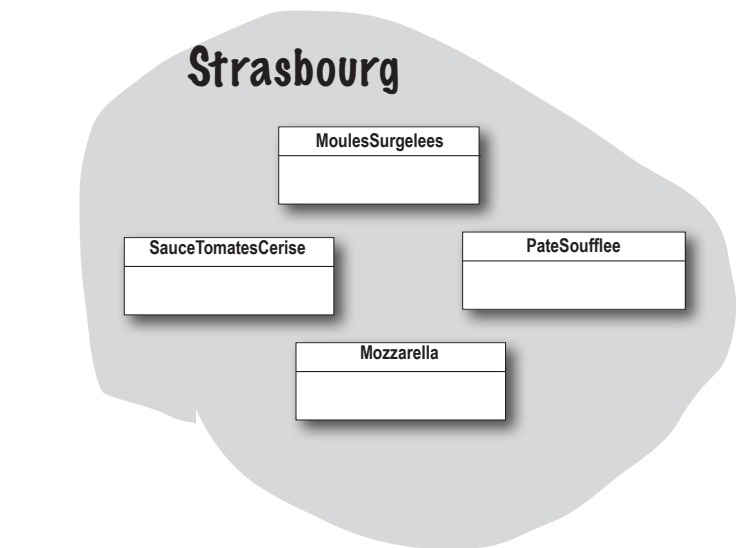
Brest utilise un ensemble de garnitures et Strasbourg un autre.

Étant donnée la popularité de la boutique d'Objectville, vous devrez probablement envoyer d'ici peu un autre ensemble de garnitures régionales à Marseille. Et après ? Lille, peut-être ?

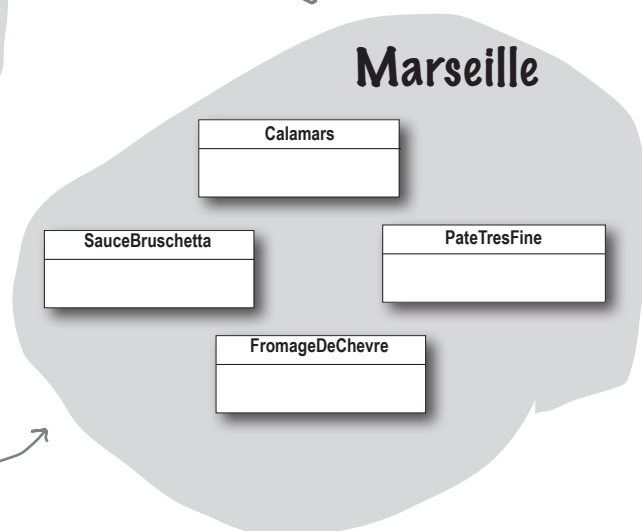
Pour ce faire, vous allez devoir imaginer comment gérer les familles de garnitures.



Chaque famille se compose d'un type de pâte, un type de sauce, un type de fromage et une garniture de fruits de mer (plus quelques autres que nous n'avons pas représentés comme les légumes et les épices).



Toutes les pizzas d'Objectville sont fabriquées à partir des mêmes composants, mais chaque région a une implémentation différente de ces composants.



Au total, ces trois régions constituent des familles d'ingrédients, chaque région implémentant une famille complète de garnitures.

Construire les fabriques d'ingrédients

Nous allons maintenant construire une fabrication pour créer nos garnitures. La fabrication sera responsable de la création de chaque ingrédient de la famille. Autrement dit, la fabrication devra créer de la pâte, de la sauce, du fromage, etc... Vous verrez bientôt comment nous allons gérer les différences régionales.

Mais commençons par définir une interface pour la fabrication qui va créer toutes nos garnitures :

```
public interface FabriqueIngredientsPizza {  
  
    public Pate creerPate ();  
    public Sauce creerSauce();  
    public Fromage creerFromage();  
    public Legumes[] creerLegumes();  
    public Poivrons creerPoivrons();  
    public Moules creerMoules();  
}
```



Pour chaque ingrédient, nous définissons une méthode de création dans notre interface.



Plein de nouvelles classes ici, une par ingrédient.

Si nous avons un « mécanisme » commun à implémenter dans chaque instance de fabrication, nous aurions pu créer à la place une classe abstraite...

Voici ce que nous allons faire :

- ❶ Construire une fabrique pour chaque région. Pour ce faire, vous allez créer une sous-classe de `FabriqueIngredientsPizza` qui implémente chaque méthode de création.
- ❷ Implémenter un ensemble de classes ingrédients qui seront utilisées avec la fabrique, comme `Reggiano`, `PoivronsRouges` et `PateSoufflee`. Ces classes peuvent être partagées entre les régions en fonction des besoins.
- ❸ Puis nous assemblerons tout en incorporant nos fabriques dans l'ancien code de `Pizzeria`.

Construire la fabrique d'ingrédients de Brest

Bien, voici l'implémentation de la fabrique d'ingrédients de Brest. Celle-ci est spécialisée dans la sauce Marinara, le parmigiano reggiano, les moules fraîches...

La fabrique d'ingrédients de Brest implémente l'interface pour toutes les fabrications d'ingrédients

```
public class FabriqueIngredientsPizzaBrest implements FabriqueIngredient-
sPizza {
```

```
    public Pate creerPate() {
        return new PateFine();
    }
```

Pour chaque ingrédient de la famille, nous créons la version Brest.

```
    public Sauce creerSauce() {
        return new SauceMarinara();
    }
```

```
    public Fromage creerFromage() {
        return new Reggiano();
    }
```

```
    public Legume[] creerLegumes() {
        Legume legume [] = { new Ail(), new Oignon(), new Champignon(), new PoivronRouge() };
        return legume;
    }
```

```
    public Poivrons creerPoivrons() {
        return new PoivronsEnRondelles();
    }
```

```
    public Moules creerMoules() {
        return new MoulesFraiches();
    }
```

```
}
```

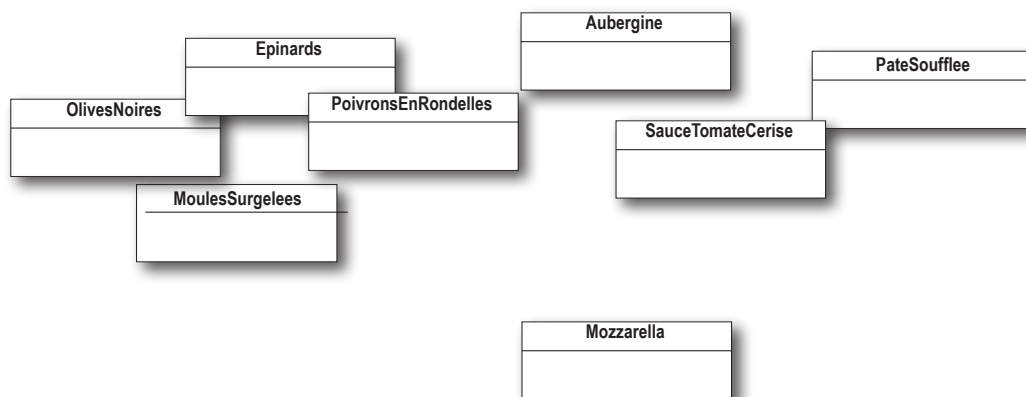
Comme Brest est sur la côte, nous avons des moules fraîches. Strasbourg devra se contenter de moules surgelées.

Pour les légumes, nous retournons un tableau de Légumes. Ici, nous avons codé les légumes en dur. Nous aurions pu trouver quelque chose de plus élaboré, mais comme cela n'ajouterait rien à l'explication du pattern, nous simplifions.

Les meilleurs poivrons en rondelles. Ils sont partagés entre Brest et Strasbourg. N'oubliez pas d'en utiliser page suivante quand vous implémenterez vous-même la fabrique de Strasbourg

À vos crayons

Écrivez la Fabrique IngredientsPizzaStrasbourg. Vous pouvez référencer les classes ci-dessous dans votre implémentation :



Retravaillons les pizzas...

Nos fabriques sont en place et prêtes à produire des ingrédients de qualité. Il ne nous reste plus qu'à retravailler nos Pizzas pour qu'elles n'emploient que les ingrédients produits par les fabriques. Nous allons commencer par notre classe abstraite, Pizza :

```
public abstract class Pizza {
    String nom;
    Pate pate;
    Sauce sauce;
    Legume legume[];
    Fromage fromage;
    Poivrons poivrons;
    Moules moules;

    abstract void preparer();

    void cuire() {
        System.out.println("Cuisson 25 minutes à 180°");
    }

    void couper() {
        System.out.println("Découpage en parts triangulaires");
    }

    void emballer() {
        System.out.println("Emballage dans une boîte officielle");
    }

    void setNom(String nom) {
        this.nom = nom;
    }

    String getNom() {
        return nom;
    }

    public String toString() {
        // code qui affiche la pizza
    }
}
```

Chaque pizza contient un ensemble d'ingrédients utilisés dans sa préparation.

Nous avons maintenant rendu la méthode `preparer()` abstraite.

C'est là que nous allons collecter les ingrédients nécessaires pour la pizza. Bien entendu, ils proviennent de la fabrication d'ingrédients.

Nos autres méthodes demeurent les mêmes : seule la méthode `preparer()` a changé.

Retravaillons les pizzas (suite)

Maintenant que nous disposons d'une Pizza abstraite comme base, il est temps de créer les pizzas de styles Brest et Strasbourg – seulement cette fois les ingrédients viendront directement de la fabrication. C'est la fin de l'époque à laquelle les franchisés lésinaient sur les ingrédients !

Quand nous avons écrit le code inspiré de Fabrication, nous avons une classe `PizzaFromageBrest` et une classe `PizzaFromageStrasbourg`. Si vous observez ces deux classes, vous constatez qu'une seule chose diffère : l'emploi d'ingrédients régionaux. Les pizzas sont faites exactement de la même manière (pâte + sauce + fromage). Il en va de même pour les autres pizzas : Végétarienne, Fruits de mer, etc. Les étapes de la réparation sont les mêmes, mais les garnitures sont différentes.

Vous voyez donc que nous n'avons pas besoin de deux classes pour chaque pizza : la fabrique d'ingrédients va gérer les différences régionales pour nous.

Voici la `Pizza Fromage` :

```
public class PizzaFromage extends Pizza {
    FabriqueIngredientsPizza fabriqueIngredients;

    public PizzaFromage(FabriqueIngredientsPizza fabriqueIngredients) {
        this.fabriqueIngredients = fabriqueIngredients;
    }

    void preparer() {
        System.out.println("Préparation de " + nom);
        pate = fabriqueIngredients.creerPate();
        sauce = fabriqueIngredients.creerSauce();
        fromage = fabriqueIngredients.creerFromage();
    }
}
```

Maintenant, pour faire une pizza, nous avons besoin d'une fabrication qui fournit les ingrédients. Une fabrication est donc transmise au constructeur de chaque classe `Pizza` et stockée dans une variable d'instance.

← C'est là que c'est magique !

La méthode `preparer()` parcourt les étapes de la création d'une pizza au fromage. Chaque fois qu'elle a besoin d'un ingrédient, elle demande à la fabrication de le produire.



Code à la loupe

Le code de Pizza code utilise la fabrication avec laquelle il a été composé pour produire les ingrédients employés dans la pizza. Les ingrédients produits dépendent de la fabrication spécifique. La classe ne s'en soucie pas : elle sait comment faire des pizzas. Elle est maintenant découplée des différences régionales et sera facile à réutiliser quand il y aura des fabrications pour les Montagnes rocheuses, la Floride et au delà.

sauce = fabriqueIngrédients.creerSauce();

Nous affectons à la variable d'instance de Pizza une référence à la sauce spécifique utilisée dans cette pizza.

Voici notre fabrique d'ingrédients. La Pizza se moque de la fabrique utilisée tant que c'est une fabrique d'ingrédients.

La méthode `creerSauce()` retourne la sauce employée dans sa région. Si c'est une fabrication d'ingrédients Brest, nous aurons une sauce marinara. La méthode `creerSauce()` retourne la sauce employée dans sa région. Si c'est une fabrication d'ingrédients Brest, nous aurons une sauce marinara.

Vérifions également le code de `PizzaFruitsDeMer` :

```
public class PizzaFruitsDeMer extends Pizza {
    FabriqueIngrédientsPizza fabriqueIngrédients;

    public PizzaFruitsDeMer(FabriqueIngrédientsPizza fabriqueIngrédients) {
        this.fabriqueIngrédients = fabriqueIngrédients;
    }

    void preparer() {
        System.out.println("Préparation de " + nom);
        pate = fabriqueIngrédients.creerPate();
        sauce = fabriqueIngrédients.creerSauce();
        fromage = fabriqueIngrédients.creerFromage();
        fruitsDeMer = fabriqueIngrédients.creerMoules();
    }
}
```

`PizzaFruitsDeMer` cache également une fabrique d'ingrédients.

Pour faire une pizza aux fruits de mer, la méthode `preparer()` se procure les bons ingrédients auprès de la fabrique locale.

Si c'est la fabrique de Brest, ce seront des moules fraîches ; si c'est celle de Strasbourg, elles seront surgelées.

Revisitons nos pizzerias

Nous y sommes presque. Il suffit d'aller faire un tour rapide chez nos franchisés pour vérifier qu'ils utilisent les bonnes Pizzas. Il faut également leur donner une référence à leur fabrique d'ingrédients locale :

```
public class PizzeriaBrest extends Pizzeria {  
  
    protected Pizza creerPizza(String item) {  
        Pizza pizza = null;  
        FabriqueIngredientsPizza fabriqueIngredients =  
            new FabriqueIngredientsPizzaBrest();  
        if (choix.equals("fromage")) {  
  
            pizza = new PizzaFromage(fabriqueIngredients);  
            pizza.setNom("Pizza au fromage style Brest");  
        } else if (choix.equals("vegetarienne")) {  
  
            pizza = new PizzaVegetarienne(fabriqueIngredients);  
            pizza.setNom("Pizza végétarienne style Brest");  
  
        } else if (choix.equals("fruitsDeMer")) {  
  
            pizza = new PizzaFruitsDeMer(fabriqueIngredients);  
            pizza.setNom("Pizza aux fruits de mer style Brest");  
        } else if (choix.equals("poivrons")) {  
  
            pizza = new PizzaPoivrons(fabriqueIngredients);  
            pizza.setNom("Pizza aux poivrons style Brest");  
        }  
        return pizza;  
    }  
}
```

La boutique de Brest est composée avec une fabrique d'ingrédients Brest. Elle sera utilisée pour produire les garnitures de toutes les pizzas style Brest.

Nous transmettons maintenant à chaque pizza la fabrique qu'il faut utiliser pour produire ses ingrédients.

Revenez page précédente et vérifiez que vous avez compris comment la pizza et la fabrique collaborent !

Pour chaque type de Pizza, nous instancions une nouvelle Pizza et nous lui transmettons la fabrique dont elle a besoin pour obtenir ses ingrédients.



MUSCLEZ VOS NEURONES

Comparez cette version de la méthode `creerPizza()` avec celle de l'implémentation de Fabrication que nous avons vue plus haut dans ce chapitre.

Qu'avons-nous fait ?

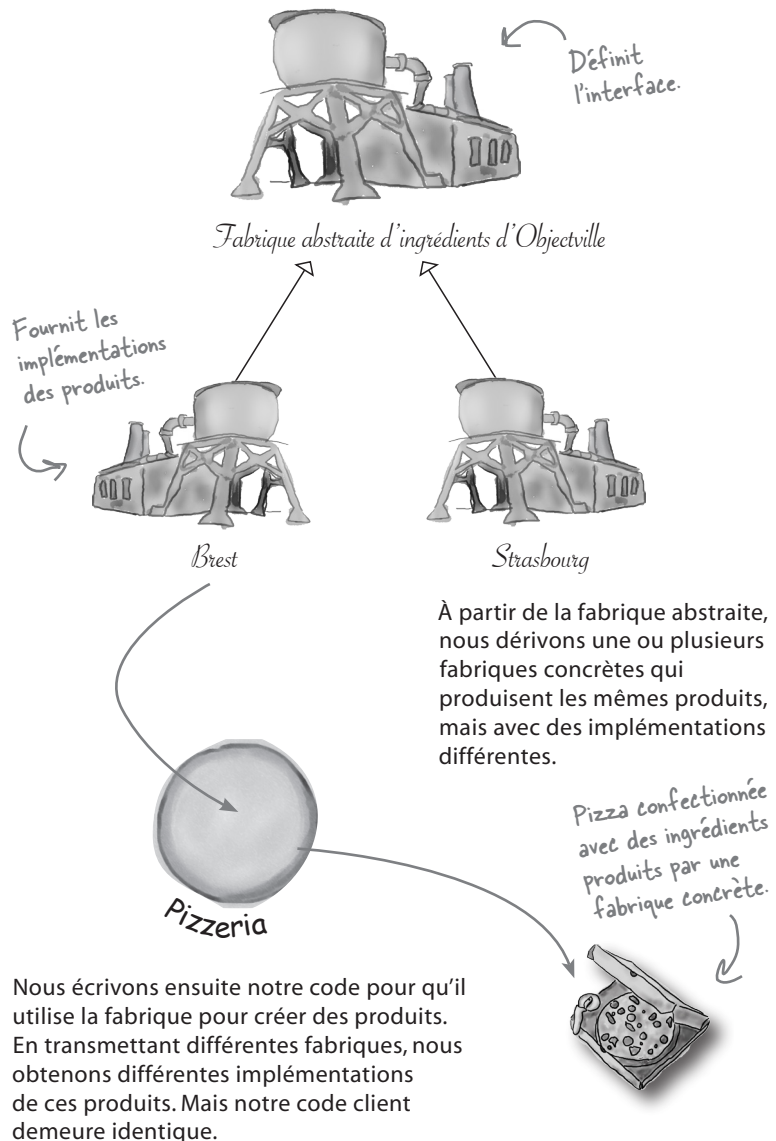
Nous avons apporté toute une série de modifications au code. Mais qu'avons-nous fait exactement ?

Nous avons fourni un moyen de créer une famille d'ingrédients pour les pizzas en introduisant un nouveau type de fabrication nommé Fabrique Abstraite.

Une Fabrique Abstraite fournit une interface pour créer des familles d'objets. En écrivant du code qui utilise cette interface, nous découplons ce code de la fabrique réelle qui crée les produits. Cela nous permet d'implémenter toute une gamme de fabriques qui créent des produits destinés à différents contextes – différentes régions, différents systèmes d'exploitation ou différents « look and feel ».

Comme notre code est découplé des produits réels, nous pouvons substituer différentes fabriques pour obtenir différents comportements (par exemple obtenir une sauce marinara au lieu d'une sauce aux tomates cerise).

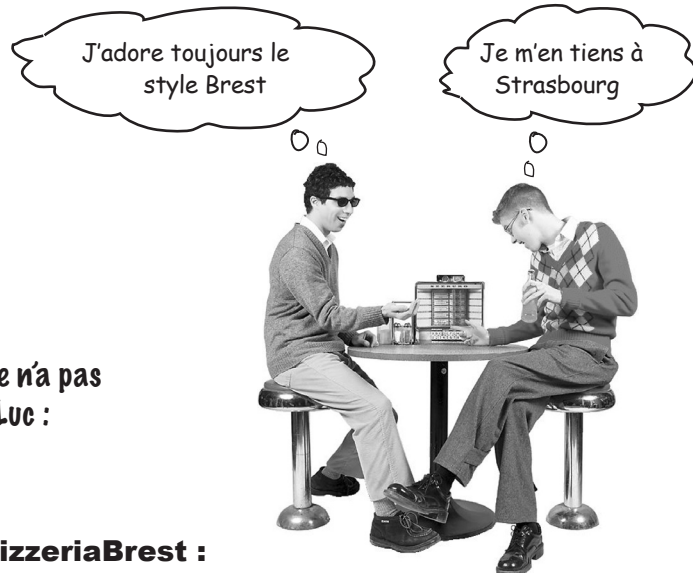
Une Fabrique Abstraite fournit une interface pour une famille de produits. Mais qu'est-ce qu'une famille ? Dans notre cas, ce sont tous les éléments nécessaires pour confectionner une pizza : pâte, sauce, fromage et autres garnitures.



Luc et Michel ont encore envie de pizzas...

Luc et Michel ne sont jamais rassasiés des pizzas d'Objectville !
Mais ce qu'ils ignorent, c'est l'existence des nouvelles fabriques d'ingrédients. Maintenant, quand ils passent commande...

Dans les
coulisses



La première partie du processus de commande n'a pas changé d'un iota. Revoyons la commande de Luc :

1 Tout d'abord, il nous faut une PizzeriaBrest :

```
Pizzeria pizzeriaBrest = new PizzeriaBrest();
```

Crée une instance de
PizzeriaBrest.

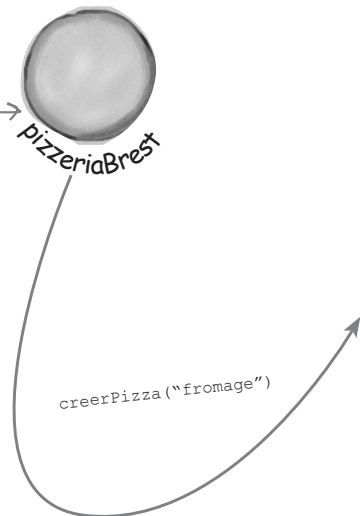
2 Maintenant que nous avons une boutique, nous pouvons prendre une commande :

```
pizzeriaBrest.commanderPizza("fromage");
```

La méthode commanderPizza() est
appelée sur l'instance de pizzeriaBrest.

3 La méthode commanderPizza() commence par appeler la méthode creerPizza() :

```
Pizza pizza = creerPizza("fromage");
```

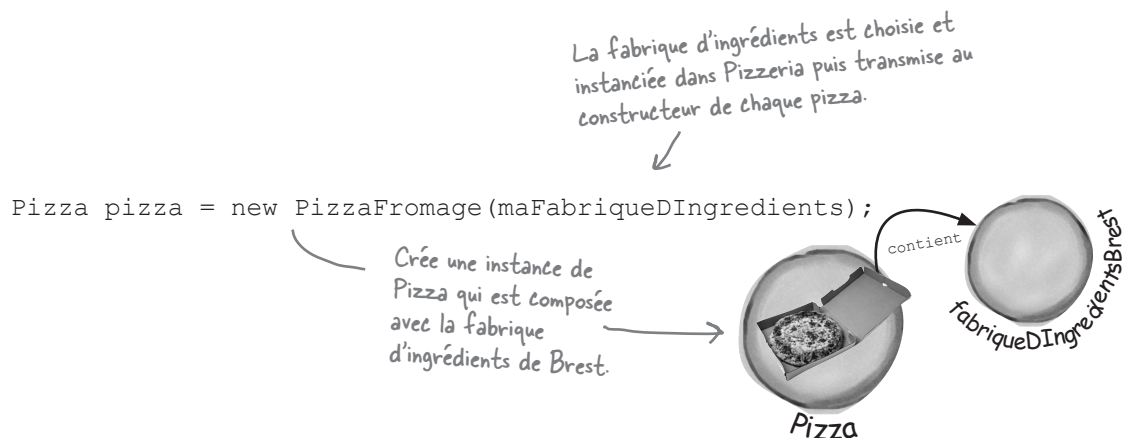


À partir de là, les choses changent, parce que nous utilisons une fabrique d'ingrédients

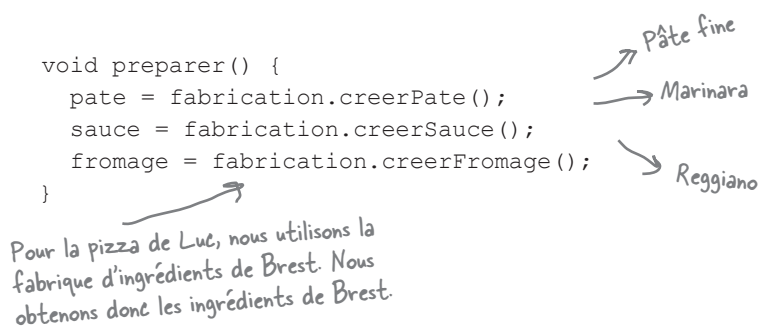
Dans les coulisses



- 4** Quand la méthode `creerPizza()` est appelée, c'est là que notre fabrique d'ingrédients entre en scène :



- 5** Nous devons ensuite préparer la pizza. Une fois la méthode `preparer()` appelée, on demande à la fabrique de préparer les ingrédients:



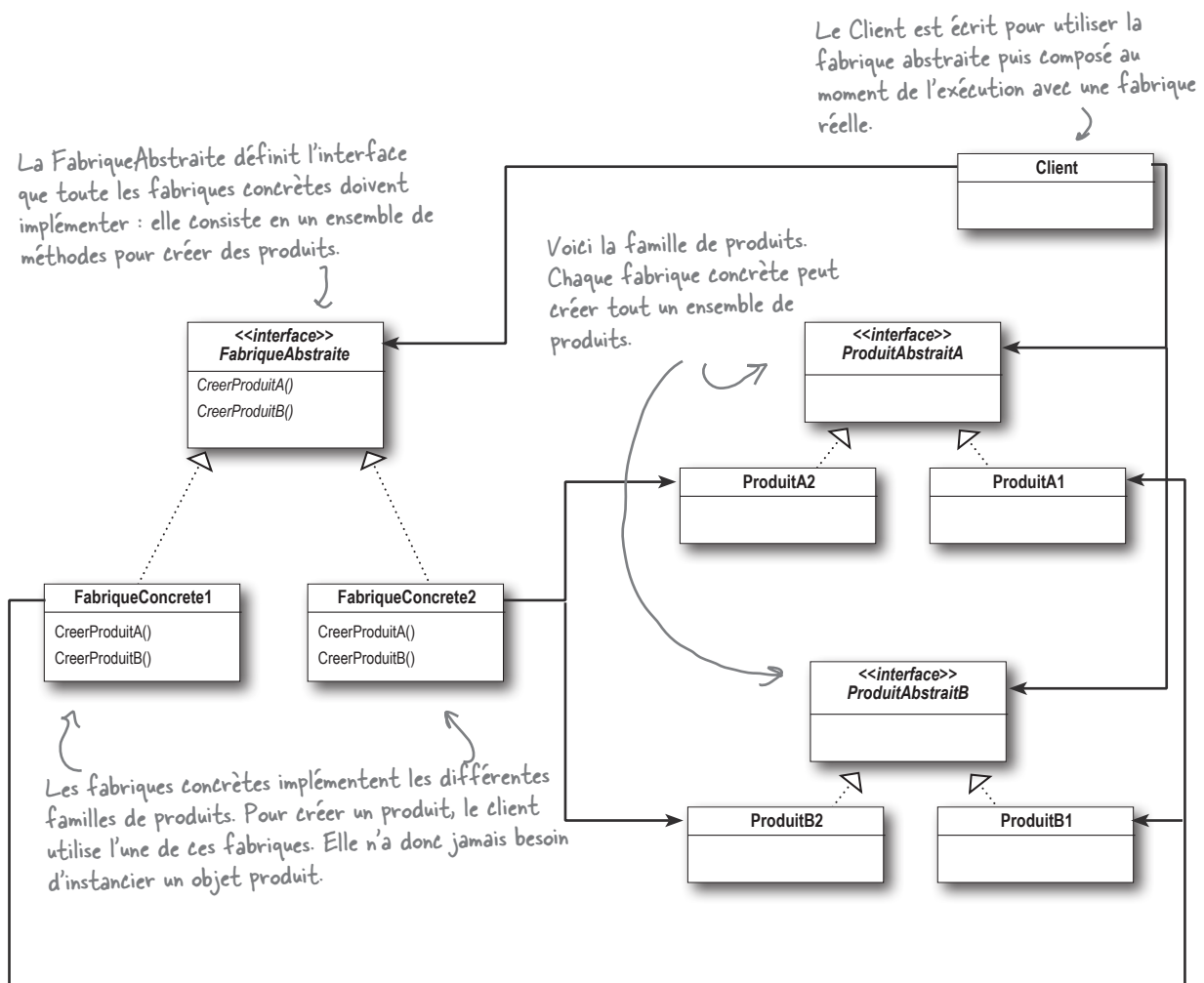
- 6** Enfin, nous avons la pizza préparée en main et la méthode `commanderPizza()` la fait cuire, la découpe et l'emballage.

Le pattern Fabrique Abstraite : définition

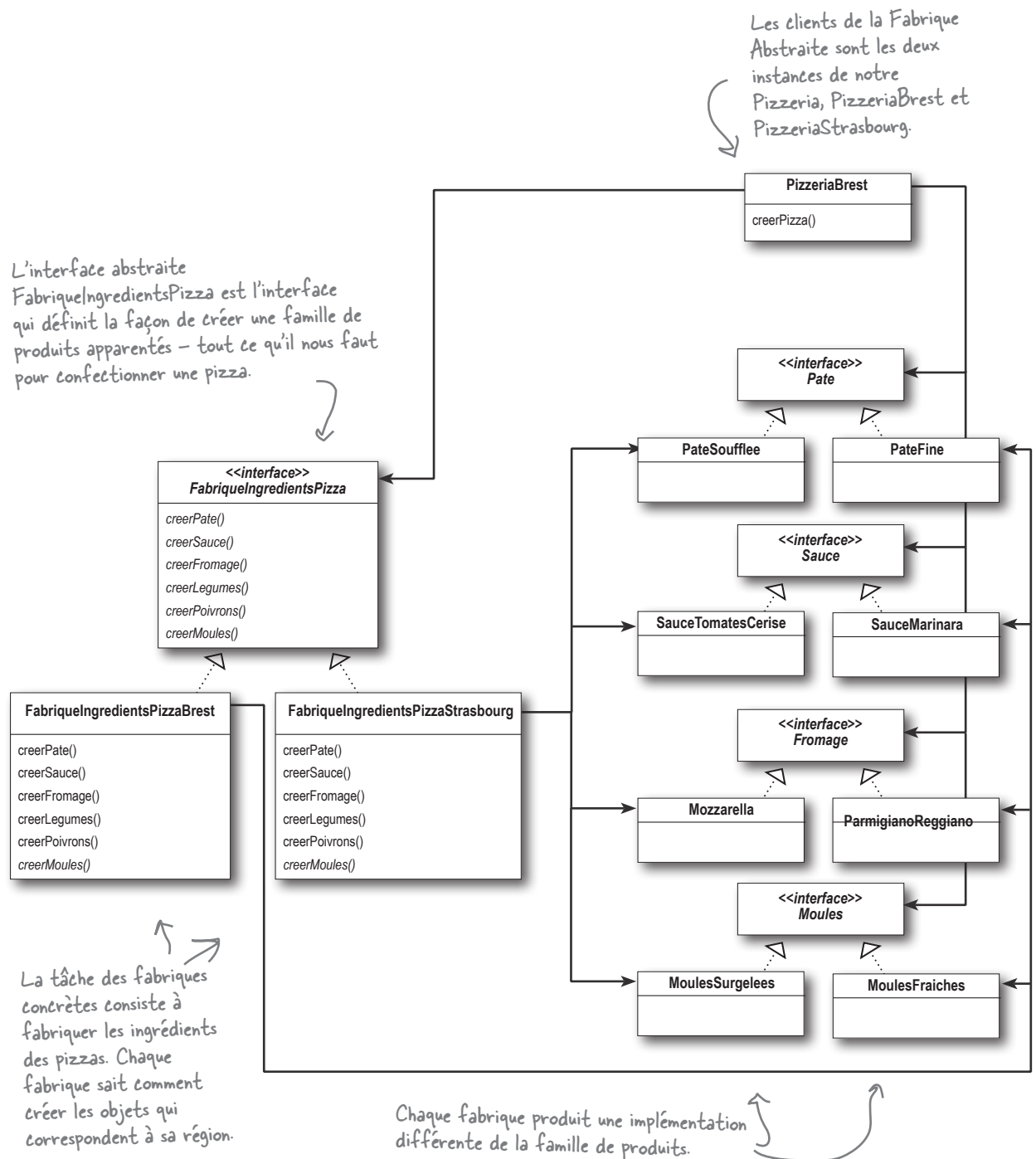
Nous allons ajouter un nouveau pattern de fabrication à notre famille de patterns. Celui-ci nous permet de créer des familles de produits. Voyons sa définition officielle :

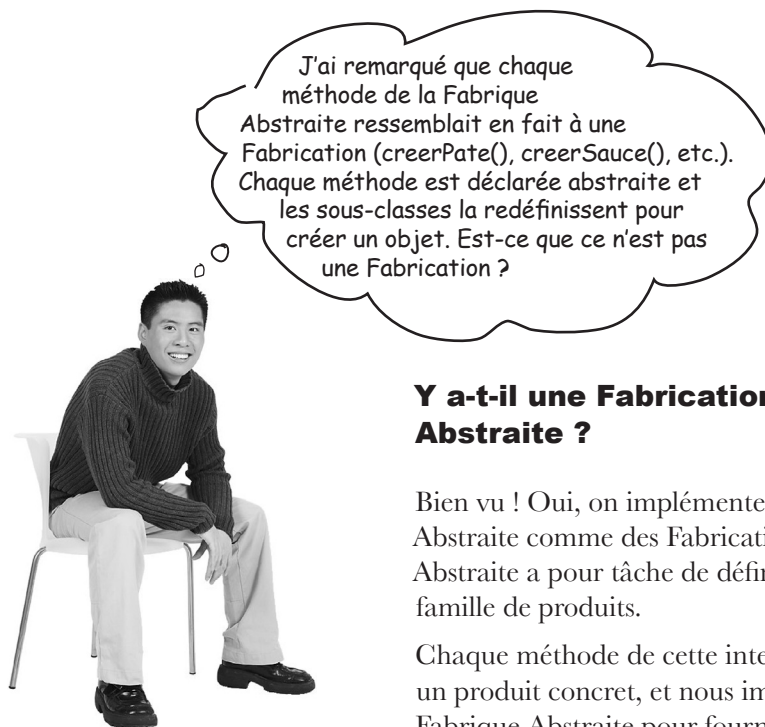
Le pattern Fabrique Abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes..

Nous avons vu que Fabrique Abstraite permet à un client d'utiliser une interface abstraite pour créer un ensemble de produits apparentés sans connaître les produits concrets qui sont réellement créés (ni même s'en soucier). Ainsi, le client est découplé de tous les détails des produits concrets. Observons le diagramme de classes pour voir la structure du pattern :



Voilà un diagramme de classes relativement compliqué. Rapportons-le à notre Pizzeria :





Y a-t-il une Fabrication tapie derrière la Fabrique Abstraite ?

Bien vu ! Oui, on implémente souvent les méthodes d'une Fabrique Abstraite comme des Fabrications. C'est logique, non ? La Fabrique Abstraite a pour tâche de définir une interface permettant de créer une famille de produits.

Chaque méthode de cette interface a la responsabilité de créer un produit concret, et nous implémentons une sous-classe de la Fabrique Abstraite pour fournir ces implémentations. C'est pourquoi les Fabrications constituent une façon naturelle d'implémenter vos méthodes de création des produits dans vos fabriques abstraites.



Interview

L'interview de cette semaine :
Fabrication et Fabrique Abstraite

DPTLP : Oaouh, un interview avec deux patterns à la fois ! C'est une première pour nous.

Fabrication : Ouais, je crois que je n'aime pas tellement être regroupé avec Fabrique Abstraite, vous savez. Ce n'est pas parce que nous sommes tous les deux des patterns de fabrication que je n'ai pas droit à mes propres interviews.

DPTLP : Ne vous fâchez pas. Nous voulions vous interviewer ensemble pour que nos lecteurs sachent qui est qui sans doute possible. Vous présentez des similitudes certaines et j'ai entendu dire que les gens vous confondent parfois.

Fabrique Abstraite : C'est vrai, il est arrivé qu'on me prenne pour Fabrication, et je sais que vous avez eu des problèmes semblables, Fabrication. Nous excellons tous les deux à découpler les applications des implémentations spécifiques, mais nous le faisons différemment. Donc je vois bien pourquoi les gens nous confondent parfois.

Fabrication : Eh bien, cela continue à m'abasourdir. Après tout, j'utilise des classes pour créer et vous utilisez des objets. C'est totalement différent !

DPTLP : Pouvez-vous nous en dire plus, Fabrication ?

Fabrication : Bien sûr. Fabrique Abstraite et moi créons tous les deux des objets – c'est notre travail. Mais moi, j'utilise l'héritage...

Fabrique Abstraite : ...et moi la composition.

Fabrication : Exactement. Cela veut dire que pour créer des objets en utilisant une Fabrication, vous devez étendre une classe et redéfinir une méthode de fabrication.

DPTLP : Et que fait cette méthode de fabrication ?

Fabrication : Elle crée des objets bien sûr ! Je veux dire que tout l'intérêt du Pattern Fabrication réside dans le fait que vous utilisez une sous-classe pour effectuer la création à votre place. De cette façon, les clients n'ont besoin de connaître que le type abstrait qu'ils utilisent, et la sous-classe s'occupe du type concret. Autrement dit, je permets de découper les clients des types concrets.

Fabrique Abstraite : Et moi aussi, seulement je procède autrement.

DPTLP : Continuez, Fabrique Abstraite... Vous avez dit quelque chose à propos de composition.

Fabrique Abstraite : Je fournis un type abstrait pour créer une famille de produits. Les sous-classes de ce type définissent le mode de création des produits. Pour utiliser la fabrique, vous en instanciez une et vous lui transmettez du code qui est écrit selon le type abstrait. Ainsi, comme pour Fabrication, mes clients sont découplés des produits concrets qu'ils utilisent.

DPTLP : Oh, Je vois. Un autre avantage est donc que vous regroupez des produits apparentés.

Fabrique Abstraite : C'est cela.

DPTLP : Que se passe-t-il si vous devez étendre cet ensemble de produits apparentés, par exemple pour en ajouter un ? Est-ce que cela ne vous oblige pas à modifier votre interface ?

Fabrique Abstraite : C'est vrai; mon interface doit changer quand on ajoute de nouveaux produits, et je sais que les gens n'aiment pas ça....

Fabrication : <ricanement>

Fabrique Abstraite : Qu'est-ce qui vous fait ricaner, Fabrication ?

Fabrication : Oh, allons, c'est toute une affaire ! Modifier votre interface signifie que vous devez modifier l'interface de chaque sous-classe ! Cela fait quand même beaucoup de travail.

Fabrique Abstraite : Oui, mais j'ai besoin d'une grosse interface parce que j'ai l'habitude de créer des familles entières de produits. Vous qui ne créez qu'un seul produit, vous n'avez pas vraiment besoin d'une grosse interface. Une seule méthode vous suffit.

DPTLP : Fabrique Abstraite, J'ai entendu dire que vous utilisiez des fabrications pour implémenter vos fabriques concrètes...

Fabrique Abstraite : Oui, je l'admets, mes fabriques concrètes implémentent une Fabrication pour créer leurs produits. Dans mon cas, elles sont utilisées uniquement pour créer des produits...

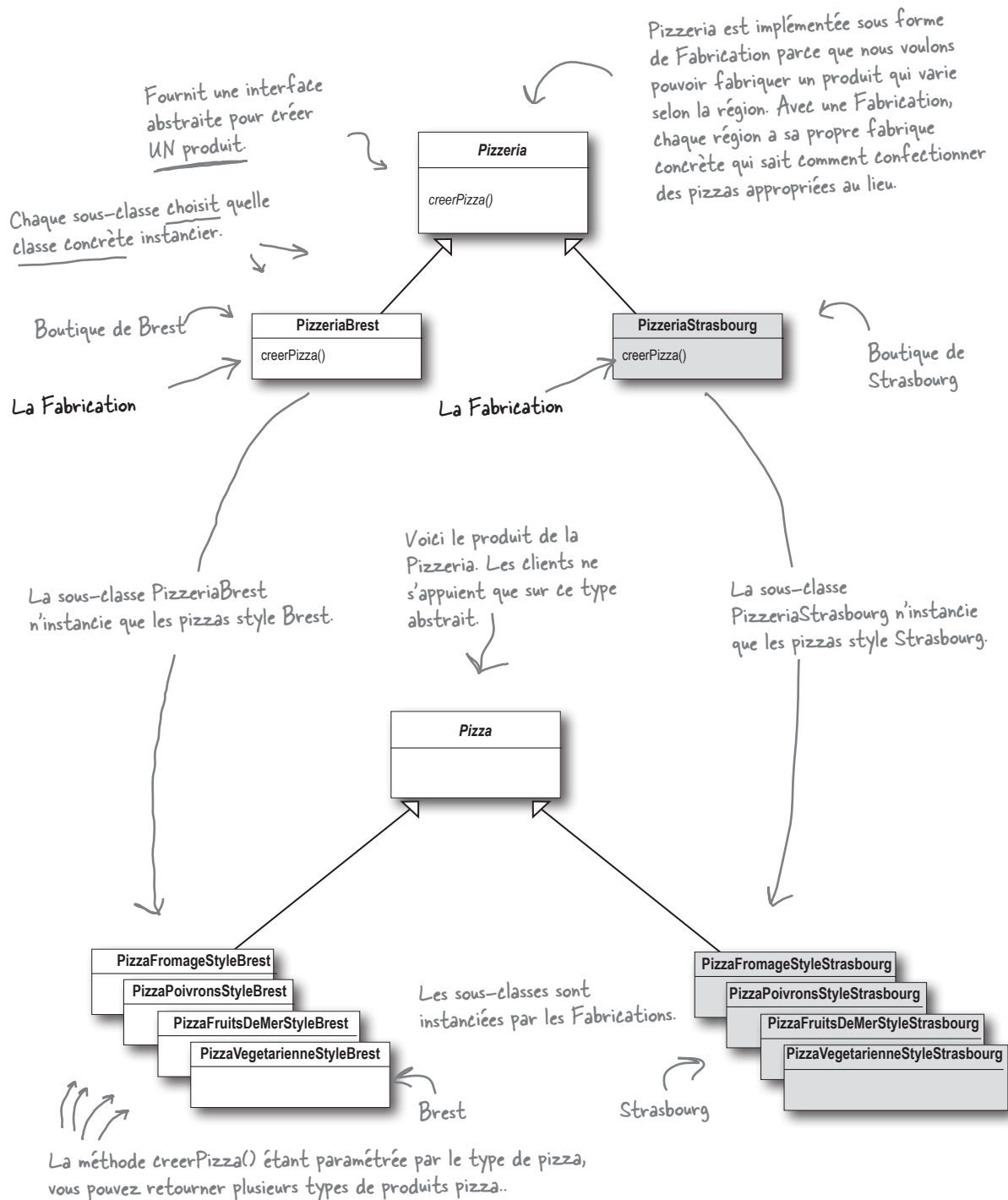
Fabrication : ...tandis que dans le mien, j'implémente généralement le code dans le créateur abstrait qui utilise les types concrets que les sous-classes créent.

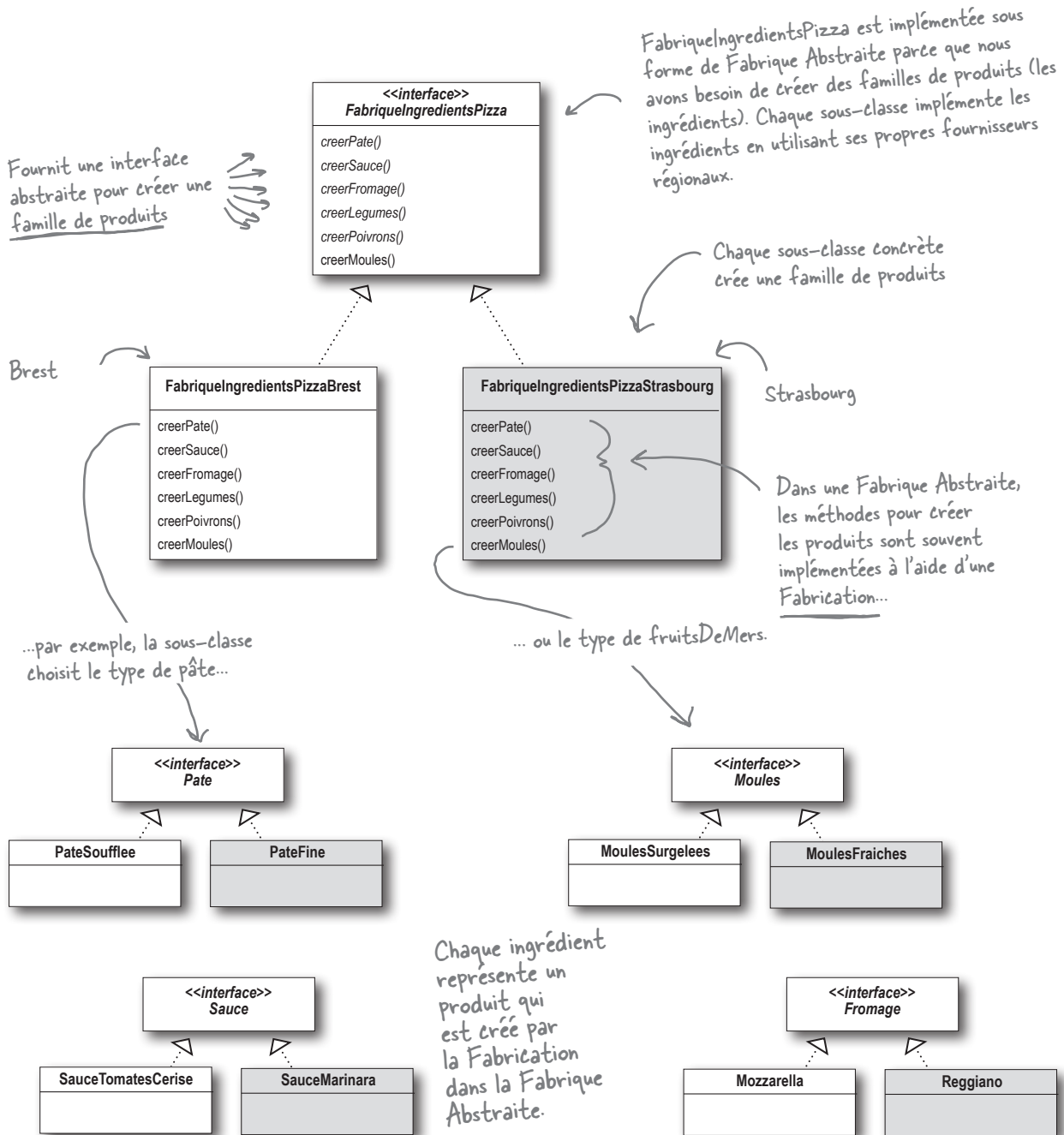
DPTLP : On dirait que vous êtes vraiment excellents dans ce que vous faites. Je suis sûr que les gens aiment avoir le choix. Après tout, les fabriques sont si utiles qu'ils voudront les utiliser dans toutes sortes de situations différentes. Vous encapsulez tous les deux la création des objets pour que les applications demeurent faiblement couplées et moins dépendantes des implémentations, ce qui est réellement formidable, que vous utilisiez une Fabrication ou une Fabrique Abstraite. Puis-je vous demander à chacun un mot d'adieu ?

Fabrique Abstraite : Merci. Souvenez-vous de moi, Fabrique Abstraite, et utilisez moi chaque fois que vous avez besoin de créer des familles de produits et que vous voulez garantir que vos clients créent des produits qui vont ensemble.

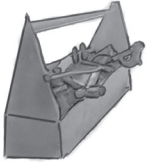
Fabrication : Et je suis Fabrication ; utilisez-moi pour découpler votre code client des classes concrètes que vous devez instancier, ou si vous ne connaissez pas d'avance toutes les classes concrètes dont vous allez avoir besoin. Pour m'utiliser, il suffit de me sous-classer et d'implémenter ma méthode de fabrication !

Fabrication et Fabrique Abstraite comparés





Les sous-classes produits créent des ensembles parallèles de familles de produits. Ici, nous avons une famille Brest et une famille Strasbourg.



Votre boîte à outils de concepteur

Dans ce chapitre, nous avons ajouté deux patterns de plus dans votre boîte à outils : **Fabrication** et **Fabrique Abstraite**. Ces deux patterns encapsulent la création des objets et vous permettent de découpler votre code des types concrets.

Principes OO

Encapsulez ce qui varie.
Préférez la composition à l'héritage.
Programmez des interfaces, non des implémentations.
Efforcez-vous de coupler faiblement les objets qui interagissent.
Les classes doivent être ouvertes à l'extension mais fermées à la modification..
Dépendez des abstractions. Ne dépendez pas des classes concrètes.

Bases de l'OO

abstraction
encapsulation
polymorphisme
héritage

Nous avons un nouveau principe qui nous conseille d'utiliser des abstractions quand c'est possible.

Patterns OO

Fabrique Abstraite - Fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes..

Fabrication - Définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Fabrication permet à une classe de déléguer l'instanciation à des sous-classes.

Ces nouveaux patterns encapsulent tous deux la création des objets et débouchent sur des conceptions plus découplées et plus souples.

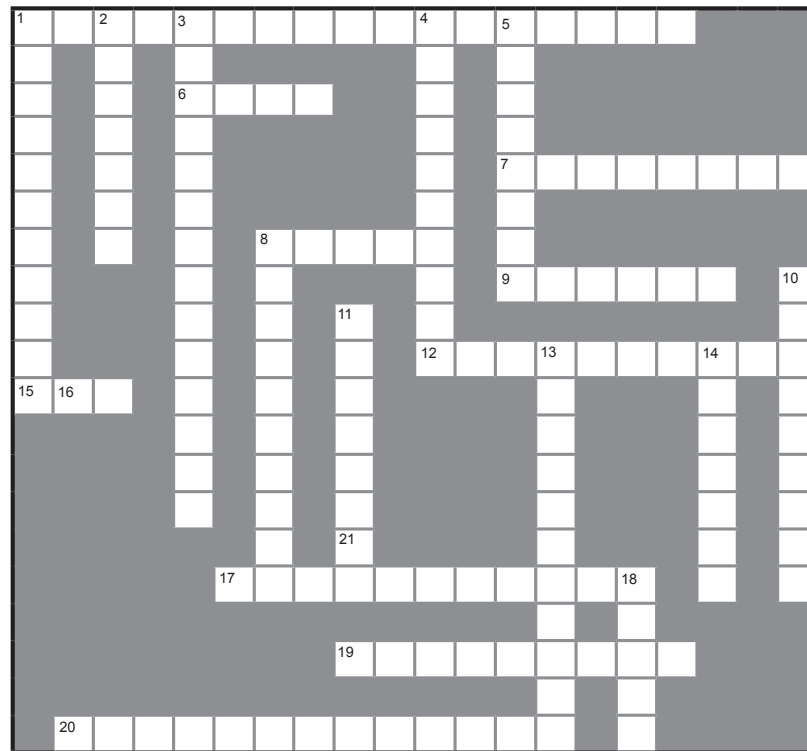


POINTS D'IMPACT

- Tous les patterns de fabrication encapsulent la création des objets.
- Fabrique Simple n'est pas un authentique design pattern, mais un moyen simple de découpler vos clients des classes concrètes.
- Fabrication s'appuie sur l'héritage : la création des objets est déléguée aux sous-classes qui implémentent la méthode de fabrication pour créer des objets.
- Fabrique Abstraite s'appuie sur la composition : la création des objets est implémentée dans les méthodes exposées dans l'interface fabrique.
- Tous les patterns de fabrique favorisent le couplage faible en réduisant la dépendance de votre application aux classes concrètes.
- L'intention de Fabrication est de permettre à une classe de déléguer l'instanciation à ses sous-classes.
- L'intention de Fabrique Abstraite est de créer des familles d'objets apparentés sans avoir à dépendre de leurs classes concrètes.
- Le Principe d'inversion des dépendances nous pousse à éviter de dépendre des types concrets et à nous appuyer sur des abstractions.
- Les fabriques constituent une technique puissante pour utiliser des abstractions, non des classes concrètes.



C'était un long chapitre. Prenez une part de pizza et relaxez-vous en faisant ces mots-croisés. Tous les mots de la solution sont dans ce chapitre.



Horizontalement

1. Permet de créer des familles d'objets apparentés (deux mots).
6. Peut être fine ou épaisse.
7. Objet.
8. Nom de l'abstraction dont dépendent les pizzas concrètes.
9. On en met sur certaines pizzas.
12. Tous les patterns de fabrication permettent d'_____ la création des objets.
15. Pas forcément idéal pour créer un objet.
16. La plupart des patterns de conception visent à les éviter.
18. Non spécialisé.
19. Peut être immédiate ou différée.

Verticalement

1. Délègue l'instanciation aux sous-classes.
2. Vrai ou faux.
3. Mise en œuvre.
4. Dans Fabrication, choisit la classe à instancier (mot composé).
5. Sorte de parmesan.
8. Qui peut prendre plusieurs formes.
10. À Strasbourg, les moules sont _____.
11. Il doit être aussi faible que possible.
13. Mieux vaut dépendre d'une _____ que d'une classe concrète.
14. Java en est un.
17. Marinara ou tomate.



Solutions des exercices

À vos crayons

Nous en avons fini avec la classe `PizzeriaBrest`. Encore deux et nous serons prêts à accorder des franchises !

Écrivez ici les implémentations de `PizzeriaStrasbourg` et de `PizzeriaMarseille` :

↪ Ces deux boutiques sont exactement comme celles de Brest...
sauf qu'elles créent des pizzas différentes

```
public class PizzeriaStrasbourg extends Pizzeria {
    protected Pizza creerPizza(String choix) {
        if (choix.equals("fromage")) {
            return new PizzaFromageStyleStrasbourg();
        } else if (choix.equals("vegetarienne")) {
            return new PizzaVegetarienneStyleStrasbourg();
        } else if (choix.equals("fruitsDeMer")) {
            return new PizzaFruitsDeMerStyleStrasbourg();
        } else if (choix.equals("poivrons")) {
            return new PizzaPoivronsStyleStrasbourg();
        } else return null;
    }
}
```

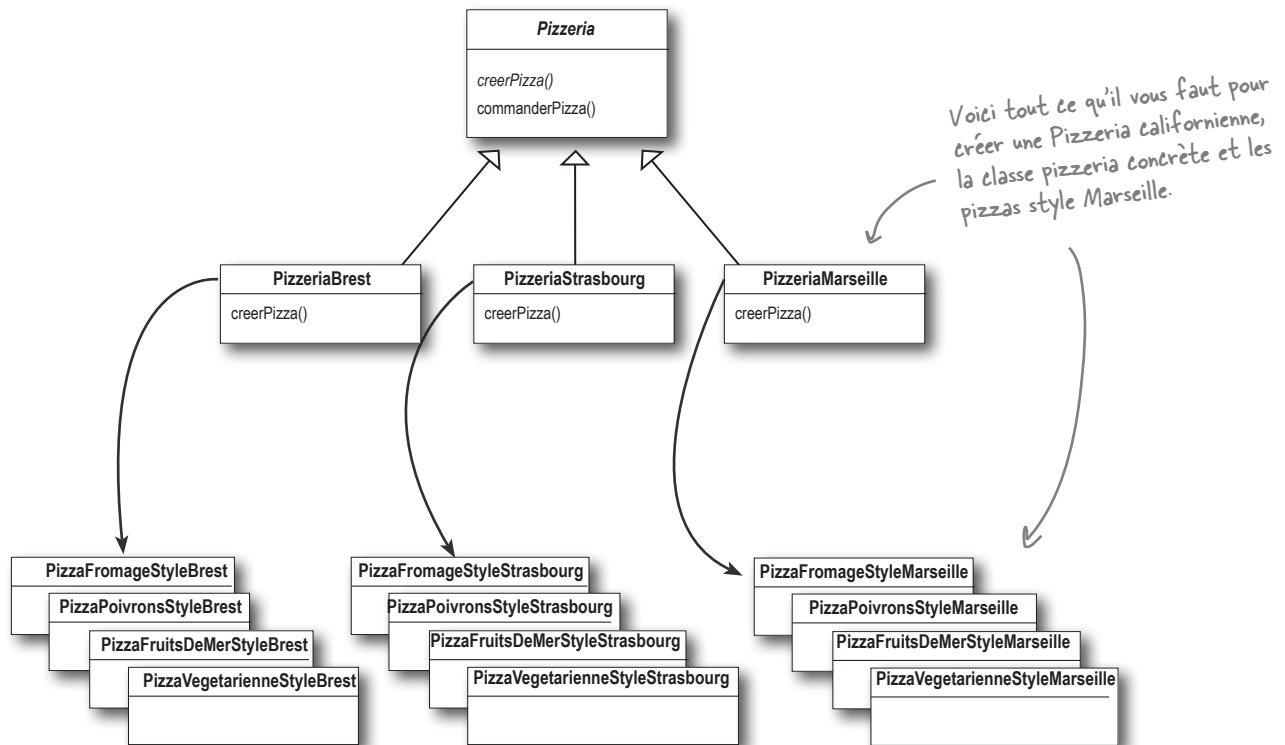
Ces deux
boutiques sont
exactement
comme celles
de Brest... sauf
qu'elles créent
des pizzas
différentes

```
public class PizzeriaMarseille extends Pizzeria {
    protected Pizza creerPizza(String choix) {
        if (choix.equals("fromage")) {
            return new PizzaFromageStyleMarseille();
        } else if (choix.equals("vegetarienne")) {
            return new PizzaVegetarienneStyleMarseille();
        } else if (choix.equals("fruitsDeMer")) {
            return new PizzaFruitsDeMerStyleMarseille();
        } else if (choix.equals("poivrons")) {
            return new PizzaPoivronsStyleMarseille();
        } else return null;
    }
}
```

et pour la
boutique de
Marseille, nous
créons des pizzas
style Marseille.

Solution du problème de conception

Il nous faut un autre type de pizza pour ces fous de Marseillais (fous dans le BON sens du terme bien sûr). Représentez l'autre ensemble de classes parallèles dont vous avez besoin pour ajouter une région marseillaise à notre Pizzeria.



Maintenant, inscrivez les cinq garnitures les plus bizarres que vous pouvez imaginer mettre sur une pizza. Ensuite, vous serez prêt à faire des affaires avec vos pizzas à Marseille !

Voici nos suggestions...

Purée d'ananas
Salami
Cœurs de palmier
Sardines
Amandes grillées

Une Pizzeria très dépendante



À vos crayons

Faisons comme si vous n'aviez jamais entendu parler de fabrication OO. Voici une version de la Pizzeria qui n'utilise pas de fabrication. Faites le compte du nombre d'objets concrets dont cette classe dépend. Si vous ajoutez des pizzas de style Marseille à cette Pizzeria, de combien d'objets dépendra-t-elle alors ?

```
public class PizzeriaDependante {

    public Pizza creerPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("Brest")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleBrest();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleBrest();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleBrest();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleBrest();
            }
        }
        else if (style.equals("Strasbourg")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleStrasbourg();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleStrasbourg();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleStrasbourg();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleStrasbourg();
            } } else {
                System.out.println("Erreur : type de pizza invalide");
                return null;
            }
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }
}
```

Gère toutes les pizzas
de style Brest

Gère toutes les pizzas
de style Strasbourg

Vous pouvez inscrire vos
réponses ici :

8

nombre

12

avec Marseille

À vos crayons

Écrivez la `FabriqueIngredientsPizzaStrasbourg`. Vous pouvez référencer les classes ci-dessous dans votre implémentation :

```
public class FabriqueIngredientsPizzaStrasbourg
    implements FabriqueIngredientsPizza
{
    public Pate creerPate() {
        return new PateSoufflee();
    }
    public Sauce creerSauce() {
        return new SauceTomatesCerise();
    }

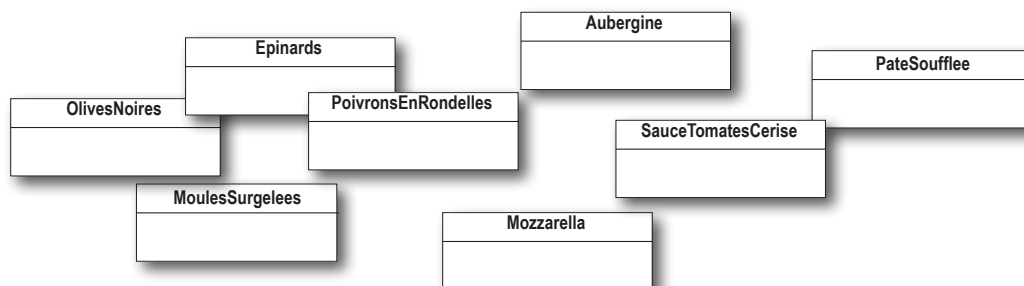
    public Fromage creerFromage() {
        return new Mozzarella();
    }

    public Legume[] creerLegumes() {
        Legume legume [] = { new OlivesNoires(),
                               new Epinards(),
                               new Aubergines() };

        return legume ;
    }

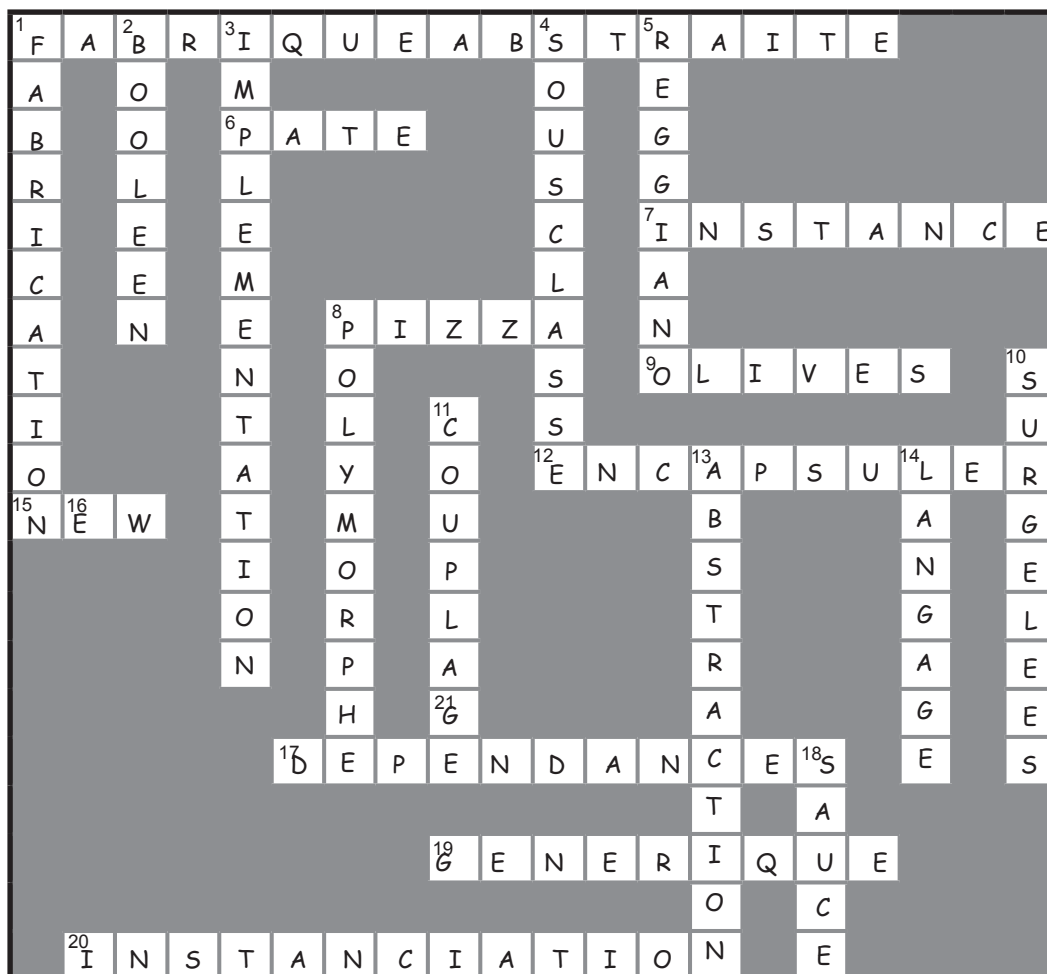
    public Poivrons creerPoivrons() {
        return new PoivronsEnRondelles();
    }

    public Moules creerMoules() {
        return new MoulesSurgelees();
    }
}
```





Solution des mots-croisés



5 le pattern Singleton

Des objets uniques en leur genre



Notre prochain arrêt est le Pattern Singleton, notre passeport pour la création d'objets uniques en leur genre dont il n'existe qu'une seule instance. Vous allez sûrement être ravi d'apprendre que, parmi tous les patterns, le Singleton est le plus simple en termes de diagramme de classes. En fait, ce diagramme ne contient qu'une seule classe ! Mais ne vous méprenez pas : malgré sa simplicité du point de vue de la conception des classes, nous allons rencontrer pas mal de bosses et de nids de poule dans son implémentation. Alors, bouclez vos ceintures.



Développeur : À quoi cela sert-il ?

Gourou : Il existe de nombreux objets dont il ne faut qu'un seul exemplaire : pools de threads, caches, boîtes de dialogue, objets qui gèrent des préférences et des paramètres de registre, objets utilisés pour la journalisation et objets qui servent de pilotes à des périphériques comme les imprimantes et les cartes graphiques. En réalité, pour bon nombre de ces types d'objets, des instanciations multiples provoqueraient toutes sortes de problèmes : le comportement des programmes serait incorrect, les ressources saturées ou les résultats incohérents.

Développeur : D'accord, il y a peut-être des classes qui ne doivent être instanciées qu'une fois, mais est-ce que cela justifie un chapitre entier ? Ne peut-on pas simplement appliquer une convention ou employer des variables globales ? En Java, je pourrais le faire avec une variable statique.

Gourou : À plus d'un titre, le Pattern Singleton est une **convention** qui garantit qu'un seul objet est instancié pour une classe donnée. Si vous en avez une meilleure, le monde entier aimerait la connaître. Mais souvenez-vous que tous les patterns sont des méthodes éprouvées, et celui-ci n'échappe pas à la règle. Outre le fait d'assurer qu'un seul objet sera créé, le Pattern Singleton nous fournit un point d'accès global, tout comme une variable globale, mais sans les inconvénients.

Développeur : Quels inconvénients ?

Gourou : Eh bien, prenons un exemple. Si vous affectez un objet à une variable globale, vous devez créer cet objet quand l'application est lancée*. D'accord ? Mais si l'objet consomme beaucoup de ressources et que votre application n'arrête pas de l'utiliser ? Comme vous allez le voir, le Pattern Singleton vous permet de ne créer vos objets que lorsqu'ils sont nécessaires.

Développeur : Cela n'a quand même pas l'air si difficile.

Gourou : Si vous maîtrisiez bien les méthodes et les variables de classe statiques ainsi que les modificateurs d'accès, ça ne l'est pas. Mais, dans tous les cas, il est intéressant de voir le fonctionnement d'un Singleton, et, malgré sa simplicité apparente, le code du Singleton est difficile à comprendre. Demandez-vous simplement : comment faire pour empêcher l'instanciation de plusieurs objets ? Ce n'est pas évident, n'est-ce pas ?

*Cela dépend en fait de l'implémentation. Certaines JVM créeront ces objets à la demande.

Le Petit Singleton

Petit exercice socratique dans le style du Petit Lispien

Comment créeriez-vous un seul objet ?

`new MonObjet();`

Et si un autre objet voulait créer un MonObjet ?
Pourrait-il de nouveau appeler new sur
MonObjet ?

Oui, bien sûr.

Tant que nous avons une classe, pouvons-nous
toujours l'instancier une ou plusieurs fois ?

Oui. Enfin, seulement si c'est une classe
publique.

Et sinon ?

Eh bien, si ce n'est pas une classe publique, seules
les classes du même package peuvent l'instancier.
Mais elles peuvent toujours l'instancier plusieurs
fois.

Mmm, intéressant...

Saviez-vous que vous pouviez écrire
ceci ?

```
public MaClasse {  
    private MaClasse() {}  
}
```

Non, je n'y avais jamais pensé, mais je suppose
que c'est possible puisque c'est une définition
légale.

Et cela signifie ?

Je suppose que c'est une classe qui ne peut pas
être instanciée parce qu'elle a un constructeur
privé.

Bien. Y a-t-il UN objet qui pourrait
utiliser le constructeur privé ?

Euh, je pense que le code de MaClasse est
le seul qui pourrait l'appeler. Mais cela n'a
pas beaucoup de sens.

Pourquoi ?

Parce qu'il me faudrait une instance de la classe pour l'appeler, mais je ne peux pas avoir d'instance parce qu'aucune autre classe ne peut la créer. C'est le problème de l'œuf et de la poule : je peux utiliser le constructeur depuis un objet de type `MaClasse`, mais je ne pourrai jamais instancier cet objet parce qu'aucun autre objet ne peut utiliser « `new MaClasse()` ».

OK. C'était juste une idée.

Qu'est-ce que cela signifie ?

```
public MaClasse {  
  
    public static MaClasse getInstance() {  
    }  
}
```

`MaClasse` est une classe qui a une méthode statique. Nous pouvons appeler la méthode statique comme ceci :

```
MaClasse.getInstance();
```

Pourquoi avez-vous utilisé `MaClasse` au lieu d'un nom d'objet ?

Eh bien, `getInstance()` est une méthode statique ; autrement dit une méthode de CLASSE. On doit utiliser le nom de la classe pour référencer une méthode statique.

Très intéressant. Et si nous résumions le tout ?
Maintenant, est-ce que je peux instancier `MaClasse` ?

Oui, bien sûr.

```
public MaClasse {  
  
    private MaClasse() {}  
  
    public static MaClasse getInstance() {  
        return new MaClasse();  
    }  
}
```

Et voyez vous une deuxième façon d'instancier un objet ?

```
MyClass.getInstance();
```

Pouvez-vous terminer le code de sorte qu'UNE SEULE instance de `MaClasse` soit jamais créée ?

Oui, je crois...

(Vous trouverez le code page suivante.)

Disséquons l'implémentation du Pattern



Regardez !

Si vous êtes en train de feuilleter ce livre, ne vous précipitez pas pour taper ce code. Nous verrons plus loin dans ce chapitre qu'il pose quelques problèmes.

```

public class Singleton {
    private static Singleton uniqueInstance;

    // autres variables d'instance

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new MaClasse();
        }
        return uniqueInstance;
    }

    // autres méthodes
}
  
```

Renommons MaClasse en Singleton

Nous avons une variable statique qui contient notre unique instance de la classe Singleton.

Notre constructeur est déclaré « private » : seul Singleton peut instancier cette classe !

La méthode getInstance() nous fournit un moyen d'instancier la classe et d'en retourner une instance.

Bien sûr, Singleton est une classe normale : elle possède d'autres variables d'instances et d'autres méthodes.



Code à la loupe

```

if (uniqueInstance == null) {
    uniqueInstance = new MaClasse();
}
return uniqueInstance;
  
```

uniqueInstance contient notre **UNIQUE** instance ; souvenez-vous que c'est une variable statique.

Si la valeur d'uniqueInstance est null, nous n'avons pas encore créé d'instance...

et, si elle n'existe pas, nous créons une instance de Singleton via son constructeur privé et nous l'affectons à uniqueInstance. Notez que si nous n'avons jamais besoin de l'instance, elle ne sera jamais créée ; c'est ce qu'on nomme instanciation à la demande.

Si uniqueInstance n'était pas null, c'est qu'elle avait été créée précédemment. Nous allons directement à l'instruction return.

Parvenus là, nous avons une instance et nous la retournons.



Interview

L'interview de cette semaine : Confessions d'un Singleton

DPTLP : Aujourd'hui, nous avons le plaisir de vous présenter un interview avec un objet Singleton. Pourquoi ne pas commencer par nous parler un peu de vous ?

Singleton : Eh bien, je suis totalement unique. Je n'existe qu'à un seul exemplaire !

DPTLP : Un seul ?

Singleton : Oui, un seul. Je suis basé sur le Pattern Singleton, qui garantit qu'à un moment donné il n'y a qu'une seule instance de moi.

DPTLP : N'est-ce pas un peu du gaspillage ? Quelqu'un a pris la peine de développer une classe à part entière et maintenant elle ne sert qu'à créer un seul objet ?

Singleton : Pas du tout ! Il y a de la puissance dans le UN. Disons que vous avez un objet qui contient des paramètres de registre. Vous ne voulez pas que plusieurs copies de cet objet et de ses valeurs courent çà et là – ce serait le chaos. En utilisant un objet comme moi, vous êtes sûr que chaque objet de votre application utilise la même ressource globale.

DPTLP : Dites nous-en plus...

Singleton : Oh, je suis bon à toutes sortes de choses. Être seul a parfois ses avantages, savez-vous. On m'emploie souvent pour gérer des pools de ressources, comme des pools de connexions ou de threads.

DPTLP : Pourtant, unique en votre genre... Vous devez vous sentir très seul.

Singleton : Comme je suis le seul dans mon genre, je suis toujours très occupé, mais ce serait bien si plus de développeurs me connaissaient. Beaucoup de programmeurs créent des bogues parce qu'ils ont plusieurs copies d'objets qui se promènent partout et ils ne s'en rendent même pas compte.

DPTLP : Mais, si je puis me permettre, comment savez-vous que vous êtes le seul ? Personne ne peut-il créer un « nouveau vous » avec un opérateur new ?

Singleton : Nan ! Je suis vraiment unique.

DPTLP : Est-ce que les développeurs feraient le serment de ne pas vous instancier plusieurs fois ?

Singleton : Bien sûr que non. Pour dire la vérité... eh bien, ceci prend un tour un peu trop personnel, mais, voyez-vous, je n'ai pas de constructeur public.

DPTLP : PAS DE CONSTRUCTEUR PUBLIC ! Oh, pardon, pas de constructeur public ?

Singleton : C'est cela. Mon constructeur est déclaré privé.

DPTLP : Mais comment est-ce possible ? Comment pouvez-vous même ÊTRE instancié ?

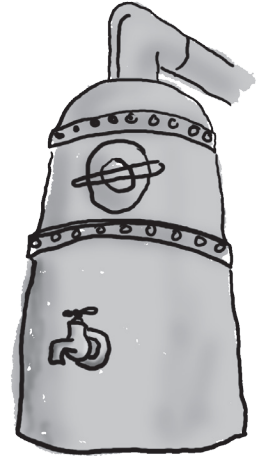
Singleton : Voyez-vous, pour obtenir un objet Singleton, vous ne l'instanciez pas, vous en demandez simplement une instance. Ma classe a donc une méthode statique nommée `getInstance()`. Appelez-la et je me montre immédiatement, prêt à me mettre au travail. En fait, je suis peut-être déjà en train d'aider d'autres objets quand vous faites appel à moi.

DPTLP : Eh bien, M. Singleton, malgré votre simplicité, cela ne doit pas être évident de faire tout ce travail. Merci de vous être confié à nous, et nous espérons vous revoir bientôt !!

La fabrique de chocolat

Chacun sait que toutes les fabriques de chocolat modernes ont des bouilleurs assistés par ordinateur. La tâche du bouilleur consiste à contenir un mélange de chocolat et de lait, à le porter à ébullition puis à le transmettre à la phase suivante où il est transformé en plaquettes de chocolat.

Voici la classe contrôleur du bouilleur industriel de Bonchoco, SA. Si vous étudiez le code, vous constatez qu'ils ont essayé très soigneusement d'éviter les catastrophes, par exemple de vider deux mille litres de mélange qui n'a pas bouilli, de remplir un bouilleur déjà plein ou de faire bouillir un bouilleur vide !



```
public class BouilleurChocolat {
    private boolean vide;
    private boolean bouilli;
    public BouilleurChocolat() {
        vide = true;
        bouilli = false;
    }
}
```

Ce code n'est exécuté que si le bouilleur est vide !

```
public void remplir() {
    if (estVide()) {
        vide = false;
        bouilli = false;
        // remplir le bouilleur du mélange lait/chocolat
    }
}

public void vider() {
    if (!estVide() && estBouilli()) {
        // vider le mélange
        vide = true;
    }
}
```

Pour pouvoir remplir le bouilleur, il doit être vide. Une fois qu'il est plein, nous positionnons les indicateurs « vide » et « bouilli ».

```
public void bouillir() {
    if (!estVide() && !estBouilli()) {
        // porter le contenu à ébullition
        bouilli = true;
    }
}
```

Pour pouvoir vider le bouilleur, il faut qu'il soit plein (non vide) ET qu'il ait bouilli. Une fois qu'il est vidé, nous repositionnons vide à « true ».

```
public boolean estVide() {
    return vide;
}

public boolean estBouilli() {
    return bouilli;
}
}
```

Pour pouvoir faire bouillir le mélange, il faut que le bouilleur soit plein et qu'il n'ait pas déjà bouilli. Une fois qu'il a bouilli, nous positionnons bouilli à « true ».



MUSCLEZ VOS NEURONES

Bonchoco a fait un travail honorable en essayant d'éviter les problèmes, n'est-ce pas votre avis ? Pourtant, vous soupçonnez probablement qu'en lâchant deux instances de BouilleurChocolat dans la nature, on s'expose à des catastrophes.

Que pourrait-il se passer si on créait plusieurs instances de BouilleurChocolat dans une application ?



À vos crayons

Pouvez-vous aider Bonchoco à améliorer sa classe BouilleurChocolat en la transformant en singleton ?

```
public class BouilleurChocolat {  
    private boolean vide;  
    private boolean bouilli;
```

```
    BouilleurChocolat() {  
        vide = true;  
        bouilli = false;
```

```
    public void remplir() {  
        if (estVide()) {  
            vide = false;  
            bouilli = false;  
            // remplir le bouilleur du mélange lait/chocolat  
        }  
    }  
    // reste du code de BouilleurChocolat...  
}
```


Le Pattern Singleton : définition

Maintenant que vous avez l'implémentation classique de Singleton en tête, il est temps de vous installer confortablement, de déguster une barre de chocolat et d'étudier de plus près les subtilités du Pattern Singleton.

Commençons par la définition concise du pattern :

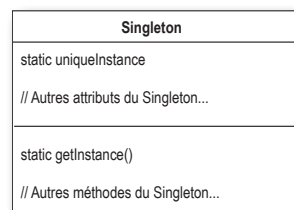
Le Pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Jusque là, pas de surprises. Mais décomposons un peu plus :

- Que se passe-t-il réellement ? Nous prenons une classe et nous lui faisons gérer une seule instance d'elle-même. Nous empêchons également toute autre classe d'en créer une autre instance. Pour obtenir une instance, vous devez passer par la classe elle-même.
- Nous fournissons également un point d'accès global à cette instance : chaque fois que vous en avez besoin, il suffit de demander à la classe et celle-ci vous retournera cette instance unique. Comme vous l'avez constaté, nous pouvons avoir une implémentation telle que le Singleton soit créé à la demande, ce qui est particulièrement important pour les objets gourmands en ressources.

Bien. Voyons le diagramme de classes :

La méthode `getInstance()` est statique. Ce qui signifie que c'est une méthode de classe. Vous pouvez donc accéder à cette méthode de n'importe quel point de votre code en utilisant `Singleton.getInstance()`. C'est tout aussi facile que d'accéder à une variable globale, mais le Singleton nous apporte d'autres avantages, comme la l'instanciation à la demande.



La variable de classe `uniqueInstance` contient notre seule et unique instance de Singleton.

Une classe mettant en œuvre le Pattern Singleton est plus qu'un Singleton : c'est une classe généraliste qui possède son propre ensemble d'attributs et de méthodes.

Allo le QG ?
~~Houston~~, Nous avons un problème...

On dirait que le bouilleur de chocolat nous a laissés tomber. Bien que nous ayons amélioré le code en utilisant un Singleton classique, la méthode remplir() de BouilleurChocolat s'est débrouillée pour commencer à remplir le bouilleur alors qu'un lot de chocolat et de lait y bouillait déjà ! Cela fait deux mille litres de lait (et de chocolat) gâchés ! Que s'est-il passé !?

Nous ne savons pas ce qui s'est passé ! Le nouveau code avec Singleton fonctionnait très bien. Notre seule idée, c'est que nous avons ajouté quelques optimisations au Contrôleur pour qu'il puisse utiliser plusieurs threads..



L'ajout de threads pourrait-il être la cause du problème ? N'est-il pas vrai qu'une fois que nous avons affecté à la variable d'instance uniqueInstance la seule instance de BouilleurChocolat, tous les appels de getInstance() devraient retourner la même instance ? D'accord ?

Vous êtes la JVM



Nous avons deux threads, chacun exécutant ce code. Votre tâche consiste à jouer le rôle de la JVM et de

déterminer s'il existe un cas dans lequel deux threads pourraient s'emparer de différents objets bouilleur. Indication : Il

suffit d'observer la séquence

d'opérations de la méthode `getInstance()` et la valeur d'`uniqueInstance` pour voir les chevauchements possibles. Utilisez les fragments de code pour comprendre comment le code peut s'intercaler pour créer deux objets bouilleur.

```
BouilleurChocolat bouilleur =
    BouilleurChocolat.getInstance();
remplir();
bouillir();
vider();
```

```
public static BouilleurChocolat
getInstance() {
```

```
if (uniqueInstance == null) {
```

```
uniqueInstance =
    new BouilleurChocolat();
```

}

```
return uniqueInstance;
```

}

**Vérifiez bien votre réponse page 188
avant de tourner la page !**

	Thread Un	Thread Deux	Valeur de uniqueInstance
<pre> instance = singletonChocolat(); </pre>			
<pre> Instance; </pre>			

Gérer le multithread

Nos malheurs avec le multithread sont facilement éliminés si nous faisons de getInstance() une méthode synchronisée :

```
public class Singleton {
    private static Singleton uniqueInstance;

    // autres variables d'instance

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // autres méthodes
}
```

En ajoutant le mot-clé `synchronized` à `getInstance()`, nous obligeons chaque thread à attendre son tour avant d'entrer dans la méthode. Autrement dit, deux threads ne peuvent pas entrer dans la méthode en même temps.

Je suis d'accord que cela règle le problème. Mais la synchronisation est coûteuse. Est-ce que ce n'est pas un autre problème ?

Oo

Bien vu, et c'est même pratiquement pire qu'avant : la seule fois que la synchronisation sert à quelque chose, c'est lors du premier passage dans la méthode. Autrement dit, une fois que nous avons affecté à la variable `uniqueInstance` une instance de `Singleton`, nous n'avons plus besoin de synchroniser cette méthode. Après la première passe, la synchronisation est une surcharge totalement inutile !



Peut-on améliorer le multithread ?

Pour la plupart des applications Java, nous devons de toute évidence garantir que le Singleton fonctionne en présence de plusieurs threads. Mais la synchronisation de la méthode `getInstance()` semble passablement coûteuse. Que faire alors ?

Plusieurs options se présentent à nous...

1. Ne rien faire si la performance `getInstance()` n'est pas vitale pour votre application

C'est vrai. Si l'appel de la méthode `getInstance()` ne cause pas de surcharge substantielle à votre application, n'y pensez plus. Synchroniser `getInstance()` est simple et efficace. Mais n'oubliez pas qu'une méthode synchronisée peut diminuer les performances par un facteur de 100. Si une partie « à fort trafic » de votre code commence à utiliser `getInstance()`, vous devrez peut-être reconsidérer la question.

2. Préférer une instance créée au démarrage à une instance créée à la demande

Si votre application crée et utilise toujours une instance du Singleton ou si les aspects de surcharge de la création et de l'exécution du Singleton sont trop onéreux, vous pouvez préférer le créer au démarrage, comme ceci :

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Allez de l'avant et créez une instance du Singleton un initialiseur statique. Ce code est garanti sans problème par rapport aux threads!

Comme nous avons déjà une instance, il nous suffit de la retourner.

En adoptant cette approche, nous nous reposons sur la JVM pour créer l'unique instance du Singleton quand la classe est chargée. La JVM garantit que l'instance sera créée avant qu'un thread quelconque n'accède à la variable statique `uniqueInstance`.

3. Utilisez le « verrouillage à double vérification » pour réduire l'usage de la synchronisation de getInstance()

Avec le verrouillage à double vérification, nous commençons par vérifier si une instance est créée, et si non, ALORS nous synchronisons. De cette façon, nous ne synchronisons que la première fois : exactement ce que nous voulons.

Examinons le code :

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Chercher une instance. S'il n'y en a pas, entrer dans un bloc synchronisé.

Notez que nous ne synchronisons que le premier passage !

Une fois dans le bloc, vérifier de nouveau. Si la valeur de la variable est toujours null, créer une instance.

* Le mot-clé volatile garantit que plusieurs threads gèrent correctement la variable uniqueInstance quand elle est initialisée à une instance de Singleton.

Si l'utilisation de la méthode getInstance() pose un problème de performances, cette façon d'implémenter le Singleton peut réduire la surcharge de manière spectaculaire.



Regardez ! Le verrouillage à double vérification ne fonctionne qu'à partir de Java 5 !

Malheureusement, jusqu'à la version 1.4, de nombreuses JVM contiennent des implémentations du mot-clé volatile permettant une synchronisation incorrecte avec le verrouillage à double vérification. Si vous devez utiliser une autre JVM que Java 5, envisagez d'autres méthodes pour écrire votre Singleton.

Pendant ce temps, à la chocolaterie...

Pendant que nous étions en train de diagnostiquer les problèmes du multithread, le bouilleur a été nettoyé et il est prêt à repartir. Mais d'abord, nous devons résoudre ces problèmes de multithread. Nous disposons de plusieurs solutions, chacune ayant ses avantages et ses inconvénients. Laquelle allons-nous employer ?



À vos crayons

Pour chaque solution, décrivez son applicabilité au problème du code de BouilleurChocolat :

Synchroniser la méthode getInstance() :

Utilisation de l'instanciation au démarrage :

Verrouillage à double vérification :

Félicitations !

À ce stade, la Fabrique de chocolat est un client heureux et Bonchoco est content que nous ayons appliqué notre expertise au code du bouilleur. Quelle que soit la solution que vous ayez choisie pour résoudre le problème de multithread, le bouilleur doit maintenant fonctionner sans autres mésaventures. Félicitations. Non seulement vous avez réussi à échapper à deux tonnes et quelques de chocolat bouillant dans ce chapitre, mais vous avez également vu tous les problèmes potentiels du Singleton.

Il n'y a pas de questions stupides

Q: Pour un pattern si simple qui ne se compose que d'une seule classe, ce Singleton semble avoir bien des problèmes.

R: Bon, nous vous avons averti ! Mais ne laissez pas ces problèmes vous décourager. Même si les Singletons peuvent être délicats à implémenter correctement, la lecture de ce chapitre vous aura permis d'être bien informés sur les techniques possibles et vous saurez en utiliser chaque fois que vous aurez besoin de contrôler le nombre d'instances que vous créez.

Q: Est-ce que je ne peux pas me contenter de créer une classe dont toutes les méthodes et les variables sont statiques ? Ne serait-ce pas identique à un Singleton ?

R: Oui, si votre classe est autonome et ne dépend pas d'une initialisation complexe. Toutefois, en raison de la façon dont Java gère les initialisations statiques, cela peut donner un vrai gâchis, surtout si plusieurs classes sont impliquées. Ce scénario aboutit souvent à des bogues subtils et difficiles à détecter, qui sont liés à l'ordre des initialisations. À moins qu'il n'y ait une raison impérative d'implémenter votre « singleton » de cette façon, il est de beaucoup préférable de rester dans le monde objet.

Q: Et les chargeurs de classes ? J'ai entendu dire qu'il y avait un risque que deux chargeurs de classes finissent par avoir chacun leur propre instance de Singleton.

R: Oui, c'est vrai, puisque chaque chargeur de classe définit un espace de nommage. Si vous avez deux chargeurs de classes ou plus, vous pouvez charger la même classe plusieurs fois (une par chargeur). Maintenant, si cette classe s'avère être un Singleton, comme nous avons plusieurs versions de la classe, nous avons également plusieurs instances du Singleton. Faites donc bien attention si vous utilisez plusieurs chargeurs de classes et des Singletons. Une façon de contourner ce problème est de spécifier le chargeur de classe vous-même.



Relax

Les rumeurs selon lesquelles des Singletons auraient été dévorés par des ramasse-miettes sont tout à fait exagérées

Avant Java 1.2, un bogue permettait au ramasse-miettes de collecter prématurément les Singletons s'ils n'avaient pas de référence globale. Autrement dit, vous pouviez créer un Singleton, et, si la seule référence au Singleton était dans le Singleton lui-même, celui-ci était collecté et détruit par le ramasse-miettes. Cela provoquait des bogues complexes : après le « ramassage » du Singleton, l'appel de `getInstance()` suivant produisait un Singleton flamant neuf. Dans de nombreuses applications, cela provoquait des comportements étranges, comme des états retrouvant mystérieusement leur valeur initiale ou des connexions réseau réinitialisées.

Ce bogue a été corrigé depuis Java 1.2, et les références globales ne sont plus requises. Si, pour une raison quelconque, vous utilisez encore une JVM antérieure à Java 1.2 JVM, sachez que ce problème existe. Sinon, vous pouvez dormir sur vos deux oreilles, vos Singletons ne seront pas prématurément détruits.

Q: On m'a toujours appris qu'une classe devait faire une chose et une seule, et qu'une classe faisant deux choses était considérée comme une mauvaise conception OO. Est-ce que Singleton n'est pas en contradiction avec cette optique ?

R: C'est ce qu'on appelle le principe « Une classe, une responsabilité ». Oui, c'est vrai, le Singleton n'est pas seulement responsable de la gestion de son unique instance (et de la fourniture d'un point d'accès global), il doit également assurer le rôle principal qu'il joue dans votre application. En ce sens, on peut certainement soutenir qu'il assume deux responsabilités. Néanmoins, il n'est pas difficile de voir qu'il est utile qu'une classe gère sa propre instance et que cela simplifie à coup sûr la conception globale. En outre, beaucoup de développeurs connaissent bien le pattern Singleton, car il est largement utilisé. Cela dit, certains éprouvent le besoin d'extraire les fonctionnalités du Singleton.

Q: Je voulais sous-classer mon Singleton, mais j'ai eu des problèmes. Est-ce qu'on a le droit de sous-classer un Singleton ?

R: L'un des problèmes du sous-classement d'un Singleton est que son constructeur est privé. On ne peut pas étendre une classe dont le constructeur est privé. Vous devez donc commencer par modifier le constructeur pour qu'il soit public ou protégé. Mais alors, ce n'est plus vraiment un Singleton, puisque d'autres classes peuvent l'instancier.

Si vous décidez néanmoins de modifier le constructeur, il y a un autre problème. L'implémentation du Singleton est basée sur une variable statique : si vous sous-classez directement, toutes vos classes dérivées partageront la même variable d'instance. Ce n'est probablement pas ce que vous aviez en tête. Pour que le sous-classement fonctionne, il est donc nécessaire d'implémenter une sorte de registre dans la classe de base.

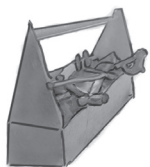
Avant de mettre en œuvre un tel schéma, demandez-vous ce que vous gagnez réellement à sous-classer un Singleton. Comme la plupart des patterns, le Singleton n'est pas nécessairement censé être une solution qui peut s'insérer dans une bibliothèque. De plus, le code du Singleton est facile à ajouter à n'importe quelle classe existante.

Enfin, si vous utilisez un grand nombre de Singletons dans votre application, vous devez examiner de très près votre conception. Les Singletons sont faits pour être consommés avec modération.

Q: Je ne comprends toujours pas vraiment pourquoi les variables globales sont pires qu'un Singleton.

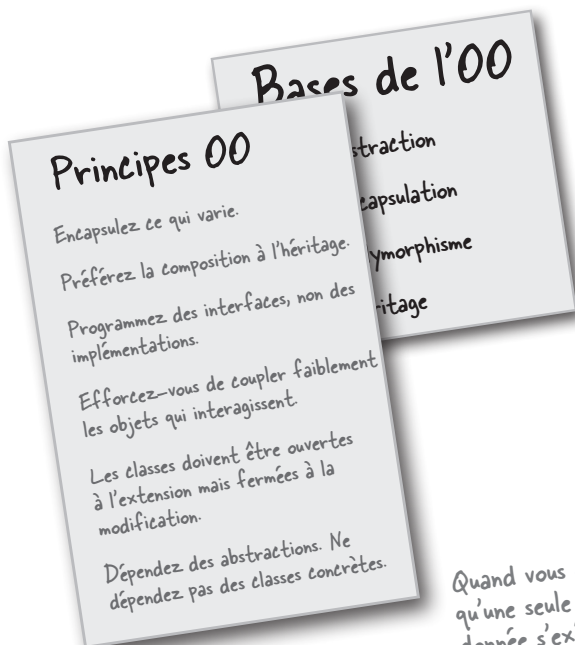
R: En Java, les variables globales sont essentiellement des références statiques à des objets.

Utiliser des variables globales de cette manière présente un certain nombre d'inconvénients. Nous en avons déjà mentionné un : la question de l'instanciation à la demande contre l'instanciation au démarrage. Mais n'oublions pas l'intention du pattern : garantir qu'une classe n'a qu'une seule instance et fournir à celle-ci un point d'accès global. Une variable globale satisfait à la seconde proposition, mais pas à la première. Les variables globales ont également tendance à encourager les développeurs à polluer l'espace de nommage avec une foule de références globales à de petits objets. Ce n'est pas le cas des Singletons, bien qu'on puisse en abuser.



Votre boîte à outils de concepteur

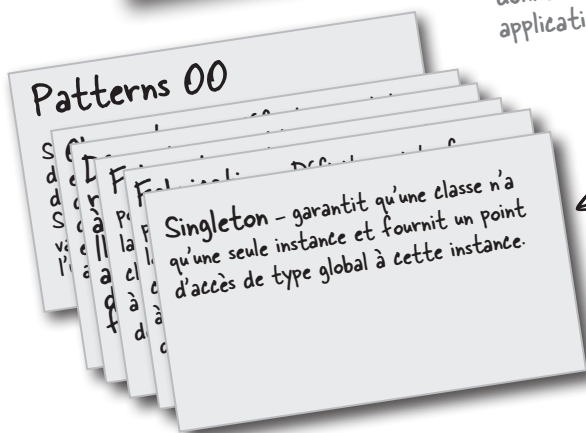
Vous venez d'ajouter un nouveau pattern à votre boîte à outils. Singleton met à votre disposition une autre méthode pour créer des objets, en l'occurrence des objets uniques.



Bases de l'OO

abstraction
encapsulation
polymorphisme
héritage

Quand vous devez garantir qu'une seule instance d'une classe donnée s'exécute dans votre application, adoptez le Singleton.



Comme vous l'avez vu, malgré sa simplicité apparente, l'implémentation de Singleton comporte de nombreux détails. Mais après avoir lu ce chapitre, vous serez prêt à appliquer Singleton, même en plein désert.

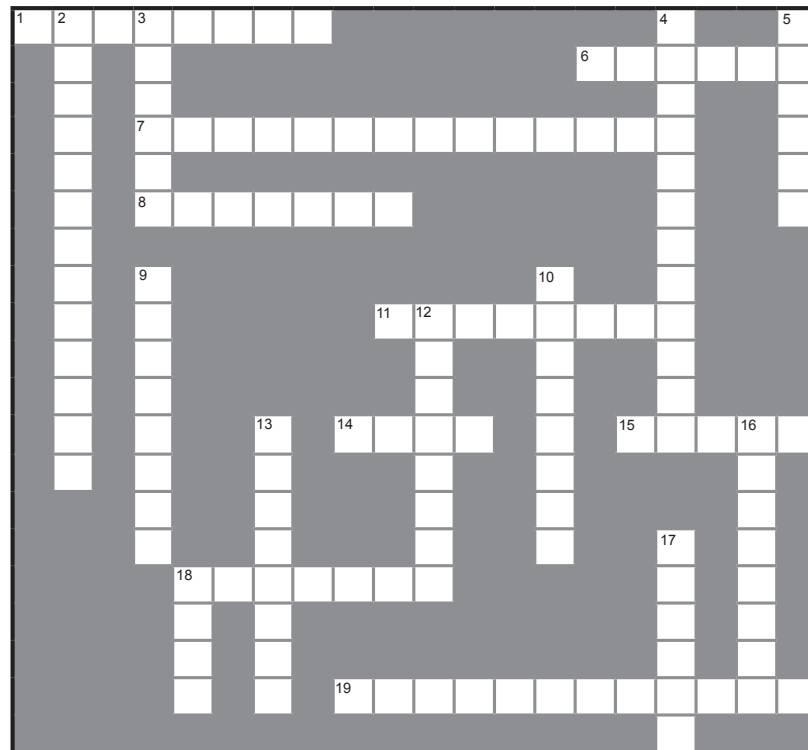


POINTS D'IMPACT

- Le Pattern garantit qu'il existe au plus une instance d'une classe donnée dans une application.
- Le Pattern Singleton fournit également un point d'accès global à cette instance.
- Java implémente le Pattern Singleton en utilisant un constructeur privé, une méthode statique combinée à une variable statique.
- Examinez vos contraintes de performances et de ressources et choisissez soigneusement une implémentation de Singleton appropriée pour les applications multithread (et il faut considérer que toutes les applications sont multithread !).
- Attention au verrouillage à double vérification. Il n'est pas fiable en cas de threads multiples dans les versions antérieures à Java 2, version 5.
- Soyez prudent si vous utilisez plusieurs chargeurs de classes. Cela pourrait mettre en échec l'implémentation du Singleton et créer plusieurs instances de la même classe.
- Si vous travaillez avec une JVM antérieure à 1.2, vous devrez créer un registre de Singletons pour battre de vitesse le ramasse-miettes.



Installez-vous confortablement, ouvrez cette boîte de chocolats qu'on vous a envoyée pour avoir résolu le problème du multithread, et accordez vous un moment de loisir avec ces petits mots-croisés. Tous les mots sont dans ce chapitre.



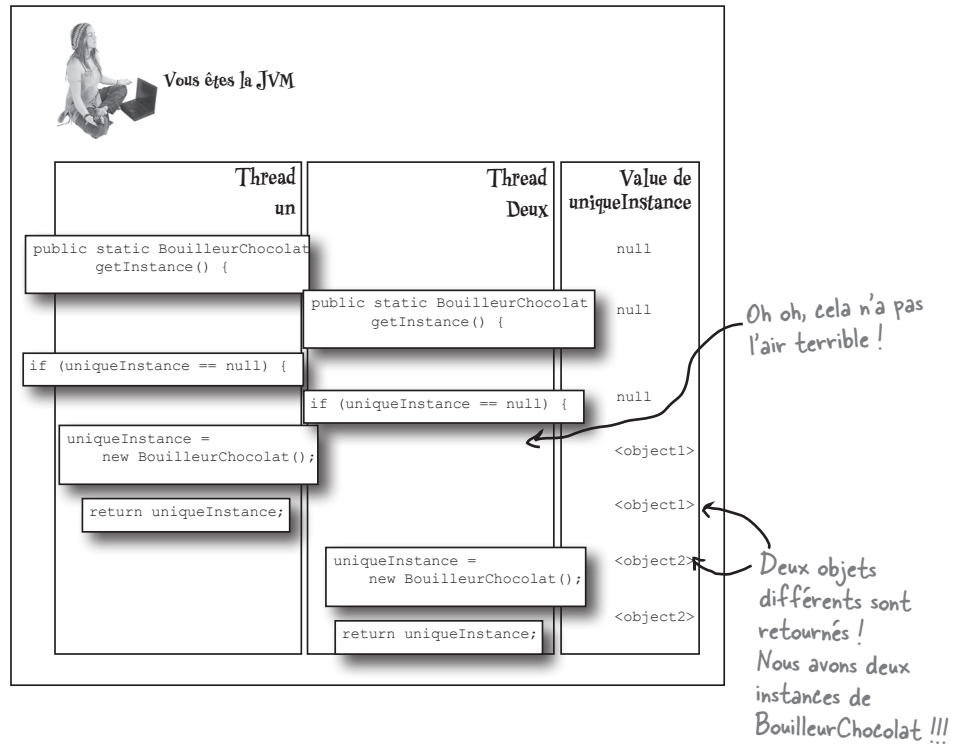
Horizontalement

1. Garantit qu'une classe n'a qu'une seule instance.
6. Le point d'accès fourni par le Singleton est _____.
7. Selon certaines rumeurs, ils leur arrivent de manger des Singletons (mot composé).
8. Certains objets doivent obligatoirement être _____.
11. Peut être noir ou au lait.
14. Mélangé au chocolat.
15. Le constructeur du Singleton est _____.
18. Page 169, elle est unique en son genre.
19. Une solution possible au problème du multithread.

Verticalement

2. Création d'objet.
3. Maître à penser.
4. Le Singleton est embarrassé : il n'a pas de _____ public.
5. Le Singleton n'en a qu'une.
9. Le Singleton assure qu'il n'en existe qu'une.
10. Le fabricant de chocolat de ce chapitre.
12. L'un des principes OO.
13. getInstance() est une méthode _____.
16. Nouveau mot-clé de Java 5, réservé jusqu'ici.
17. N'est donc pas privé.
18. Condition préalable au remplissage.

Solutions des exercices



À vos crayons

Pouvez-vous aider Bonchoco à améliorer sa classe BouilleurChocolat en la transformant en singleton ?

```
public class BouilleurChocolat {
    private boolean vide;
    private boolean bouilli;

    private static BouilleurChocolat uniqueInstance;

    private BouilleurChocolat() {
        empty = true;
        boiled = false;
    }

    public static BouilleurChocolat getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new BouilleurChocolat();
        }
        return uniqueInstance;
    }

    public void remplir() {
        if (estVide()) {
            vide = false;
            bouilli = false;
            // remplir le bouilleur d'un mélange lait/chocolat
        }
    }
    // reste du code de BouilleurChocolat ...
}
```


Solutions des exercices

À vos crayons

Pour chaque solution, décrivez son applicabilité au problème du code de BouilleurChocolat :

Synchroniser la méthode getInstance() :

Technique simple dont le fonctionnement est garanti. Comme il semble que nous n'ayons aucune
contrainte de performance avec le bouilleur, ce serait un bon choix.

Utiliser l'instanciation au démarrage :

Comme nous allons toujours instancier le bouilleur dans notre code, l'initialisation statique de
l'instance ne poserait pas de problème. Cette solution fonctionnerait aussi bien que la méthode
synchronisée, bien qu'un peu moins évidente pour un développeur familier du pattern standard.

Utiliser le verrouillage à double vérification :

Comme nous n'avons pas de contraintes de performance, le verrouillage à double vérification semble
excessif. En outre, nous devons utiliser au moins Java 5.



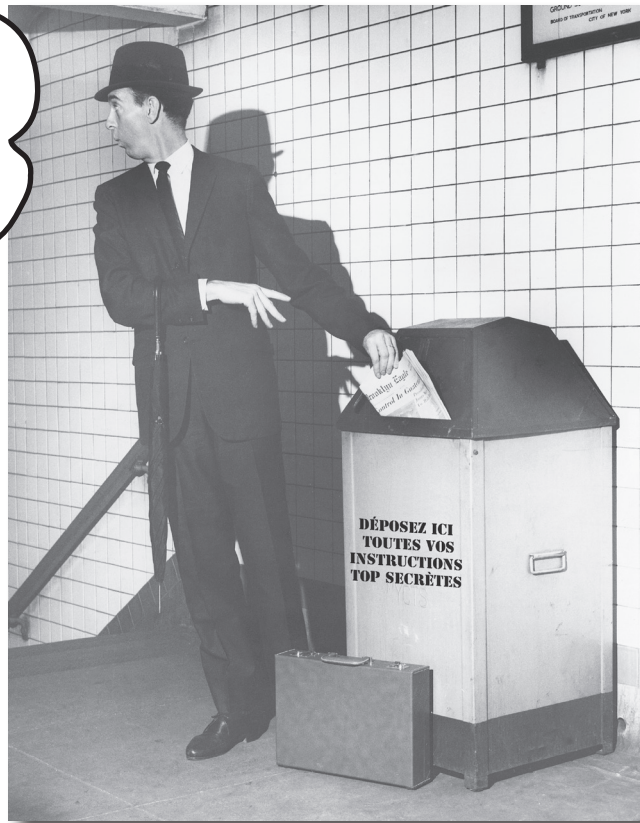
Solutions des mots-croisés

1S	2I	N	3G	L	E	T	O	N			4C			5C					
	N									6G	L	O	B	A	L				
	S										N				A				
	T		7R	A	M	A	S	S	E	M	I	E	T	T	E	S			
	A											T				S			
	N		8U	N	I	Q	U	E	S			R				E			
	C											U							
	I		9I								10B					C			
	A		N					11C	12H	O	C	O	L	A	T				
	T		S						E			O				E			
	I		T						R			C				U			
	O		A		13S		14L	A	I	T		H		15P	R	I	16V	E	
	N		N		T				T			O					O		
			C		A				A			C					L		
			E		T				G			O			17P		A		
		18V	O	I	T	U	R	E							U		T		
		I		Q											B		I		
		D		U											L		L		
		E		E				19S	Y	N	C	H	R	O	N	I	S	E	R
																C			

6 le pattern Commande

✦ ***Encapsuler l'invocation*** ✦

Ces boîtes aux lettres pour documents top secrets ont révolutionné l'industrie de l'espionnage. Je n'ai qu'à déposer ma requête et des gens disparaissent, des gouvernements sont renversés du jour au lendemain, et en plus elles servent même de dépôt de pressing. Inutile de se demander où, quand et comment ! Ça marche, tout simplement !



Dans ce chapitre, nous allons envisager l'encapsulation à un tout autre niveau : nous allons encapsuler l'invocation des méthodes.

Oui, en encapsulant les appels de méthodes, nous allons pouvoir « cristalliser » des traitements afin que l'objet qui les invoque n'ait pas besoin de savoir comment il sont exécutés : il suffit qu'il utilise nos méthodes cristallisées pour obtenir un résultat. Nous pourrions même faire des choses drôlement futées avec ces appels de méthodes encapsulés, par exemple les sauvegarder pour la journalisation ou les réutiliser pour implémenter des annulations dans notre code.



Maisons de rêve, SÀRL.
1221, avenue de l'Industrie, Bureau 2000
99999 Futurville

Bonjour,

J'ai récemment reçu de Jean-Loup Ragan, PDG de MétéoExpress, une documentation et une version de démonstration de leur nouvelle station météo extensible. Je dois dire que j'ai été si impressionné par la qualité de votre architecture logicielle que j'aimerais de vous demander de concevoir une API pour notre nouveau système de programmation d'équipements ménager à distance. Pour vous remercier de vos services, nous serons heureux de vous offrir des stock options gratuites de notre société.

Vous trouverez ci-joint pour information un prototype de notre télécommande révolutionnaire. Elle dispose de sept emplacements programmables, chacun d'eux pouvant être affecté à un appareil différent et possédant son propre interrupteur. La commande est également équipée d'un bouton Annulation global.

Je joins également sur CD-ROM un ensemble de classes Java qui ont été créées par différents fournisseurs pour contrôler des équipements domotiques : éclairages, ventilateurs, jacuzzis, équipements audio et autres appareils contrôlables similaires. Nous aimerions que vous écriviez une API pour programmer la télécommande, de sorte que chaque emplacement puisse être affecté au contrôle d'un équipement ou d'un ensemble d'équipements. Notez qu'il est important que nous puissions commander les appareils figurant actuellement sur le disque, mais aussi tous ceux que les fournisseurs pourraient proposer à l'avenir.

Étant donné le travail que vous avez effectué sur la station de MétéoExpress, nous sommes convaincus que notre télécommande sera une réussite !

Dans l'attente de recevoir votre conception.

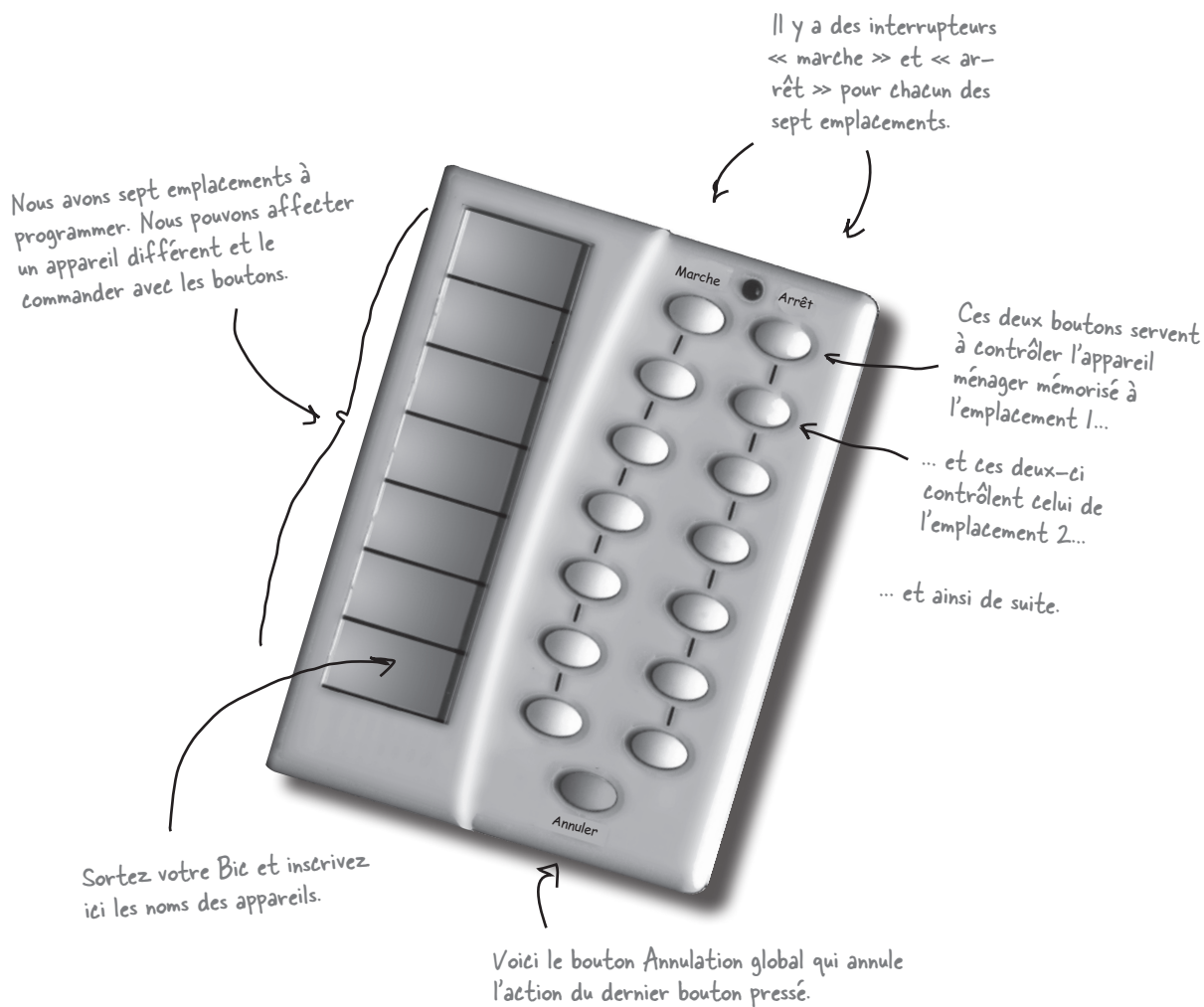
Cordialement,

X. Disse, PDG

Bill "X-10" Thompson, CEO



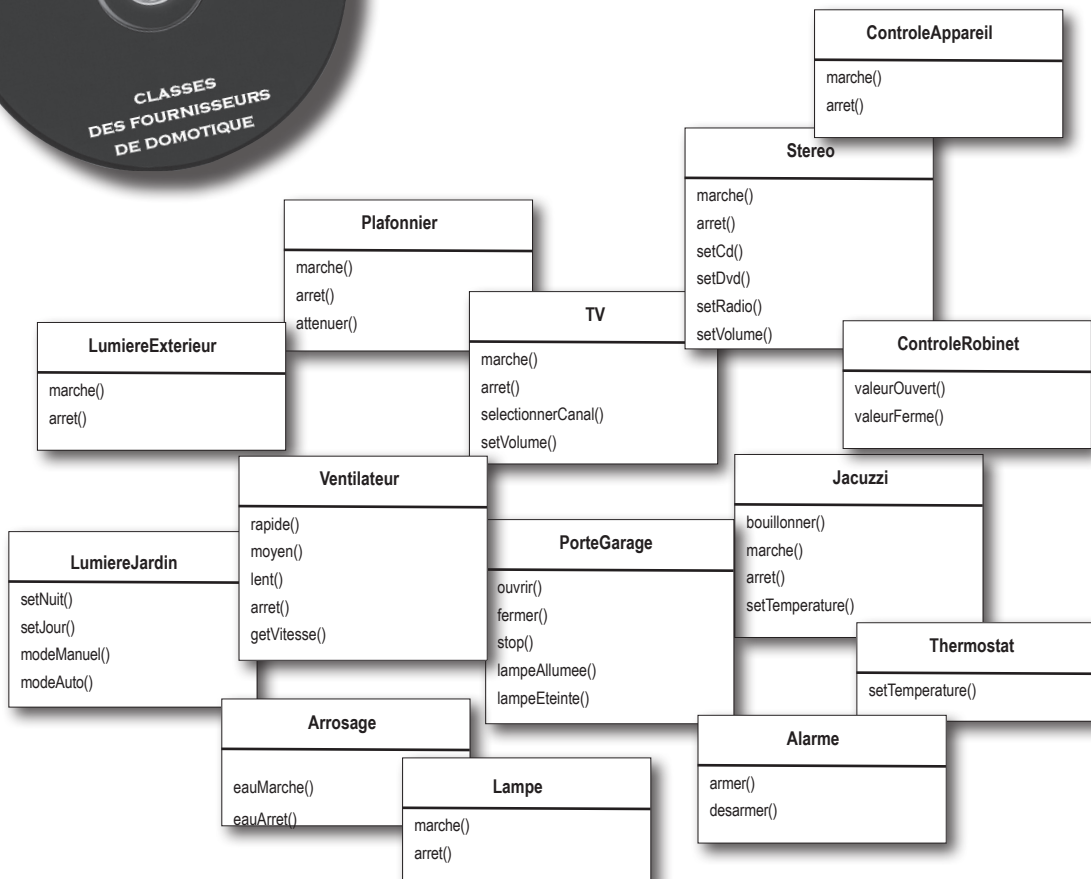
Matériel gratuit ! Examinons la commande à distance...



Jetons un coup d'œil aux classes des fournisseurs



Voyons les classes qui se trouvent sur le CD-ROM. Elles devraient nous donner une idée des interfaces des objets que nous devons contrôler avec la télécommande.

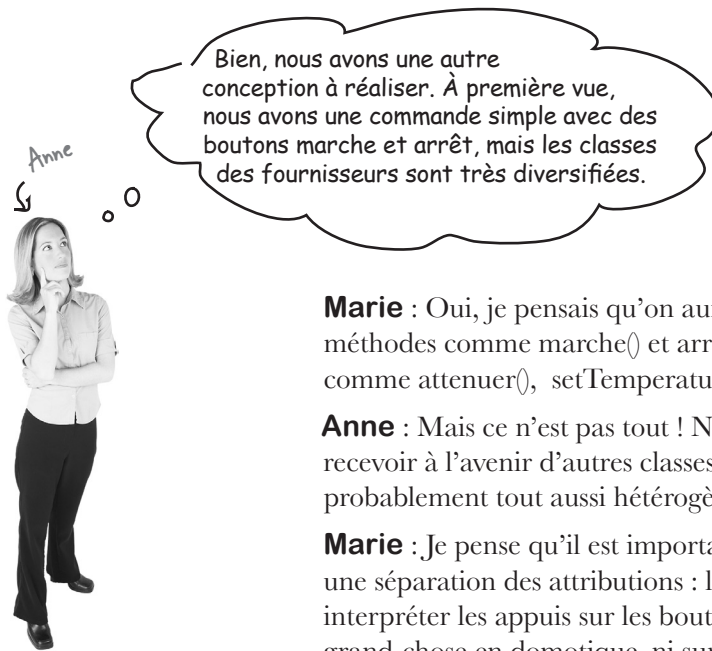


Certes, nous avons un bel ensemble de classes, mais on dirait que les industriels n'ont pas déployé beaucoup d'efforts pour proposer des interfaces communes. Pis encore, il paraît qu'elles seront de plus en plus nombreuses à l'avenir. Développer une API pour la télécommande va être une tâche intéressante.

Attelons-nous à la conception.

Conversation dans un box

Vos collègues sont déjà en train de réfléchir à la conception de l'API de la télécommande...



Marie : Oui, je pensais qu'on aurait un ensemble de classes avec des méthodes comme `marche()` et `arret()`, mais ici nous avons des méthodes comme `attenuer()`, `setTemperature()`, `setVolume()`, `setDirection()`.

Anne : Mais ce n'est pas tout ! Nous pouvons nous attendre à recevoir à l'avenir d'autres classes propriétaires, avec des méthodes probablement tout aussi hétérogènes.

Marie : Je pense qu'il est important de voir le problème comme une séparation des attributions : la commande doit savoir comment interpréter les appuis sur les boutons, mais elle n'a pas besoin de savoir grand-chose en domotique, ni sur la façon de mettre en marche un jacuzzi.

Anne : L'idée semble bonne. Mais si la commande est passive et ne sait qu'émettre des requêtes génériques, comment allons nous la concevoir pour qu'elle puisse appeler une action qui va par exemple allumer une lampe ou ouvrir une porte de garage ?

Marie : Je ne sais pas, mais il ne faut pas que la commande ait besoin de connaître les détails des classes propriétaires.

Anne : Qu'est-ce que tu veux dire ?

Marie : Il ne faut pas que la commande consiste en une suite d'instructions du type `if emplacement1 == Lampe, then lampe.marche(), else if emplacement1 == Jacuzzi then jacuzzi.marche()`. Nous savons que c'est une mauvaise pratique de conception.

Anne : Je suis d'accord. Il faudrait modifier le code chaque fois qu'un fournisseur écrirait une nouvelle classe. Il y aurait potentiellement plus de bogues... et plus de travail pour nous !



Je suis d'accord. Il faudrait modifier le code chaque fois qu'un fournisseur écrirait une nouvelle classe. Il y aurait potentiellement plus de bogues... et plus de travail pour nous !

Marie : Oui ? Tu peux nous en dire plus ?

Joël : Le pattern Commande permet de découpler l'auteur d'une requête d'action de l'objet qui effectue réellement l'action. En l'occurrence, le demandeur serait la télécommande et l'objet qui effectue l'action serait une instance de l'une de nos classes propriétaires.

Anne : Comment est-ce possible ? Comment peut-on les découpler ? Après tout, quand j'appuie sur un bouton, la télécommande doit allumer une lumière.

Joël : Vous pouvez le faire en introduisant des « objets de commande » dans votre conception. Un objet de commande encapsule une requête (par exemple allumer une lampe) à un objet spécifique (par exemple la lampe du séjour). Si on mémorise un objet de commande pour chaque bouton, quand on appuie sur celui-ci, on demande à l'objet de commande d'effectuer un certain travail. La télécommande n'a aucune idée de la nature de ce travail : c'est simplement un objet de commande qui sait comment parler au bon objet pour que le travail soit fait. Et voilà comment la télécommande est découplée de l'objet Lampe !

Anne : Oui, on dirait bien que nous sommes dans la bonne direction.

Marie : Pourtant, je n'arrive pas encore à me faire une idée du fonctionnement du pattern.

Joël : Étant donné que les objets sont si découplés, il est un peu difficile d'imaginer comment le pattern fonctionne réellement.

Marie : Voyons au moins si mon idée est juste : en appliquant ce pattern, nous pourrions créer une API dans laquelle ces objets de commande seraient chargés aux emplacements qui correspondent aux boutons, ce qui permettrait au code de la télécommande de demeurer très simple. Et les objets de commande encapsuleraient la procédure pour effectuer une tâche d'automatisation dans l'objet qui en a besoin.

Joël : Oui, il me semble. Je pense aussi que ce pattern peut vous être utile pour ce bouton Annulation, mais je n'ai pas encore étudié cette partie.

Marie : Cela semble vraiment encourageant, mais je crois qu'il me reste du travail pour « saisir » réellement le pattern.

Anne : Moi aussi.

Pendant ce temps, à la cafétéria..., ou, Une brève introduction au pattern Commande

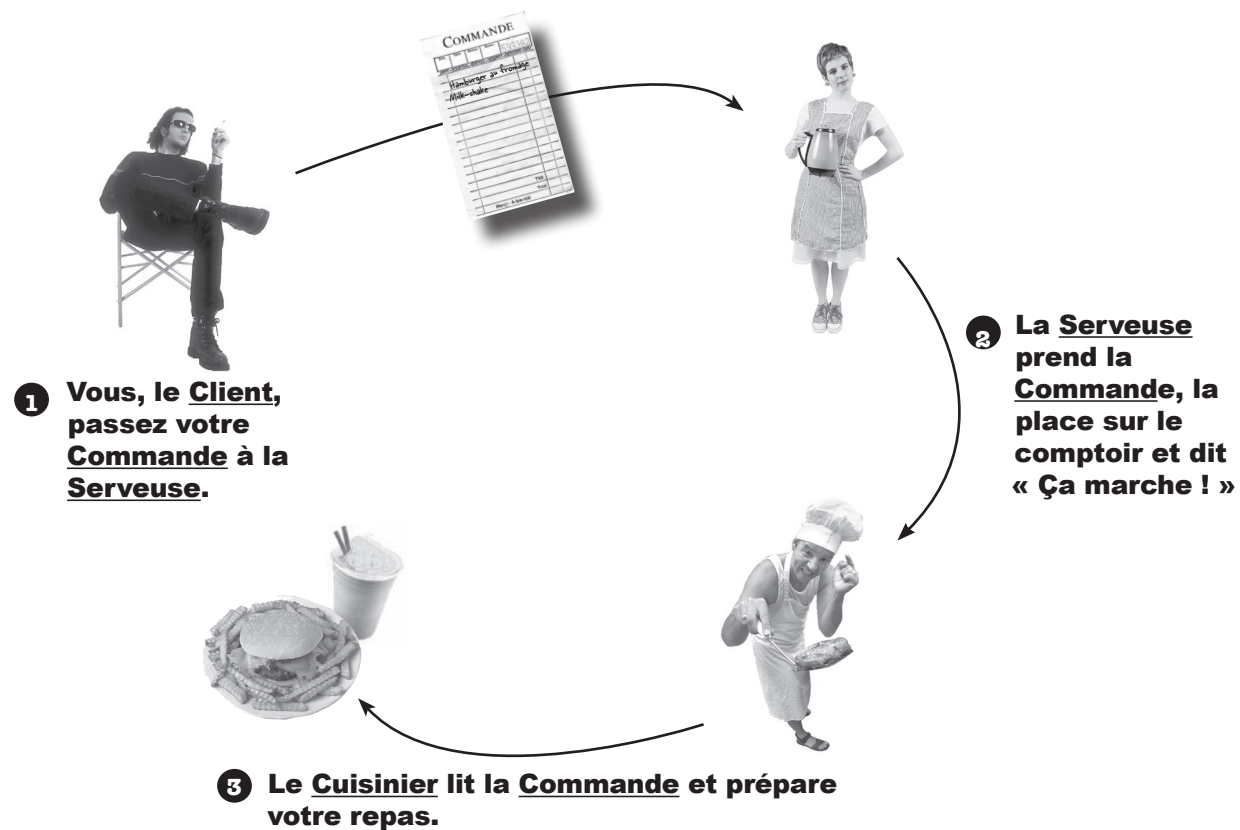
Comme l'a dit Joël, le pattern Commande est un peu difficile à comprendre à la seule lecture de sa description. Mais n'ayez crainte, nous avons des amis prêts à nous aider. Vous souvenez-vous de notre sympathique cafétéria du chapitre 1 ? Il y a bien longtemps que nous n'avons pas vu Alice, Flo et le chef, mais nous avons de bonnes raisons d'y retourner (enfin en plus de la cuisine et de la conversation grandiose) : la cafétéria va nous aider à comprendre le pattern Commande.

Faisons donc un petit détour par la cafétéria et étudions les interactions entre les clients, la serveuse, les commandes et le cuisinier. Grâce à ces interactions, vous allez comprendre les objets impliqués dans le pattern Commande et avoir une petite idée du fonctionnement du découplage. Après quoi, nous allons torcher l'API de la télécommande.

Entrons à la cafétéria d'Objectville...

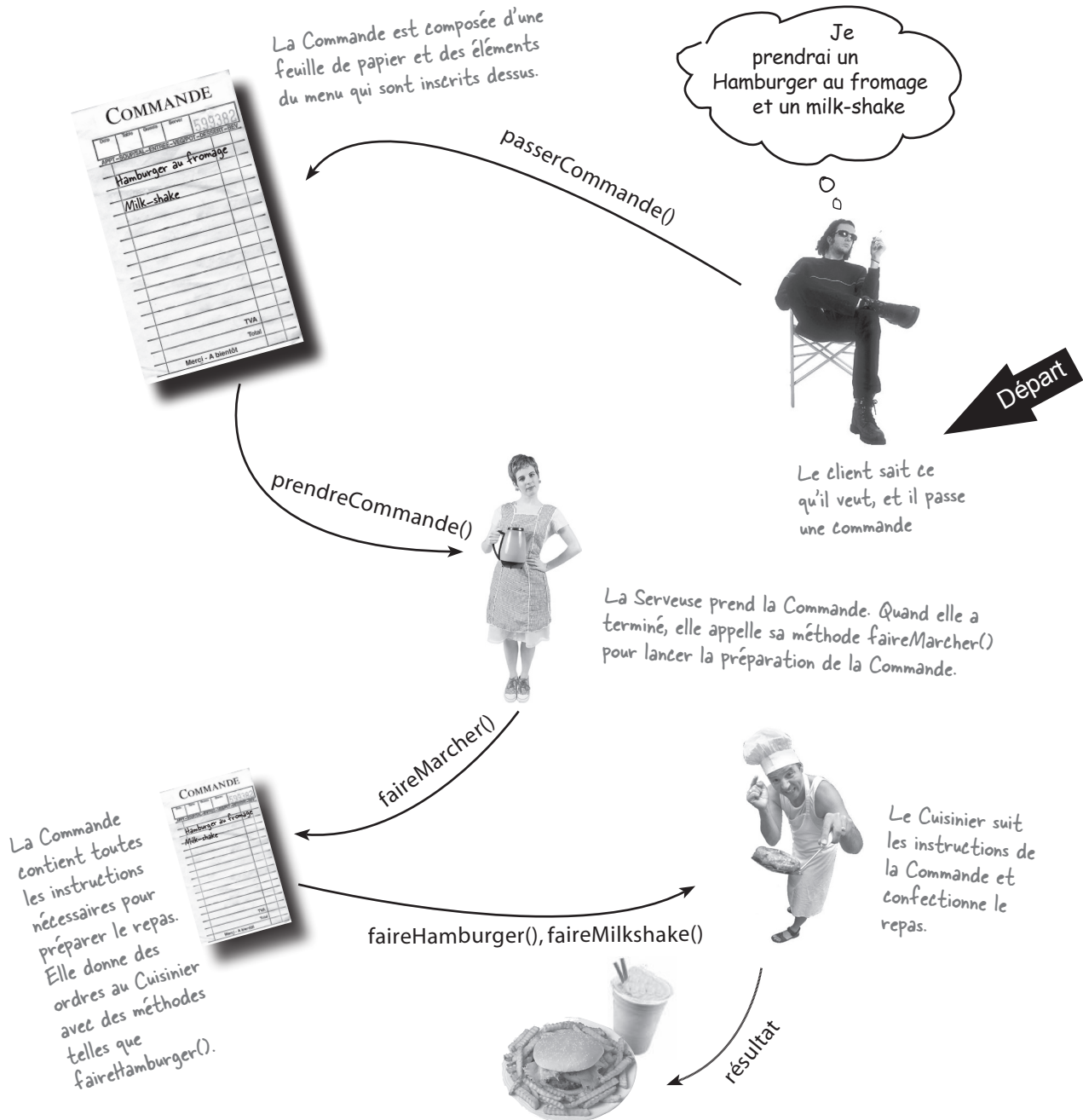


Bien, nous savons tous comment la cafétéria fonctionne :



Étudions les interactions un peu plus en détail...

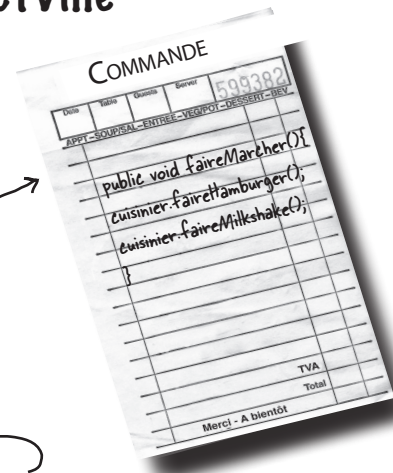
...et puisque nous sommes à la cafétéria d'Objectville, réfléchissons aussi aux objets et aux appels de méthodes impliqués !



Rôles et responsabilités de la Cafétéria d'Objectville

Une Commande encapsule une requête pour préparer un repas.

Représentez-vous la Commande comme un objet, un objet qui fait fonction de requête pour préparer un repas. Comme tout autre objet, il peut être transmis, de la Serveuse au comptoir, ou à la prochaine Serveuse qui prendra son poste. Il possède une interface qui ne contient qu'une seule méthode, `faireMarcher()`, qui encapsule les actions nécessaires pour préparer le repas. Il possède également une référence à l'objet qui doit le préparer (en l'occurrence le Cuisinier). Il est encapsulé au sens où la Serveuse n'a pas besoin de savoir ce que contient la commande, ou même qui prépare le repas : il suffit qu'elle pose la commande sur le passe-plats et qu'elle crie « ça marche » !



Bon, c'est vrai, dans la vie réelle, une serveuse s'occuperait probablement de ce qu'il y a sur la commande et de qui la prépare, mais nous sommes à Objectville... bienvenue au club !

La tâche de la Serveuse consiste à prendre les Commandes et à invoquer leur méthode `faireMarcher()`.

La Serveuse a la belle vie : prendre la commande du client et continuer à servir les autres clients jusqu'au moment où elle retourne au comptoir et où elle invoque la méthode `faireMarcher()` pour faire préparer le repas. Nous le savons déjà : à Objectville, la Serveuse est totalement indifférente à ce qui est inscrit sur la commande ou à qui va la préparer. Une seule chose l'intéresse : les commandes possèdent une méthode `faireMarcher()` qu'elle peut appeler pour que le travail soit fait.

Maintenant, tout au long de la journée, la méthode `prendreCommande()` de la Serveuse reçoit en argument différentes commandes de différents clients, mais peu lui importe. Elle sait que toutes les commandes prennent en charge la méthode `faireMarcher()` et qu'elle peut l'appeler chaque fois qu'elle a besoin de faire préparer un repas.

Ne me demandez pas de cuisiner. Je me contente de prendre les commandes et de crier « ça marche ! ».

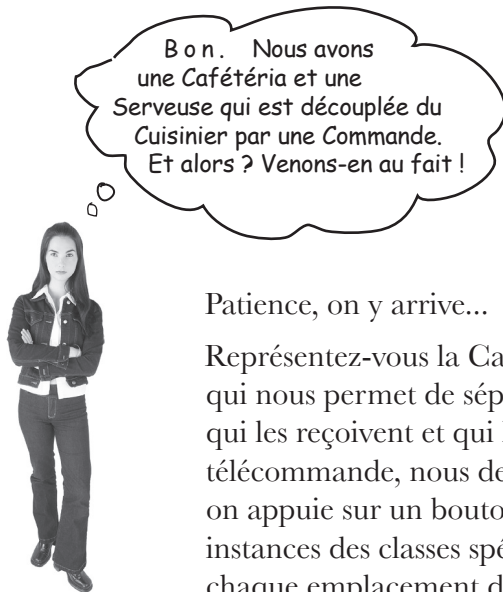


Le Cuisinier a les connaissances requises pour préparer le repas.

Le Cuisinier est l'objet qui sait réellement comment préparer des repas. Une fois que la Serveuse a invoqué la méthode `faireMarcher()`, le Cuisinier prend le relais et implémente toutes les méthodes nécessaires pour confectionner des repas. Remarquez que la Serveuse et le Cuisinier sont totalement découplés : la Serveuse a des Commandes qui encapsulent les détails du repas, et elle se contente d'appeler une méthode sur chaque commande pour qu'elle soit préparée. De même, le Cuisinier reçoit ses instructions de la Commande et n'a jamais besoin de communiquer directement avec la Serveuse.



La serveuse et moi, on est découplés, vous pouvez le dire. Elle n'est même pas mon type !



Patience, on y arrive...

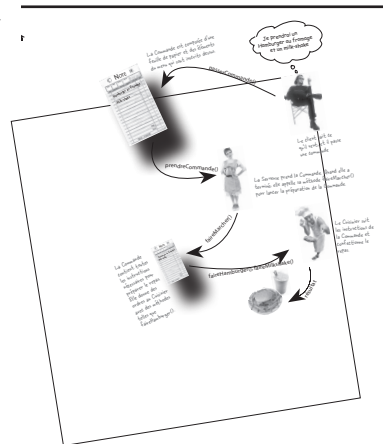
Représentez-vous la Cafétéria comme le modèle d'un design pattern qui nous permet de séparer un objet qui émet des requêtes de ceux qui les reçoivent et qui les exécutent. Par exemple, dans l'API de la télécommande, nous devons séparer le code qui est appelé quand on appuie sur un bouton des objets qui exécutent les requêtes, les instances des classes spécifiques aux fournisseurs. Que se passerait-il si chaque emplacement de la télécommande contenait un objet tel que la Commande de la Cafétéria ? Lorsqu'un bouton serait pressé, nous nous bornerions à appeler l'équivalent de la méthode `faireMarcher()` sur cet objet, et la lumière s'allumerait sans que la télécommande connaisse les détails de la façon d'allumer une lampe ni des objets nécessaires pour ce faire.

Maintenant, passons à la vitesse supérieure et voyons comment tout ce discours sur la Cafétéria correspond au pattern Commande...



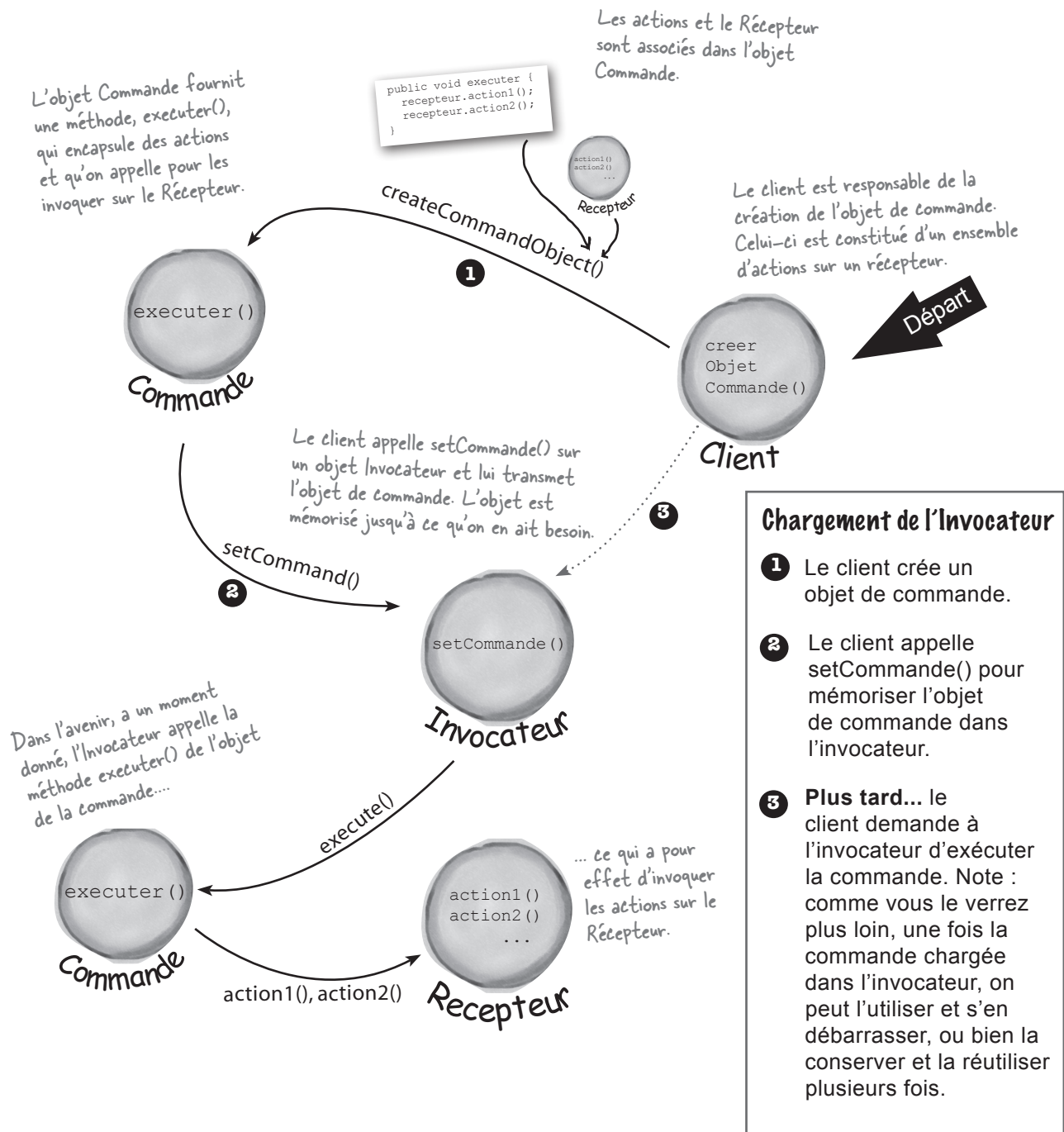
MUSCLEZ VOS NEURONES

Avant de poursuivre, consacrez quelques instants à étudier le diagramme de la page 198, ainsi que les rôles et les responsabilités de la Cafétéria pour bien comprendre les objets et les relations de la Cafétéria d'Objectville. Ceci fait, tenez-vous prêt à vous attaquer au pattern Commande !



De la Cafétéria au pattern Commande

Parfait. Nous avons passé suffisamment de temps à la Cafétéria d'Objectville pour bien connaître tous les personnages et leurs responsabilités. Nous allons maintenant retravailler le diagramme de la Cafétéria et le comparer au pattern Commande. Vous verrez que tous les acteurs sont les mêmes : seul leur nom a changé.



Qui fait quoi ?

Reliez les objets et les méthodes de la Cafétéria à leurs homologues dans le pattern Commande.

Cafétéria

Pattern Commande

Serveuse

Commande

Cuisinier

executer()

faireMarcher()

Client

Commande

Invocateur

Client

Récepteur

prendreCommande()

setCommande()

Notre premier objet de commande

Ne serait-il pas temps de construire notre premier objet de commande ? Allons-y !
Écrivons un peu de code pour la télécommande. Comme nous ne savons pas encore comment nous allons concevoir l'API de la télécommande, une démarche ascendante pourra peut-être nous aider...



Implémenter l'interface Commande

Commençons par le commencement : tous les objets de commande implémentent la même interface, laquelle ne contient qu'une seule méthode. Dans le cas de la Cafétéria, nous avons nommé cette méthode `faireMarcher()` ; toutefois, on se contente généralement de la nommer `executer()`.

Voici l'interface `Commande` :

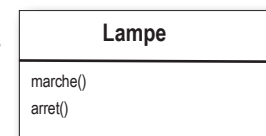
```
public interface Commande {
    public void executer();
}
```

Simple. Il suffit d'une seule méthode nommée `executer()`.

Implémenter une Commande pour allumer une lampe

Maintenant, imaginons que vous vouliez implémenter une commande pour allumer une lampe.

Si nous regardons de nouveau nos classes propriétaires, la classe `Lampe` possède deux méthodes : `marche()` et `arret()`. Voici comment vous pouvez implémenter la commande :



Comme c'est une commande, nous devons implémenter l'interface `Commande`.

```
public class CommandeAllumerLampe implements Commande {
    Lampe lampe;

    public CommandeAllumerLampe(Lampe lampe) {
        this.lampe = lampe;
    }

    public void executer() {
        lampe.marche();
    }
}
```

Nous transmettons au constructeur la lampe spécifique que cette commande va contrôler – par exemple la lampe du séjour – et il la planque dans la variable d'instance `lampe`. Quand `executer()` est appelée, c'est l'objet `Lampe` qui va être le Récepteur de la requête.

La méthode `executer()` appelle la méthode `marche()` sur l'objet récepteur : la lampe que nous voulons contrôler.

Maintenant que nous avons une classe `CommandeAllumerLampe`, voyons si nous pouvons la mettre au travail...

Utiliser l'objet de commande

Bien, simplifions les choses : disons que notre télécommande ne dispose que d'un bouton et de l'emplacement correspondant pour contenir l'appareil à contrôler :

```
public class TelecommandeSimple {
    Commande emplacement;

    public TelecommandeSimple() {}
    public void setCommande(Commande commande) {
        emplacement = commande;
    }

    public void boutonPresse() {
        emplacement.executer();
    }
}
```

Notre commande n'a qu'un seul bouton qui ne contrôle qu'un équipement

Nous avons une méthode pour indiquer à la commande l'emplacement qu'il va contrôler. Elle pourrait être appelée plusieurs fois si le client de code voulait modifier le comportement du bouton de la commande.

Cette méthode est appelée quand on presse le bouton. Il suffit de prendre la commande courante associée à l'emplacement et d'appeler sa méthode executer().

Créer un test simple pour essayer la télécommande

Voici un petit programme qui va tester notre télécommande simple. Regardons-le et voyons comment ses éléments correspondent au diagramme de classes du pattern Commande :

```
public class TestTelecommande {
    public static void main(String[] args) {
        TelecommandeSimple telecommande = new TelecommandeSimple();
        Lampe lampe = new Lampe();
        CommandeAllumerLampe lampeAllumee =
            new CommandeAllumerLampe(lampe);

        telecommande.setCommande(lampeAllumee);
        telecommande.boutonPresse();
    }
}
```

Voici notre Client, dans le langage du pattern Commande

La télécommande est notre Invokeur ; nous lui transmettons un objet de commande qui servira pour émettre des requêtes.

Maintenant, nous créons l'objet Lampe. Ce sera le Récepteur de la requête.

Ici, nous créons une commande et nous la transmettons au Récepteur

Là, nous transmettons la commande à l'Invokeur.

Et là nous simulons l'appui sur le bouton.

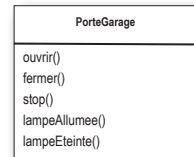
Voici le résultat de l'exécution de ce test !

```
Fichier Édition Fenêtre Aide FiatLux
%java TestTelecommande
Lampe allumée
%
```




Bien. Il est temps pour vous d'implémenter la classe `CommandeOuvrirPorteGarage`. Commencez par le code de la classe ci-après. Vous aurez besoin du diagramme de classes de `PorteGarage`.

```
public class CommandeOuvrirPorteGarage
implements Commande {
```



```
}
```

Écrivez votre code ici.

Maintenant que vous avez votre classe, quel est le résultat du code suivant ? (Indication : la méthode `ouvrir()` de `PorteGarage` affiche « Porte garage ouverte » quand elle se termine.)

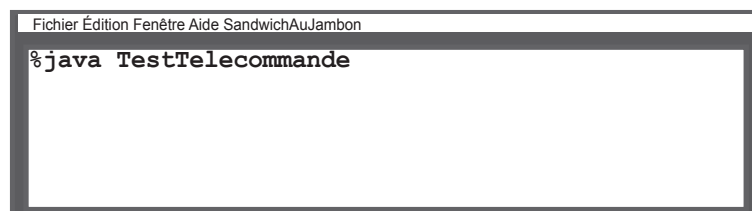
```
public class TestTelecommande {

    public static void main(String[] args) {

        TelecommandeSimple telecommande = new TelecommandeSimple();
        Lampe lampe = new Lampe();
        PorteGarage porteGarage = new PorteGarage();
        CommandeAllumerLampe lampeAllumee = new CommandeAllumerLampe(lampe);
        CommandeOuvrirPorteGarage garageOuvert = new CommandeOuvrirPorteGarage
                                                    (porteGarage);

        telecommande.setCommande(lampeAllumee);
        telecommande.boutonPresse();
        telecommande.setCommande(garageOuvert);
        telecommande.boutonPresse();
    }
}
```

Inscrivez le résultat ici.



Le pattern *Commande* : définition

Vous avez passé un certain temps à la Cafétéria d'Objectville, vous avez partiellement implémenté l'API de la télécommande et le processus vous a permis de vous faire une bonne idée des interactions des classes et des objets dans le pattern *Commande*. Nous allons maintenant définir ce pattern *Commande* et en étudier tous les détails.

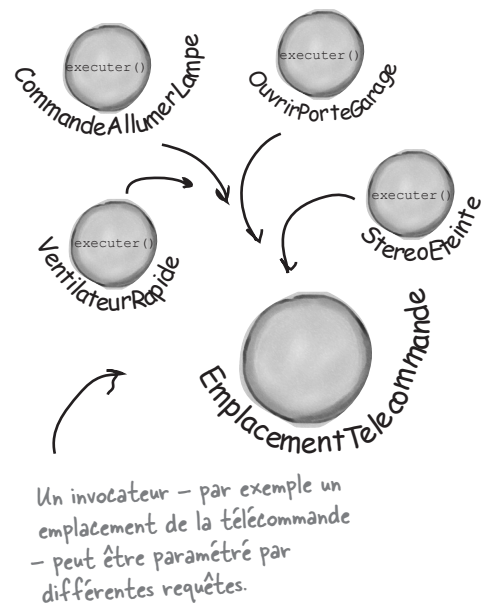
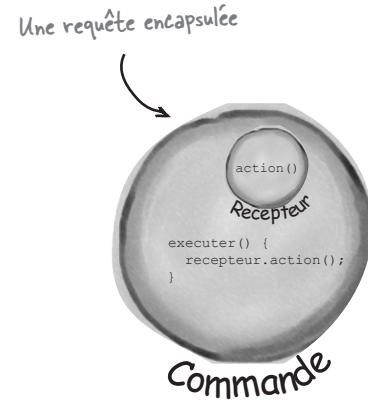
Commençons par la définition officielle :

Le pattern *Commande* encapsule une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes, et de plus, permettant la réversibilité des opérations.

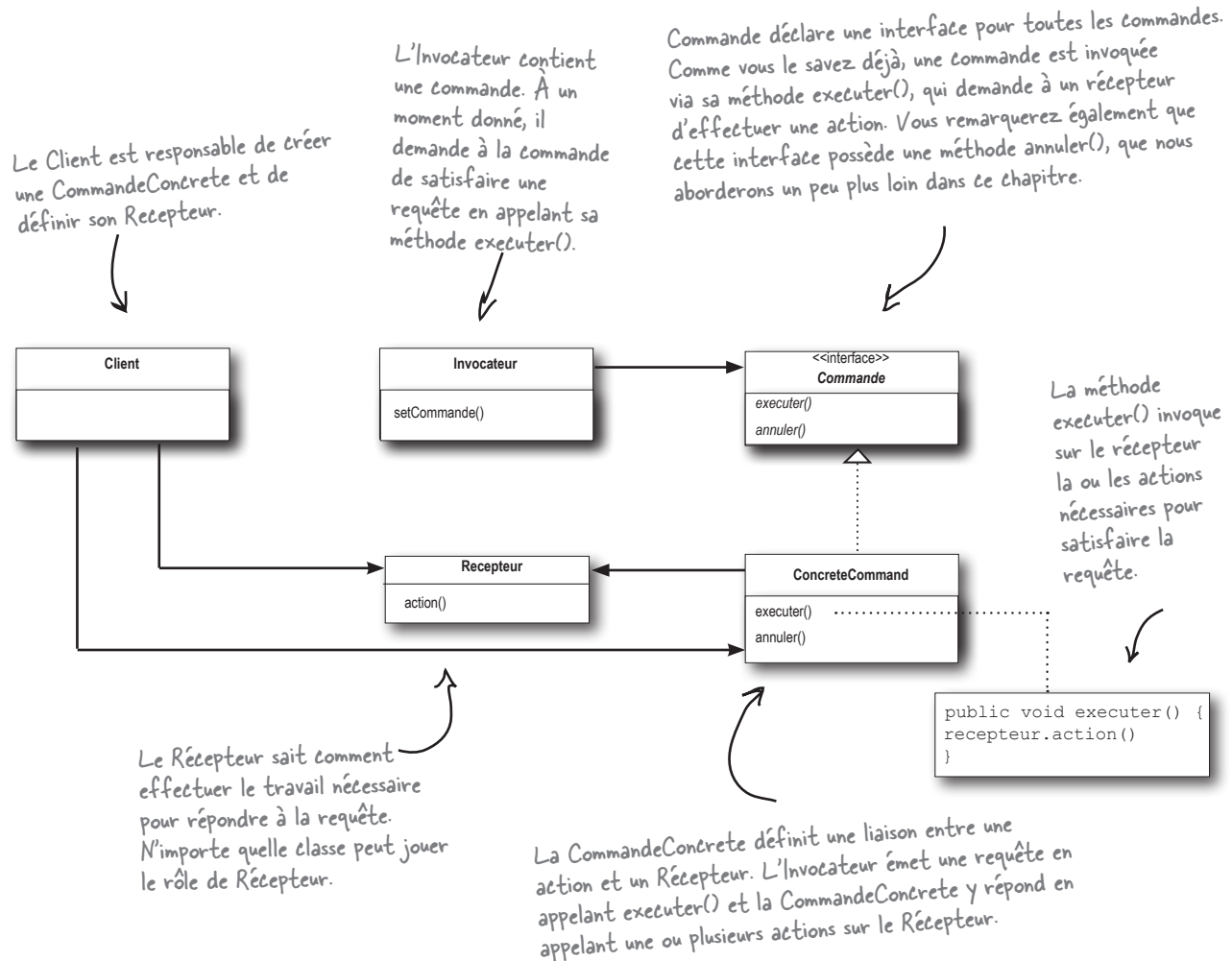
Décortiquons un peu. Nous savons qu'un objet de commande *encapsule une requête* en associant un ensemble d'actions à un récepteur spécifique. Pour ce faire, il groupe les actions et le récepteur dans un objet qui n'expose qu'une seule méthode, `executer()`. Quand elle est appelée, `executer()` provoque l'invocation des actions sur le récepteur. De l'extérieur, aucun autre objet ne sait réellement quelles sont les actions exécutées ni de quel récepteur il s'agit. Ils ne savent qu'une chose : s'ils appellent la méthode `executer()`, leur requête sera satisfaite.

Nous avons également vu quelques exemples de *paramétrage d'un objet* avec une commande. À la Cafétéria, la Serveuse était paramétrée par de multiples commandes toute la journée. Dans la simple télécommande, nous avons commencé par charger dans l'emplacement du bouton une commande « allumer la lumière » que nous avons remplacée plus tard par une commande « ouvrir la porte du garage. Comme la Serveuse, l'emplacement en question ne se soucie pas de savoir quel objet de commande il manipule, tant qu'il implémente l'interface *Commande*.

Ce que nous n'avons pas encore abordé, c'est l'utilisation de commandes pour implémenter des files d'attente et des récapitulatifs de requêtes et pour prendre en charge la *réversibilité des opérations*. Ne vous inquiétez pas, ce sont des extensions relativement simples du pattern *Commande* de base et nous les étudierons bientôt. Nous verrons également qu'il est facile d'appliquer ce qu'on appelle le pattern *MétaCommande* une fois les bases en place. Le pattern *MétaCommande* permet de créer des macro-commandes pour pouvoir exécuter plusieurs commandes à la fois.

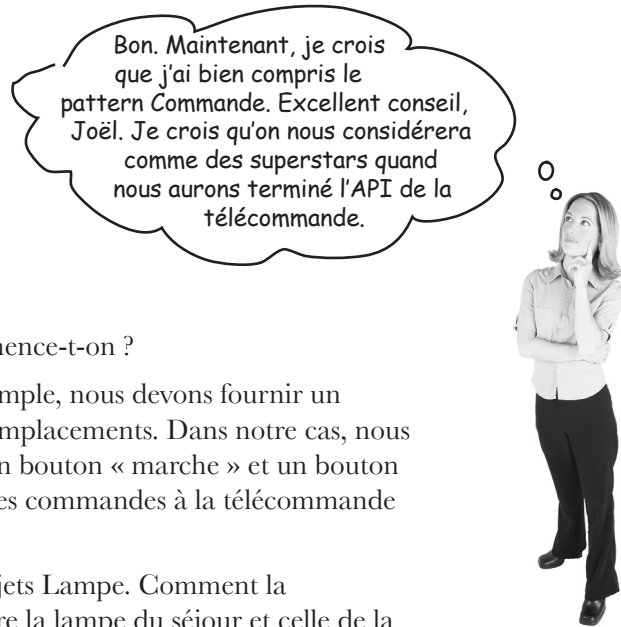


Le pattern Commande : le diagramme de classes



MUSCLEZ VOS NEURONES

Comment la conception du pattern Commande permet-elle de découpler l'invocateur de la requête de son récepteur ?



Marie : Moi aussi. Alors, par où commence-t-on ?

Anne : Comme pour `TelecommandeSimple`, nous devons fournir un moyen d'affecter des commandes aux emplacements. Dans notre cas, nous avons sept emplacements et chacun a un bouton « marche » et un bouton « arrêt ». Nous pouvons donc affecter des commandes à la télécommande un peu de la même manière.

Marie : C'est faisable, sauf pour les objets `Lampe`. Comment la télécommande fait-elle la différence entre la lampe du séjour et celle de la cuisine ?

Anne : Ah, mais justement, elle ne la fait pas ! Elle ne sait rien faire d'autre qu'appeler `executer()` sur l'objet de commande correspondant quand on presse un bouton.

Marie : Oui, j'ai bien compris, mais dans l'implémentation, comment allons-nous nous assurer que les bons objets allument et éteignent les bons appareils ?

Anne : Quand nous créerons les commandes qui seront chargées dans la télécommande, nous créerons une `CommandeLampe` qui sera associée à la lampe du séjour et une autre qui sera associée à celle de la cuisine. N'oublie pas que le récepteur de la requête est lié à la commande dans laquelle il est encapsulé. À partir du moment où le bouton est pressé, personne ne se soucie de savoir de quelle lampe il s'agit : tout se passe correctement quand la méthode `executer()` est appelée.

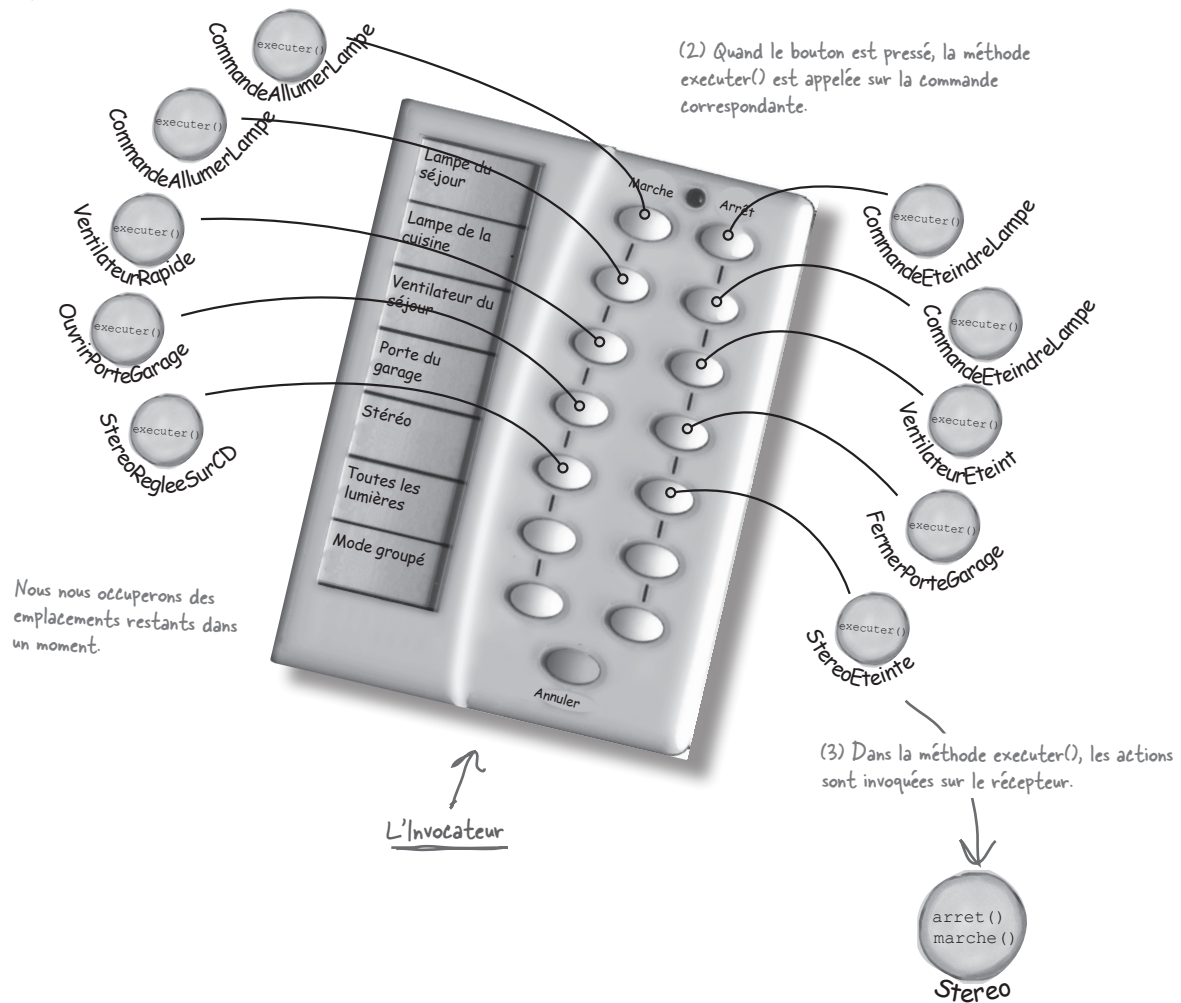
Marie : Je crois que ça y est. Implémentons la télécommande et je pense que tout va s'éclaircir !

Anne : À la bonne heure. Allons-y...

Affecter des Commandes aux emplacements

Nous avons donc un plan. Nous allons affecter une commande à chaque emplacement de la télécommande. Cela fait de la télécommande notre *Invoker*. Lorsqu'on appuiera sur un bouton, la méthode `executer()` sera appelée sur la commande correspondante, ce qui aura pour effet d'invoquer des actions sur un récepteur (par exemple une lampe, un ventilateur ou une chaîne stéréo).

(1) Chaque emplacement reçoit une commande.



Implémenter la télécommande

```
public class Telecommande {
    Commande[] commandesMarche;
    Commande[] commandesArret;
```

Cette fois, la télécommande va gérer sept commandes Marche ou Arrêt, que nous allons mémoriser dans les tableaux correspondants.

```
    public Telecommande() {
        commandesMarche = new Commande[7];
        commandesArret = new Commande[7];
```

Dans le constructeur, il suffit d'instancier et d'initialiser deux tableaux pour les deux types de commandes.

```
        Commande pasDeCommande = new PasDeCommande();
        for (int i = 0; i < 7; i++) {
            commandesMarche[i] = pasDeCommande;
            commandesArret[i] = pasDeCommande;
        }
    }
```

```
    public void setCommande(int empt, Commande comMarche, Commande comArret) {
        commandesMarche[empt] = comMarche;
        commandesArret[empt] = comArret;
```

La méthode `setCommande()` accepte la position d'un emplacement et les commandes Marche et Arrêt qui y seront stockées. Puis elle place ces commandes dans les tableaux correspondants pour qu'on puisse les utiliser plus tard.

```
    public void boutonMarchePresse(int empt) {
        commandesMarche[empt].executer();
    }
```

```
    public void boutonArretPresse(int empt) {
        commandesArret[empt].executer();
    }
```

Quand on appuie sur un bouton Marche ou Arrêt, le matériel se charge d'appeler les méthodes correspondantes : `boutonMarchePresse()` ou `boutonArretPresse()`.

```
    public String toString() {
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append("\n----- Télécommande ----- \n");
        for (int i = 0; i < commandesMarche.length; i++) {
            stringBuffer.append("[empt " + i + "] " + commandesMarche[i].getClass().getName()
                + " " + commandesArret[i].getClass().getName() + "\n");
        }
        return stringBuffer.toString();
    }
}
```

Nous avons redéfini `toString()` pour afficher chaque emplacement et la commande correspondante. Vous verrez pourquoi quand nous testerons la télécommande.

Implémenter les Commandes

Bien. Nous nous sommes déjà mouillé les pieds en implémentant `CommandeAllumerLampe` pour la `TelecommandeSimple`. Nous pouvons insérer ce même code ici et tout marche magnifiquement bien. Les commandes Arrêt ne sont pas si différentes. En fait, `CommandeEteindreLampe` ressemble à ceci :

```
public class CommandeEteindreLampe implements Commande {
    Lampe lampe;

    public CommandeEteindreLampe(Lampe lampe) {
        this.lampe = lampe;
    }

    public void executer() {
        lampe.arret();
    }
}
```

La `CommandeEteindreLampe` fonctionne également de la même manière que `CommandeAllumerLampe`, excepté que nous associons le récepteur à une action différente : la méthode `arret()`.

Essayons quelque chose d'un peu plus difficile ! Que diriez-vous décrire des commandes marche et arrêt pour la chaîne stéréo ? Pour l'éteindre, c'est facile : il suffit d'associer la `Stereo` à la méthode `arret()` dans la classe `CommandeEteindreStereo`. En revanche, c'est un peu plus compliqué pour l'allumer. Disons que nous voulons écrire une classe `CommandeAllumerStereoAvecCD`...

Stereo
<code>marche()</code> <code>arret()</code> <code>setCd()</code> <code>setDvd()</code> <code>setRadio()</code> <code>setVolume()</code>

```
public class CommandeAllumerStereoAvecCD implements Commande {
    Stereo stereo;

    public CommandeAllumerStereoAvecCD(Stereo stereo) {
        this.stereo = stereo;
    }

    public void executer() {
        stereo.marche();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Tout comme pour `CommandeAllumerLampe`, on passe au constructeur l'instance de la stéréo que nous allons contrôler et on la mémorise dans une variable d'instance locale.

Pour satisfaire à cette requête, nous devons appeler trois méthodes sur la stéréo : d'abord l'allumer, puis lui dire de jouer le CD et enfin régler le volume à 11. Pourquoi 11 ? Eh bien, c'est mieux que 10, non ?

Pas si mal. Jetons un œil au reste des classes propriétaires : ici et maintenant, vous pouvez faire un sort aux autres classes `Commande` dont nous avons besoin.

Voyons ce dont la télécommande est capable

Nous en avons presque terminé avec la télécommande : il ne nous reste plus qu'à exécuter quelques tests et à rassembler la documentation pour décrire l'API. Ils vont sûrement être impressionnés chez Maisons de rêve SARL., ne croyez-vous pas ? Nous avons réussi à mettre au point une conception qui va leur permettre de produire une télécommande facile à maintenir et ils n'auront aucun problème pour convaincre les fournisseurs d'écrire à l'avenir des classes de commandes simples, puisqu'elles sont si faciles à créer.

Testons donc ce code !

```
public class ChargeurTelecommande {

    public static void main(String[] args) {
        Telecommande teleCommande = new Telecommande();

        Lampe lampeSejour= new Lampe("Séjour");
        Lampe lampeCuisine = new Lampe("Cuisine");
        Ventilateur ventilateur= new Ventilateur("Séjour");
        PorteGarage porteGarage = new PorteGarage("");
        Stereo stereo = new Stereo("Séjour");

        CommandeAllumerLampe lampeSejourAllumee =
            new CommandeAllumerLampe(lampeSejour);
        CommandeEteindreLampe lampeSejourEteinte =
            new CommandeEteindreLampe(lampeSejour);
        CommandeAllumerLampe lampeCuisineAllumee =
            new CommandeAllumerLampe(lampeCuisine);
        CommandeEteindreLampe lampeCuisineEteinte =
            new CommandeEteindreLampe(lampeCuisine);

        CommandeAllumerVentilateur ventilateurAllume =
            new CommandeAllumerVentilateur(ventilateur);
        CommandeEteindreVentilateur ventilateurEteint =
            new CommandeEteindreVentilateur(ventilateur);

        CommandeOuvrirPorteGarage porteGarageOuvverte =
            new CommandeOuvrirPorteGarage(porteGarage);
        CommandeFermerPorteGarage porteGarageFermee =
            new CommandeFermerPorteGarage(porteGarage);

        CommandeAllumerStereoAvecCD stereoAvecCD =
            new CommandeAllumerStereoAvecCD(stereo);
        CommandeEteindreStereo stereoEteinte =
            new CommandeEteindreStereo(stereo);
    }
}
```

Créer tous les appareils à l'emplacement approprié.

Créer tous les objets de commande des lampes.

Créer les commandes Marche et Arrêt pour le ventilateur.

Créer les commandes Marche et Arrêt pour le garage.

Créer les commandes Marche et Arrêt pour la stéréo.


```

teleCommande.setCommande(0, lampeSejourAllumee, lampeSejourEteinte);
teleCommande.setCommande(1, lampeCuisineAllumee, lampeCuisineEteinte);
teleCommande.setCommande(2, ventilateurAllume, ventilateurEteint);
teleCommande.setCommande(3, stereoAvecCD, stereoEteinte);

System.out.println(teleCommande);

teleCommande.boutonMarchePresse(0);
teleCommande.boutonArretPresse(0);
teleCommande.boutonMarchePresse(1);
teleCommande.boutonArretPresse(1);
teleCommande.boutonMarchePresse(2);
teleCommande.boutonArretPresse(2);
teleCommande.boutonMarchePresse(3);
teleCommande.boutonArretPresse(3);

}
}

```

Maintenant que nous avons toutes nos commandes, nous pouvons les charger dans les emplacements de la télécommande.

C'est ici que nous utilisons la méthode `toString()` pour afficher chaque emplacement et la commande à laquelle il est affecté.

Voilà, nous sommes prêts ! Maintenant nous parcourons chaque emplacement et nous pressons son bouton Marche ou Arrêt.

Vérifions maintenant l'exécution du test de notre télécommande...

Fichier Édition Fenêtre Aide MesSuperCommandes

```

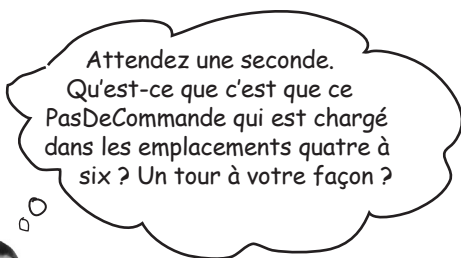
% java ChargeurTelecommande
----- Télécommande -----
[empt 0] tete premiere.commande.telecommande.CommandeAllumerLampe      tete premiere.commande.telecommande.CommandeEteindreLampe
[empt 1] tete premiere.commande.telecommande.CommandeAllumerLampe      tete premiere.commande.telecommande.CommandeEteindreLampe
[empt 2] tete premiere.commande.telecommande.CommandeAllumerVentilateur tete premiere.commande.telecommande.CommandeEteindreVentilateur
[empt 3] tete premiere.commande.telecommande.CommandeAllumerStereoAvecCD tete premiere.commande.telecommande.CommandeEteindreStereo
[empt 4] tete premiere.commande.telecommande.PasDeCommande             tete premiere.commande.telecommande.PasDeCommande
[empt 5] tete premiere.commande.telecommande.PasDeCommande             tete premiere.commande.telecommande.PasDeCommande
[empt 6] tete premiere.commande.telecommande.PasDeCommande             tete premiere.commande.telecommande.PasDeCommande

Séjour: lampe allumée
Séjour: lampe éteinte
Cuisine: lampe allumée
Cuisine: lampe éteinte
Séjour: ventilateur sur rapide
Séjour: ventilateur arrêté
Séjour: stéréo allumée
Séjour: stéréo réglée pour le CD
Séjour: le volume stéréo est 11
Séjour: stéréo éteinte
%

```

emplacements Marche emplacements Arrêt

Nos commandes en action ! Souvenez-vous que ce qui est affiché pour chaque appareil vient d'une classe du fournisseur. Par exemple, quand un objet Lampe est allumée, un message affiche << Lampe séjour allumée >>



Bien vu. Nous avons glissé là un petit quelque chose. Dans la télécommande, nous ne voulions pas vérifier qu'une commande était chargée chaque fois que nous référençons un emplacement. Par exemple, dans la méthode boutonMarchePresse(), il nous faudrait un fragment de code de ce genre :

```
public void boutonMarchePresse(int empt) {
    if (commandesMarche[empt] != null) {
        commandesMarche[empt].executer();
    }
}
```

Alors, comment résoudre le problème ? Implémenter une commande qui ne fait rien !

```
public class PasDeCommande implements Commande {
    public void executer() { }
}
```

Ainsi, dans le constructeur de notre Télécommande, nous affectons à chaque emplacement un objet PasDeCommande par défaut et nous savons que nous avons toujours une commande à appeler à chaque emplacement.

```
Commande pasDeCommande = new PasDeCommande();
for (int i = 0; i < 7; i++) {
    commandesMarche[i] = pasDeCommande;
    commandesArret[i] = pasDeCommande;
}
```

Et dans le résultat de notre test, vous voyez des emplacements qui ne contiennent rien d'autre que l'objet PasDeCommande par défaut que nous avons affecté quand nous avons créé la Télécommande.



Mention Honorable

L'objet PasDeCommande est un exemple d'objet null. Un objet null est utile quand il n'y a pas d'objet significatif à retourner mais que vous ne voulez pas laisser au client la responsabilité de gérer null. Par exemple, dans notre télécommande, nous n'avions pas d'objets significatifs à affecter à chaque emplacement du boîtier et nous avons créé un objet PasDeCommande qui sert de substitut et ne fait rien quand sa méthode executer() est appelée.

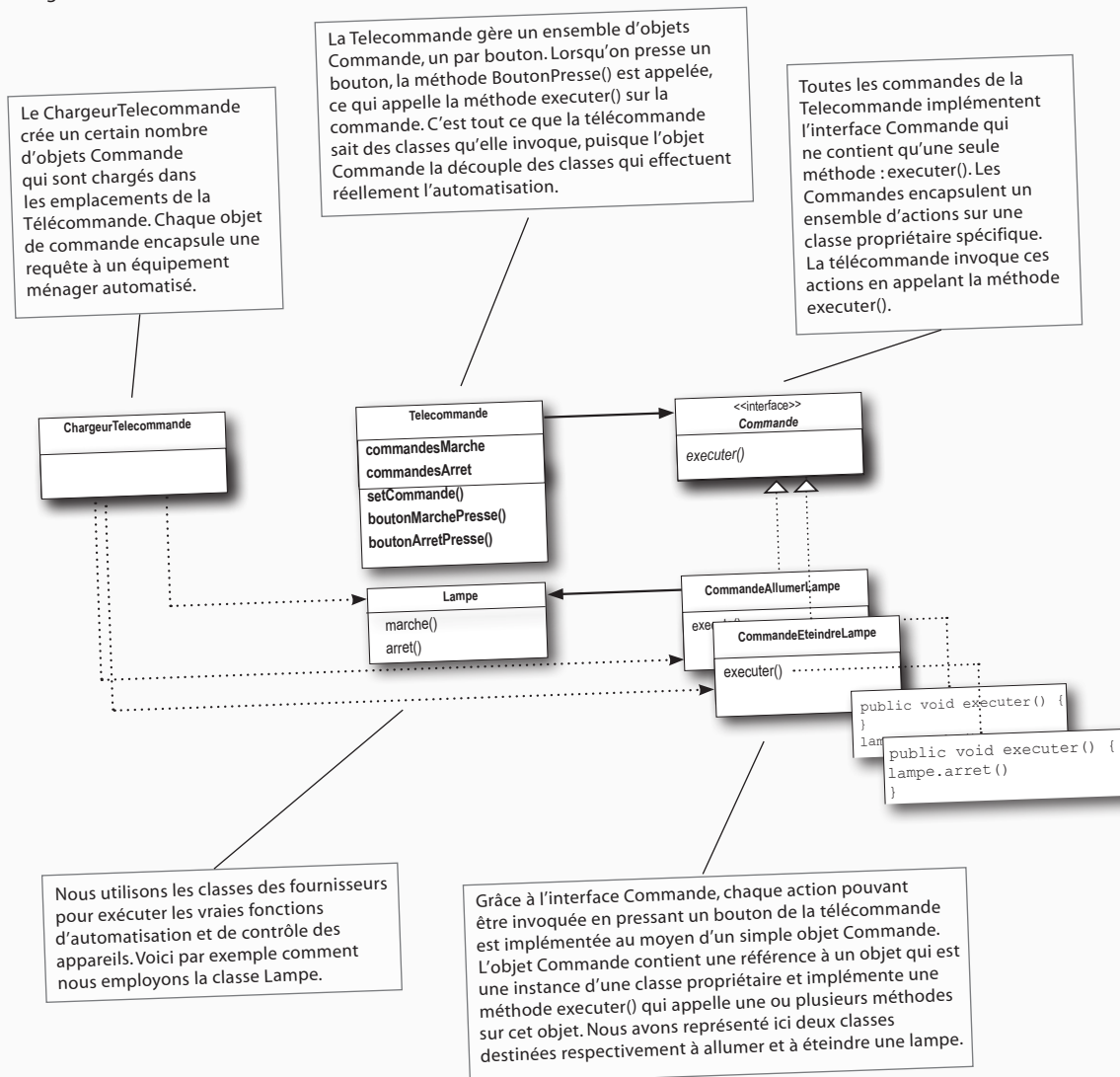
Vous trouverez des utilisations d'Objet Null combinées à de nombreux design patterns, et vous verrez même que certains auteurs le considèrent comme un pattern à part entière.

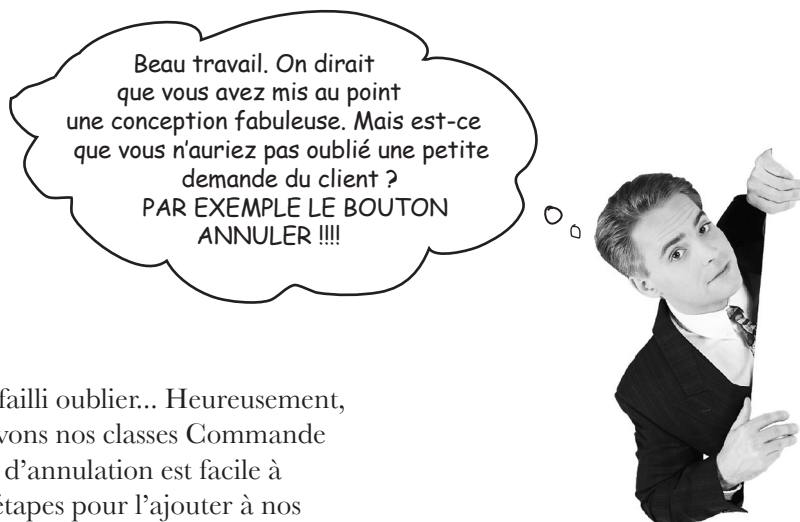
Il est temps de rédiger cette documentation...

Conception de l'API de la télécommande pour Maisons de rêve, SARL

Nous avons le plaisir de vous présenter la conception de l'API de votre télécommande domotique. Notre premier objectif a été de faire en sorte que le code de la télécommande demeure aussi simple que possible pour ne pas nécessiter de modifications quand les fournisseurs créeront de nouvelles classes. À cette fin, nous avons employé le pattern Commande pour découpler logiquement la classe Telecommande des classes propriétaires. Nous sommes convaincus que cela réduira les coûts de production de la télécommande tout en diminuant radicalement les coûts de maintenance.

Le diagramme de classes ci-dessous fournit une vue d'ensemble de notre conception :





Oups ! Nous avons failli oublier... Heureusement, une fois que nous avons nos classes Commande de base, la fonction d'annulation est facile à insérer. Voyons les étapes pour l'ajouter à nos commandes et à la télécommande...

Qu'allons-nous faire ?

Nous devons ajouter une fonctionnalité qui prendra en charge le bouton Annuler de la télécommande. En voilà le principe. Disons que la lampe du séjour est éteinte et que vous appuyez sur le bouton de la télécommande. De toute évidence, la lumière s'allume. Maintenant, si vous appuyez sur le bouton Annuler, la dernière commande est inversée – en l'occurrence la lumière s'éteint. Avant de rentrer dans des exemples plus complexes, manipulons la lampe avec le bouton Annuler :

- 1 Quand les commandes peuvent être inversées, elles possèdent une commande `annuler()` qui est l'homologue de la méthode `executer()`. Quelle que soit la dernière action d'`executer()`, `annuler()` l'inverse. Avant d'ajouter une fonction d'annulation à nos commandes, nous devons donc insérer une méthode `annuler()` dans l'interface `Commande` :

```
public interface Commande {  
    public void executer();  
    public void annuler();  
}
```

Voici la nouvelle méthode annuler().

L'enfance de l'art !

Lançons-nous maintenant dans la commande de la Lampe et implémentons la méthode `annuler()`.

2

Commençons par `CommandeAllumerLampe`. Si la méthode `executer()` de `CommandeAllumerLampe` a été appelée, alors la méthode `marche()` a été appelée en dernier. Nous savons qu'`annuler()` doit faire l'opposé en appelant la méthode `arret()`.


```
public class CommandeAllumerLampe implements Commande {
    Lampe lampe;

    public CommandeAllumerLampe(Lampe lampe) {
        this.lampe = lampe;
    }

    public void executer() {
        lampe.marche();
    }

    public void annuler() {
        lampe.arret();
    }
}
```

executer() allume la lumière, et, par conséquent, annuler() l'éteint.



Du gâteau ! Maintenant passons à `CommandeEteindreLampe`. Ici, la méthode `annuler()` doit simplement appeler la méthode `marche()` de `Lampe`.


```
public class CommandeEteindreLampe implements Commande {
    Lampe lampe;

    public CommandeEteindreLampe(Lampe lampe) {
        this.lampe = lampe;
    }

    public void executer() {
        lampe.arret();
    }

    public void annuler() {
        lampe.marche();
    }
}
```

Et ici, annuler() rallume la lumière !



Quoi de plus facile ? D'accord, nous n'avons pas fini : il nous reste un peu de travail sur la Télécommande pour gérer la mémorisation du dernier bouton pressé et le bouton Annuler.

- 3 Pour ajouter la prise en charge du bouton Annuler, il suffit d'apporter quelques modifications à la classe Télécommande. Voici comment nous allons procéder : nous allons insérer une nouvelle variable d'instance pour mémoriser la dernière commande invoquée, puis, à chaque appui sur le bouton Annuler, nous extrairons cette commande et nous appellerons sa méthode annuler().

```
public class TelecommandeAvecAnnul {
    Commande[] commandesMarche;
    Commande[] commandesArret;
    Commande commandeAnnulation;
```

C'est ici que nous mémorisons la dernière commande exécutée à l'usage du bouton Annuler.

```
public TelecommandeAvecAnnul() {
    commandesMarche = new Commande[7];
    commandesArret = new Commande[7];
```

Tout comme les autres emplacements, Annuler est initialisé à PasDeCommande : si on appuie dessus avant tout autre bouton, il ne fait strictement rien.

```
    Commande pasDeCommande = new PasDeCommande();
    for(int i=0;i<7;i++) {
        commandesMarche[i] = pasDeCommande;
        commandesArret[i] = pasDeCommande;
    }
    commandeAnnulation = pasDeCommande;
}
```

```
public void setCommande(int empt, Commande comMarche, Commande comArret) {
    commandesMarche[empt] = comMarche;
    commandesArret[empt] = comArret;
}
```

```
public void boutonMarchePresse(int empt) {
    commandesMarche[empt].executer();
    commandeAnnulation = commandesMarche[empt];
}
```

Quand un bouton est pressé, nous prenons la commande et nous commençons par l'exécuter ; puis nous sauvegardons une référence à celle-ci dans la variable d'instance commandeAnnulation. Nous procédons ainsi pour les commandes « marche » et pour les commandes « arrêt ».

```
public void boutonArretPresse(int empt) {
    commandesArret[empt].executer();
    commandeAnnulation = commandesArret[empt];
}
```

```
public void boutonAnnulPresse() {
    commandeAnnulation.annuler();
}
```

Quand le bouton Annuler est pressé, nous invoquons la méthode annuler() de la commande mémorisée dans commandeAnnulation. Ceci inverse l'effet de la dernière commande exécutée.

```
public String toString() {
    // code de toString...
}
```


Il est temps de tester le bouton Annuler !

Retravaillons un peu le test en prenant en compte le bouton Annuler :

```
public class ChargeurTelecommande {

    public static void main(String[] args) {
        TelecommandeAvecAnnul teleCommande = new TelecommandeAvecAnnul();

        Lampe lampeSejour= new Lampe("Séjour");

        CommandeAllumerLampe lampeSejourAllumee =
            new CommandeAllumerLampe(lampeSejour);

        CommandeEteindreLampe lampeSejourEteinte =
            new CommandeEteindreLampe(lampeSejour);
        teleCommande.setCommande(0, lampeSejourAllumee, lampeSejourEteinte);

        teleCommande.boutonMarchePresse(0);
        teleCommande.boutonArretPresse(0);
        System.out.println(teleCommande);
        teleCommande.boutonAnnulPresse();
        teleCommande.boutonArretPresse(0);
        teleCommande.boutonMarchePresse(0);
        System.out.println(teleCommande);
        teleCommande.boutonAnnulPresse();
    }
}
```

Créer une Lampe et nos nouvelles méthodes Marche et arrêt prenant en charge l'annulation.

Ajouter les commandes de la lampe à l'emplacement 0.

Allumer la lampe, puis éteindre, puis annuler..

puis éteindre de nouveau la lampe, la rallumer et annuler.

Et voici les résultats du test...

```
Fichier Édition Fenêtre Aide Defiert.Entropie
% java ChargeurTeleCommande
Lumière allumée
Lumière éteinte

----- Télécommande -----
[empt 0] teteptremiere.commande.annulation.CommandeAllumerLampe teteptremiere.commande.annulation.CommandeEteindreLampe
[empt 1] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 2] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 3] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 4] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 5] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 6] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[undo] teteptremiere.commande.annulation.CommandeEteindreLampe

Lumière allumée
Lumière éteinte
Lumière allumée

----- Télécommande -----
[empt 0] teteptremiere.commande.annulation.CommandeAllumerLampe teteptremiere.commande.annulation.CommandeEteindreLampe
[empt 1] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 2] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 3] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 4] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 5] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[empt 6] teteptremiere.commande.annulation.PasDeCommande teteptremiere.commande.annulation.PasDeCommande
[undo] teteptremiere.commande.annulation.CommandeAllumerLampe

Lumière éteinte
```

Allumer la lampe, puis l'éteindre.

Voici les commandes de la Lampe

Maintenant, Annuler détient CommandeEteindreLampe, la dernière commande invoquée.

Annuler a été pressé... la méthode annuler() de CommandeEteindreLampe rallume la lumière.

Puis nous l'éteignons et nous la rallumons.

Maintenant, Annuler détient CommandeAllumerLampe, la dernière commande invoquée.

Annuler a été pressé, la lumière s'éteint de nouveau.

Utiliser un état pour implémenter Annuler

L'implémentation d'une annulation pour la Lampe était instructif mais un peu trop facile. En général, il faut gérer un état pour implémenter une annulation. Essayons quelque chose d'un peu plus intéressant, comme la classe propriétaire Ventilateur. Celle-ci dispose de trois méthodes pour régler la vitesse et d'une méthode pour arrêter l'appareil.

Voici le code source du Ventilateur :

```
public class Ventilateur {
    public static final int RAPIDE = 3;
    public static final int MOYEN = 2;
    public static final int LENT = 1;
    public static final int ARRET = 0;
    String localisation;
    int vitesse;
    public Ventilateur(String localisation) {
        this.localisation = localisation;
        vitesse = ARRET;
    }
    public void rapide() {
        vitesse = RAPIDE;
        // code pour régler la vitesse sur rapide
    }

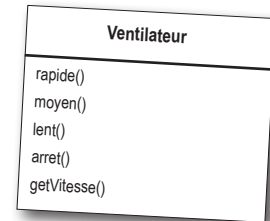
    public void moyen() {
        vitesse = MOYEN;
        // code pour régler la vitesse sur moyen
    }

    public void lent() {
        vitesse = LENT;
        // code pour régler la vitesse sur lent
    }

    public void arret() {
        vitesse = ARRET;
        // code pour arrêter le ventilateur
    }

    public int getVitesse() {
        return vitesse;
    }
}
```

Remarquez que la classe Ventilateur contient un état local qui représente la vitesse de l'appareil.



Mmm, alors pour implémenter correctement Annuler, je devrais prendre en compte la dernière vitesse du ventilateur...

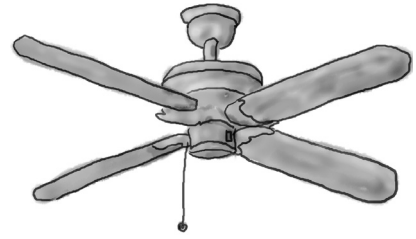
Ces méthodes permettent de régler la vitesse du ventilateur.

La méthode getVitesse() nous permet d'obtenir la vitesse actuelle du ventilateur.



Ajouter l'annulation aux commandes du ventilateur

Attaquons maintenant l'ajout d'une annulation aux différentes commandes du Ventilateur. Pour ce faire, nous devons mémoriser le dernier réglage du ventilateur et, si la méthode `annuler()` est appelée, restaurer le réglage précédent. Voici le code de `CommandeVentilateurRapide` :



```
public class CommandeVentilateurRapide implements Commande {
    Ventilateur ventilateur;
    int derniereVitesse;
```

Nous avons ajouté un état local pour mémoriser la dernière vitesse du ventilateur.

```
    public CommandeVentilateurRapide(Ventilateur ventilateur) {
        this.ventilateur = ventilateur;
    }
```

```
    public void executer() {
        derniereVitesse = ventilateur.getVitesse();
        ventilateur.rapide();
    }
```

Dans `executer()`, avant de modifier la vitesse du ventilateur, nous devons d'abord enregistrer son état précédent, juste au cas où nous devrions annuler nos actions.

```
    public void annuler() {
        if (derniereVitesse == Ventilateur.RAPIDE) {
            ventilateur.rapide();
        } else if (derniereVitesse == Ventilateur.MOYEN) {
            ventilateur.moyen();
        } else if (derniereVitesse == Ventilateur.LENT) {
            ventilateur.lent();
        } else if (derniereVitesse == Ventilateur.ARRET) {
            ventilateur.arret();
        }
    }
}
```

Pour annuler, nous redonnons au ventilateur sa vitesse précédente



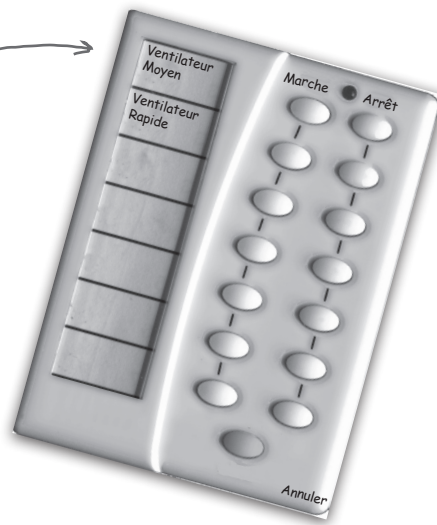
MUSCLEZ VOS NEURONES

Il nous reste trois commandes à écrire : lent, moyen et arrêt. Voyez-vous comment elles sont implémentées ?

Préparons-nous à tester le ventilateur

Il est temps de charger dans notre télécommande les commandes du ventilateur. Nous allons charger le réglage moyen dans le bouton Marche de l'emplacement zéro et le réglage rapide dans celui de l'emplacement un. Les deux boutons Arrêt correspondants contiendront la commande d'arrêt du ventilateur.

Voici notre script de test :



```
public class ChargeurTelecommande {

    public static void main(String[] args) {
        TelecommandeAvecAnnul teleCommande = new TelecommandeAvecAnnul();

        Ventilateur ventilateur = new Ventilateur("Séjour");

        CommandeVentilateurMoyen ventilMoyen =
            new CommandeVentilateurMoyen(ventilateur);
        CommandeVentilateurRapide ventilRapide =
            new CommandeVentilateurRapide(ventilateur);
        CommandeEteindreVentilateur ventilateurEteint =
            new CommandeEteindreVentilateur(ventilateur);

        teleCommande.setCommande(0, ventilMoyen, ventilateurEteint);
        teleCommande.setCommande(1, ventilRapide, ventilateurEteint);

        teleCommande.boutonMarchePresse(0);
        teleCommande.boutonArretPresse(0);
        System.out.println(teleCommande);
        teleCommande.boutonAnnulPresse();

        teleCommande.boutonMarchePresse(1);
        System.out.println(teleCommande);
        teleCommande.boutonAnnulPresse();
    }
}
```

Ici, nous instancions trois commandes :
rapide, moyen et éteint.

Là, nous affectons moyen
à l'emplacement zéro et
rapide à l'emplacement un.
Nous chargeons également les
commandes d'arrêt.

← D'abord, allumer le ventilateur sur moyen.

← Ensuite l'éteindre.

← Annuler ! Il doit revenir à moyen...

← Cette fois, le rallumer sur rapide.

← On annule encore : il doit revenir à moyen.

Tester le ventilateur...

Bien, allumons la télécommande, chargeons-y quelques commandes et appuyons sur quelques boutons !

```
Fichier Edition Fenêtre Aide DéfaireTout!

% java ChargeurTelecommande

Ventilateur sur moyen
Ventilateur éteint

----- Télécommande -----
[empt 0] tete premiere.commande.annulation.CommandeVentilateurMoyen  tete premiere.commande.annulation.CommandeEteindreVentilateur
[empt 1] tete premiere.commande.annulation.CommandeVentilateurRapide  tete premiere.commande.annulation.CommandeEteindreVentilateur
[empt 2] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 3] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 4] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 5] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 6] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[annulation] tete premiere.commande.annulation.CommandeEteindreVentilateur

Ventilateur sur moyen
Ventilateur sur rapide

----- Télécommande -----
[empt 0] tete premiere.commande.annulation.CommandeVentilateurMoyen  tete premiere.commande.annulation.CommandeEteindreVentilateur
[empt 1] tete premiere.commande.annulation.CommandeVentilateurRapide  tete premiere.commande.annulation.CommandeEteindreVentilateur
[empt 2] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 3] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 4] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 5] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[empt 6] tete premiere.commande.annulation.PasDeCommande  tete premiere.commande.annulation.PasDeCommande
[annulation] tete premiere.commande.annulation.CommandeVentilateurRapide

Ventilateur sur moyen
%
```

Voici les commandes de la télécommande...

...et Annuler contient la dernière commande exécutée, CommandeEteindreVentilateur.

Allumer le ventilateur sur moyen, puis l'éteindre.

On annule la dernière commande : il revient à moyen.

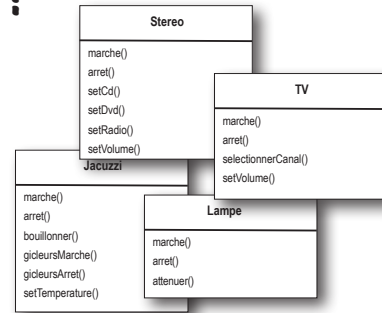
Maintenant, on le règle sur rapide.

Maintenant, rapide est la dernière commande exécutée.

Encore une annulation et le ventilateur revient à la vitesse moyenne.

Toute télécommande a besoin d'un mode groupé !

À quoi sert de posséder une télécommande si on ne peut pas appuyer sur un bouton qui allume les lampes, allume la télé et la stéréo pour jouer un DVD et fait chauffer le jacuzzi ?



Mmm,
notre télécommande
devrait avoir un bouton pour
chaque appareil. Je ne crois
pas que ce soit possible.

Pas si vite, Anne, ce
n'est pas si sûr. Je pense
qu'on peut le faire sans rien
changer à la télécommande !

L'idée de Marie consiste à créer une nouvelle
sorte de Commande qui peut exécuter
d'autres Commandes... et plus d'une !

Joliment bonne idée, non ?

```
public class MacroCommande implements Commande {
    Commande[] commandes;
```

```
    public MacroCommande(Commande[] commandes) {
        this.commandes = commandes;
    }
    public void executer() {
        for (int i = 0; i < commandes.length; i++) {
            commandes[i].executer();
        }
    }
}
```

On prend le tableau
de Commandes et on les stocke
dans la MacroCommande.

Quand la télécommande exécute la macro, ces
commandes sont toutes exécutées une à une.

Utiliser une macro-commande

Voyons les étapes de l'utilisation d'une macro-commande :

- 1 Nous commençons par créer le jeu de commandes que nous voulons placer dans la macro :

```
Lampe lampe = new Lampe("Séjour");
TV tv = new TV("Séjour");
Stereo stereo = new Stereo("Séjour");
Jacuzzi jacuzzi = new Jacuzzi();
CommandeAllumerLampe lampeAllumee = new CommandeAllumerLampe(lampe);
CommandeAllumerStereo stereoAllumee = new CommandeAllumerStereo(stereo);
CommandeAllumerTV tvAllumee = new CommandeAllumerTV(tv);
CommandeAllumerJacuzzi jacuzziAllumee = new CommandeAllumerJacuzzi(jacuzzi);
```

Créer tous les équipements :
une lampe, une télé, une stéréo et un jacuzzi.

Maintenant, créer toutes les commandes Marche pour les contrôler.



À vos crayons

Il nous faut également des commandes pour les boutons Arrêt. Écrivez ici le code de ceux-ci :

- 2 Puis nous créons deux tableaux, un pour les commandes Marche et un pour les commandes Arrêt, et on y charge les commandes correspondantes :

```
Commande[] allumageGroupe = { lampeAllumee, stereoAllumee, tvAllumee, jacuzziAllumee };
Commande[] extinctionGroupe = { lampeEteinte, stereoEteinte, tvEteinte, jacuzziEteint };
MacroCommande macroAllumageGroupe = new MacroCommande(allumageGroupe);
MacroCommande macroExtinctionGroupe = new MacroCommande(extinctionGroupe);
```

Créer un tableau pour les commandes Marche et un pour les commandes Arrêt...

...et créer les deux macros correspondantes pour les contenir.

- 3 Ensuite, nous affectons MacroCommande à un bouton, comme nous le faisons toujours :

```
teleCommande.setCommande(0, macroAllumageGroupe, macroExtinctionGroupe);
```

Affecter la macro-commande à un bouton comme nous le ferions pour tout autre commande.

- 4 Enfin, il suffit d'appuyer sur les boutons et de voir si cela fonctionne.

```
System.out.println(teleCommande);
System.out.println("---Exécution de Macro Marche ---");
teleCommande.boutonMarchePresse(0);
System.out.println("--- Exécution de Macro Arrêt ---");
teleCommande.boutonArrêtPresse(0);
```

Voici le résultat

```
Fichier Edition Fenêtre Aide BeBopALulla
% java ChargeurTelecommande
----- Telecommande -----
[empt 0] tete premiere.commande.groupe.MacroCommande tete premiere.commande.groupe.MacroCommande
[empt 1] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[empt 2] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[empt 3] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[empt 4] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[empt 5] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[empt 6] tete premiere.commande.groupe.PasDeCommande tete premiere.commande.groupe.PasDeCommande
[undo] tete premiere.commande.groupe.PasDeCommande

--- Exécution de Macro Marche ---
Séjour: lumière allumée
Séjour: stéréo allumée
Séjour: la télé est allumée
Séjour: le canal est positionné sur VCR
Le jacuzzi chauffe à 40°
Le jacuzzi bouillonne !

--- Exécution de Macro Arrêt ---
Séjour: lumière éteinte
Séjour: stéréo éteinte
Séjour: la télé est éteinte
Le jacuzzi refroidit à 36°
%
```

Voilà les deux macro-commandes.

Toutes les commandes sont exécutées quand nous appelons la macro Marche

... et de même quand nous invoquons la macro Arrêt. Observez le fonctionnement.



Il ne manque rien à notre MacroCommande sauf une fonctionnalité d'annulation. Quand on appuie sur le bouton Annuler après l'exécution d'une macro-commande, toutes les commandes que la macro a invoquées doivent défaire leurs actions précédentes. Voici le code de MacroCommande. Continuez et implémentez la méthode annuler() :

```
public class MacroCommande implements Commande {
    Commande[] commandes;

    public MacroCommande(Commande[] commandes) {
        this.commandes = commandes;
    }

    public void executer() {
        for (int i = 0; i < commandes.length; i++) {
            commandes[i].executer();
        }
    }

    public void annuler() {
        
    }
}
```

Il n'y a pas de questions stupides

Q: Est-ce que j'ai toujours besoin d'un récepteur ? Pourquoi l'objet de commande ne peut-il pas implémenter les détails de la méthode executer() ?

R: En général, nous nous efforçons d'avoir des objets de commande « passifs » qui invoquent simplement une action sur un récepteur, mais il existe de nombreux exemples d'objets de commande « intelligents » qui implémentent la plus grande partie, sinon la totalité, de la logique nécessaire pour répondre à une requête. Bien sûr, vous pouvez le faire. Souvenez-vous seulement que vous n'aurez plus le même niveau de découplage entre l'invocateur et le récepteur et que vous ne pourrez plus paramétrer vos commandes avec des récepteurs.

Q: Comment puis-je implémenter un historique des opérations d'annulation ? Autrement dit, que faire si je veux pouvoir appuyer sur le bouton Annuler plusieurs fois ?

R: Excellente question ! C'est assez facile en réalité. Au lieu de conserver seulement une référence à la dernière Commande exécutée, vous mémorisez une pile de commandes. Puis, à chaque appui sur le bouton Annuler, votre invocateur dépile le premier élément et appelle sa méthode annuler().

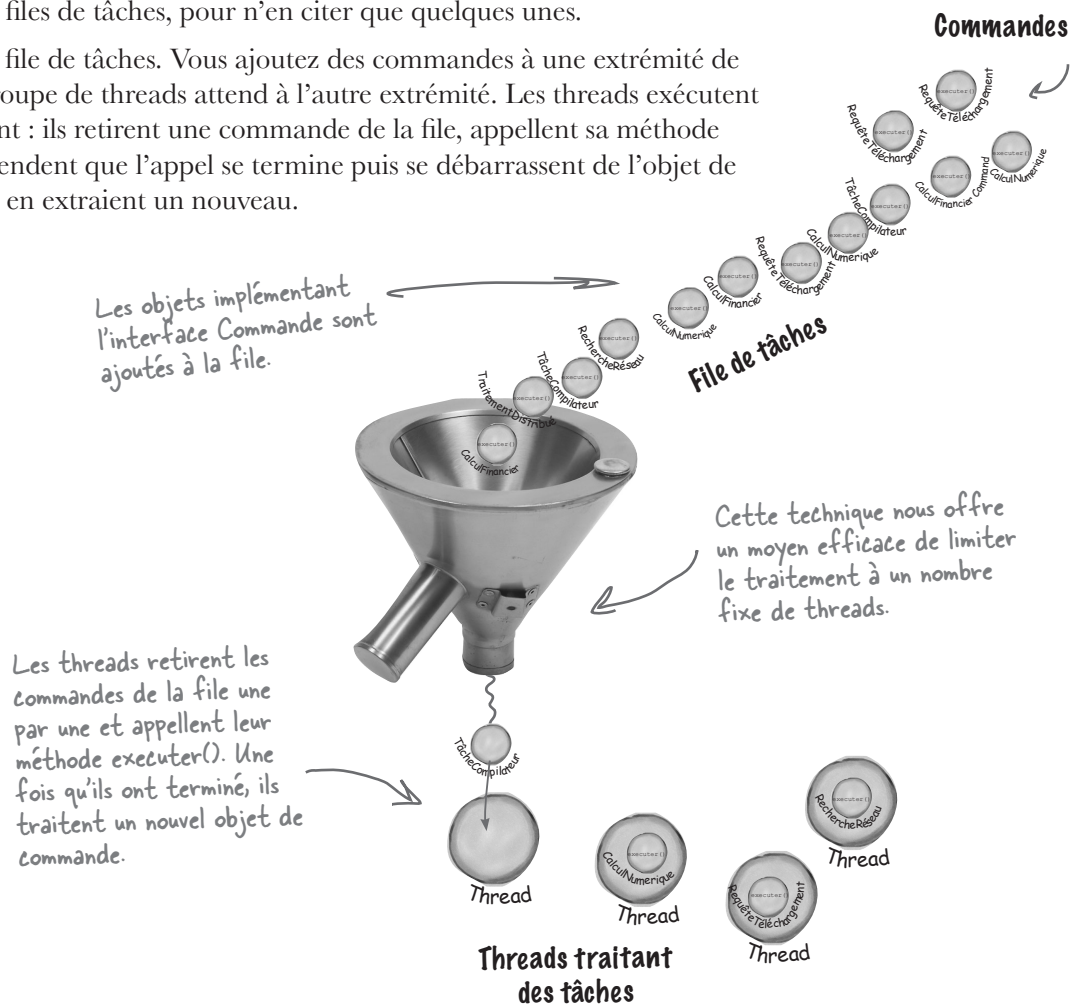
Q: Est-ce que je pourrais me contenter d'implémenter le mode groupé comme une Commande en créant une CommandeGroupe et en plaçant les appels pour exécuter les autres Commandes dans la méthode executer() de CommandeGroupe ?

R: Oui, mais cela reviendrait en substance à « coder en dur » le mode groupé dans CommandeGroupe. Pourquoi chercher les ennuis ? Avec MacroCommande, vous pouvez décider dynamiquement quelles Commandes vous voulez placer dans la CommandeGroupe, et vous disposez de plus de souplesse pour exécuter des MacroCommandes. En général, MacroCommande constitue une solution plus élégante et exige d'écrire moins de code.

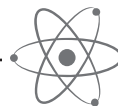
Autre utilisation du pattern Commande : files de requêtes

Le pattern Commande nous fournit un moyen de grouper l'intégralité d'un traitement (un récepteur et un ensemble d'actions) et de le transmettre comme un objet standard. Le traitement lui-même peut être invoqué bien après qu'une application cliente a créé l'objet de commande. En fait, il peut même être invoqué par un thread différent. Nous pouvons nous inspirer de ce scénario pour de nombreuses applications utiles comme des ordonnanceurs, des pools de threads et des files de tâches, pour n'en citer que quelques unes.

Imaginez une file de tâches. Vous ajoutez des commandes à une extrémité de la file et un groupe de threads attend à l'autre extrémité. Les threads exécutent le script suivant : ils retirent une commande de la file, appellent sa méthode `executer()`, attendent que l'appel se termine puis se débarrassent de l'objet de commande et en extraient un nouveau.



Notez que les classes de la file de tâches sont totalement découplées des objets qui effectuent le traitement. Un thread peut exécuter un calcul financier pendant une minute et télécharger quelque chose la minute suivante. Les objets de la file ne s'en soucient pas : ils se bornent à extraire les commandes et à appeler `executer()`. De même, tant que vous placerez dans la file des objets qui implémentent le pattern Commande, votre méthode `executer()` sera invoquée quand un thread sera disponible.



MUSCLEZ VOS NEURONES

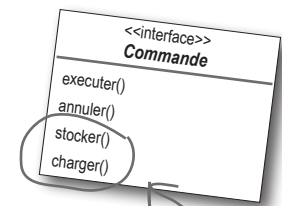
Comment un serveur web pourrait-il utiliser une telle file ? À quelles autres applications pourrait-on penser ?

Autre utilisation du pattern Commande : journaliser les requêtes

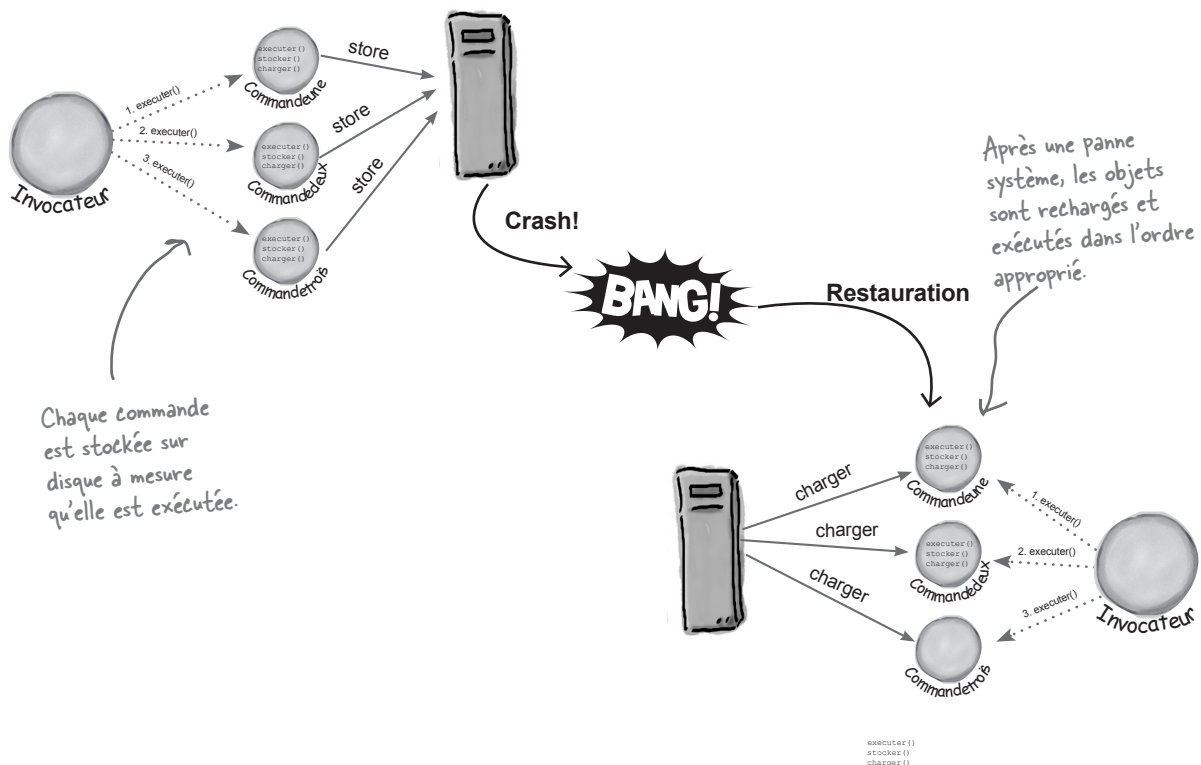
La sémantique de certaines applications nécessite de journaliser toutes les actions et de pouvoir les invoquer afin de réaliser une reprise sur incident. Le pattern Commande peut prendre en charge cette sémantique en ajoutant deux méthodes : `stocker()` et `charger()`. En Java, nous pouvons utiliser la sérialisation pour implémenter ces méthodes, mais les réserves habituelles sur l'utilisation de la sérialisation pour la persistance continuent à s'appliquer.

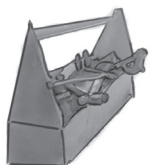
Comment cette technique fonctionne-t-elle ? À mesure que nous exécutons les commandes, nous en stockons l'historique sur disque. Lorsqu'une panne se produit, nous rechargeons les objets de commande et nous invoquons leurs méthodes `executer()` en un seul lot et dans l'ordre.

Ce type de journalisation n'aurait aucun sens pour une télécommande. Toutefois, il existe de nombreuses applications qui invoquent des actions sur de gros volumes de données qui ne peuvent être sauvegardés rapidement chaque fois qu'une modification est apportée. Avec un mécanisme de journalisation, nous pouvons sauvegarder toutes les opérations postérieures au dernier point de contrôle, et, s'il y a une panne système, appliquer ces opérations à partir du point de contrôle. Prenons l'exemple d'un tableur : nous pourrions implémenter la récupération sur panne en journalisant les actions dans la feuille de calcul au lieu d'écrire une copie de celle-ci sur disque à chaque modification. Dans des applications plus avancées, ces techniques peuvent être étendues pour appliquer des ensembles d'opérations de manière transactionnelle, de sorte que toutes les opérations se terminent ou aucune.



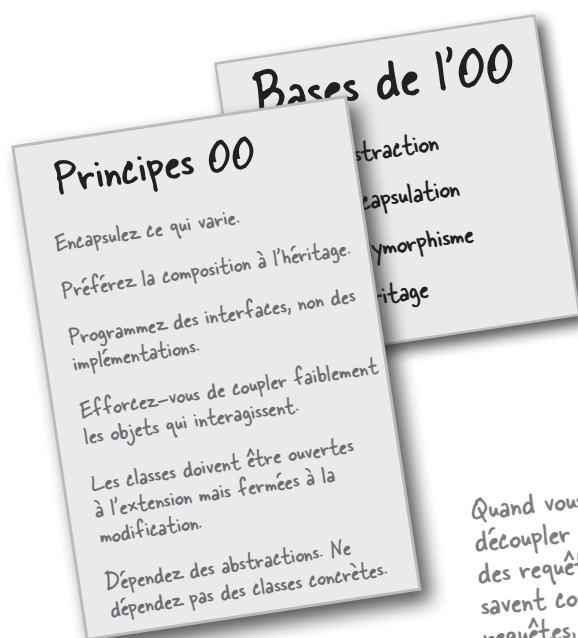
Nous ajoutons deux méthodes pour la journalisation.



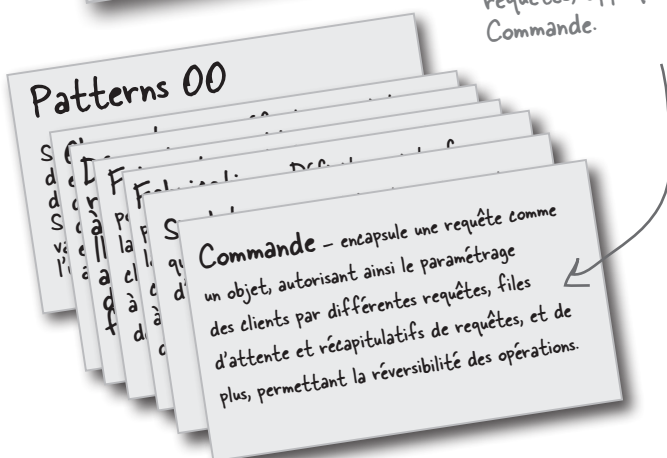


Votre boîte à outils de concepteur

Votre boîte à outils commence à s'alourdir ! Dans ce chapitre, nous lui avons ajouté un pattern qui nous permet d'encapsuler des méthodes dans des objets de commande, de les mémoriser, de les transmettre et de les invoquer quand nous en avons besoin



Quand vous avez besoin de découpler un objet émettant des requêtes des objets qui savent comment satisfaire ces requêtes, appliquez le pattern Commande.

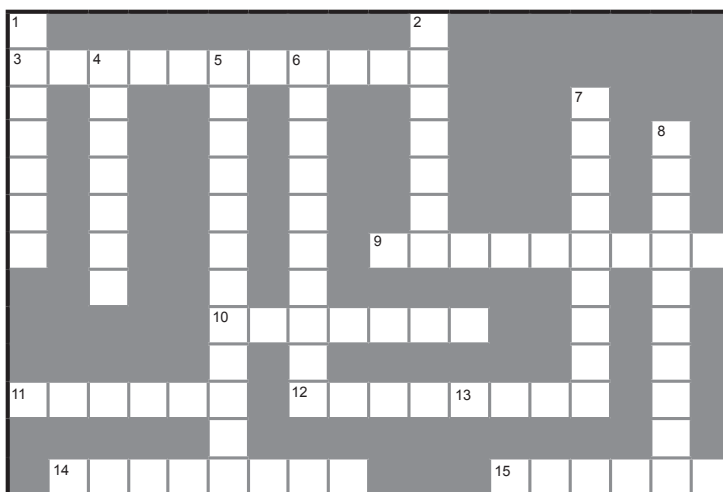


POINTS D'IMPACT

- Le pattern Commande découple un objet émettant une requête de celui qui sait comment l'exécuter.
- Un objet Commande est au centre du mécanisme de découplage et encapsule un récepteur avec une action (ou un ensemble d'actions).
- Un invocateur requiert un objet Commande en appelant sa méthode `executer()`, laquelle invoque ces actions sur le récepteur.
- Les invocateurs peuvent être paramétrés par des Commandes, même dynamiquement lors de l'exécution.
- Les Commandes peuvent prendre en charge l'annulation en implémentant une méthode `annuler()` qui restitue à l'objet l'état qui était le sien avant le dernier appel de la méthode `executer()`.
- Les MacroCommandes sont une simple extension de Commande qui permet d'invoquer plusieurs commandes. Les macro-commandes supportent également l'annulation.
- Dans la pratique, il n'est pas rare que des objets Commande « intelligents » implémentent eux-mêmes le traitement de la requête au lieu de la déléguer à un récepteur.
- On peut également utiliser des Commandes pour implémenter des mécanismes de journalisation ou des systèmes transactionnels.



Il est temps de souffler un peu et de prendre le temps d'assimiler.
Un autre mots-croisés ; toutes les solutions sont dans ce chapitre.



Horizontalement

3. Notre ville préférée.
9. Objet qui sait comment faire les choses.
10. À Objectville, elles sont automatisées.
11. Le personnage de la première page de ce chapitre en est un.
12. Objet _____ mérite une mention honorable.
13. Notre langage favori.
14. Découplée du chef.
15. Ni serveuse, ni serveur.

Verticalement

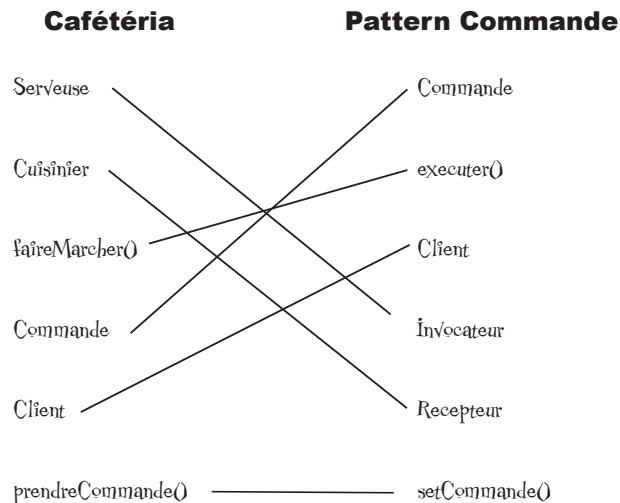
1. Sujet de ce chapitre.
2. Encapsulée dans une commande.
4. Peut bouillonner.
5. Ce que Maisons de Rêve vous demande d'implémenter.
6. Commande sert à l'encapsuler.
7. La serveuse et le chef y travaillent.
8. Commande permet d'en implémenter une.



Solutions des exercices

Qui fait quoi ?

Reliez les objets et les méthodes de la Cafétéria à leurs homologues dans le pattern Commande.



À vos crayons

```
public class GarageDoorOpenCommand implements Command {
    public class CommandeOuvrirPorteGarage implements Commande {
        PorteGarage porteGarage;
        public CommandeOuvrirPorteGarage(PorteGarage porteGarage) {
            this.porteGarage = porteGarage;
        }
        public void executer() {
            porteGarage.ouvrir();
        }
    }
}
```

```
Fichier Edition Fenêtre Aide SandwichAuJambon
%java TestTelecommande
Lumière allumée
Porte garage ouverte
%
```




Solutions des exercices



Exercice

Écrivez la méthode `annuler()` pour `MacroCommande`

```
public class MacroCommande implements Commande {

    Commande[] commandes;

    public MacroCommande(Commande[] commandes) {
        this.commandes = commandes;
    }

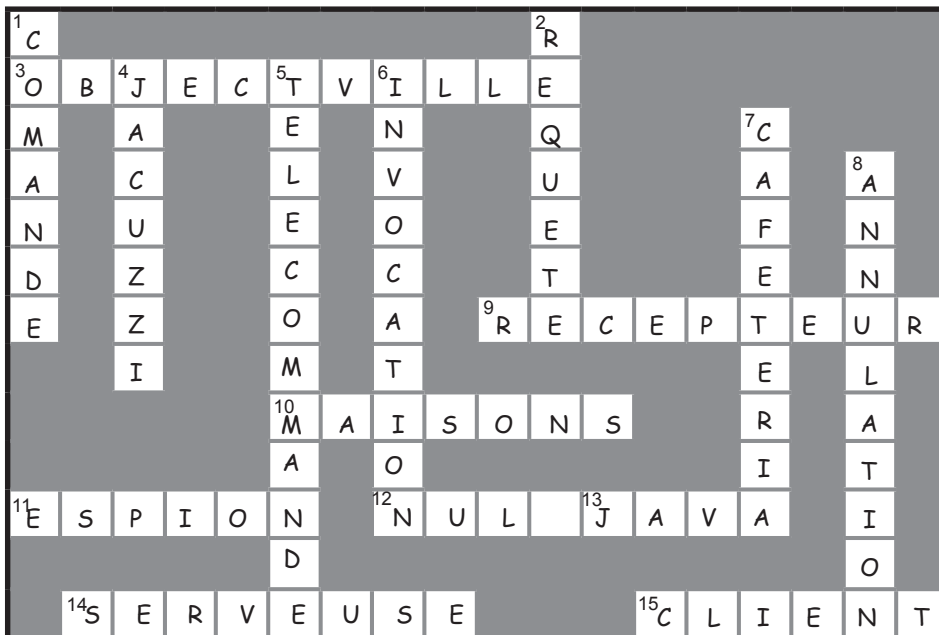
    public void executer() {
        for (int i = 0; i < commandes.length; i++) {
            commandes[i].executer();
        }
    }

    public void annuler() {
        for (int i = 0; i < commandes.length; i++) {
            commandes[i].annuler();
        }
    }
}
```



À vos crayons

```
CommandeEteindreLampe lampeEteinte = new CommandeEteindreLampe(lampe);
CommandeEteindreStereo stereoEteinte = new CommandeEteindreStereo(stereo);
CommandeEteindreTV tvEteinte = CommandeEteindreTV (tv);
CommandeEteindrejacuzzi jacuzziEteint = new CommandeEteindrejacuzzi (jacuzzi);
```



7 les patterns Adaptateur et Façade

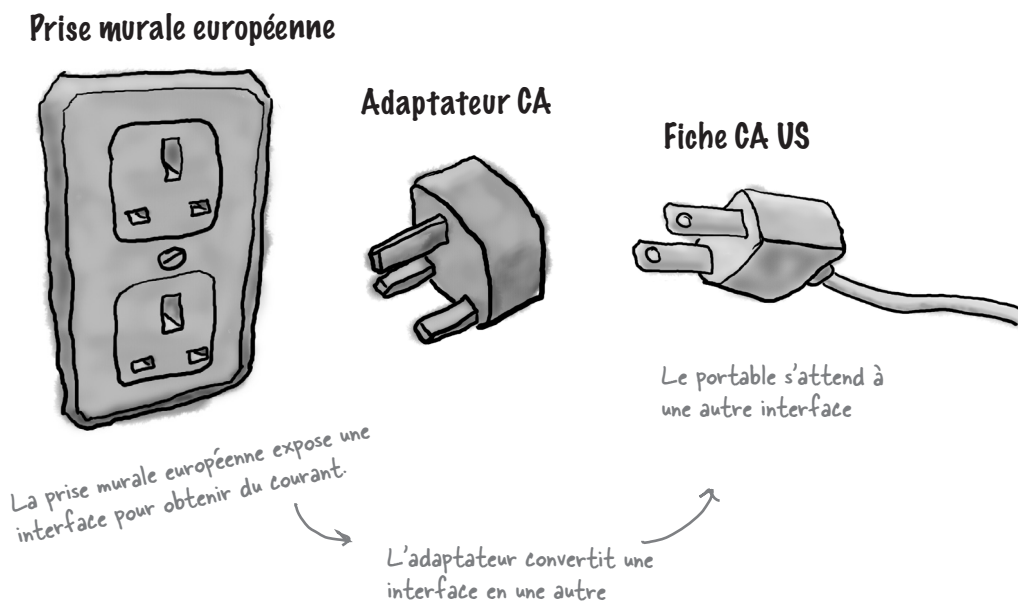
Savoir s'adapter



Dans ce chapitre, nous allons entreprendre des choses impossibles, comme faire entrer une cheville ronde dans un trou carré. Cela vous semble impossible ? Plus maintenant avec les design patterns. Vous souvenez-vous du pattern Décorateur ? Nous avons **enveloppé des objets** pour leur attribuer de nouvelles responsabilités. Nous allons recommencer, mais cette fois avec un objectif différent : faire ressembler leurs interfaces à quelque chose qu'elles ne sont pas. Pourquoi donc ? Pour pouvoir adapter une conception qui attend une interface donnée à une classe qui implémente une interface différente. Et ce n'est pas tout. Pendant que nous y sommes, nous allons étudier un autre pattern qui enveloppe des objets pour simplifier leur interface.

Nous sommes entourés d'adaptateurs

Vous n'éprouverez aucune difficulté à comprendre ce qu'est un adaptateur OO parce que le monde réel en est plein. Prenons un exemple. Avez-vous déjà eu besoin d'utiliser dans un pays d'Europe un ordinateur portable fabriqué aux États-Unis ? Si oui, vous avez sans doute eu besoin d'un adaptateur CA...



Vous savez comment l'adaptateur opère : il se place entre la fiche du portable et la prise européenne et sa tâche consiste à s'adapter à celle-ci pour que vous puissiez y brancher le portable et recevoir du courant. Ou, pour voir le problème autrement, l'adaptateur transforme l'interface de la prise pour offrir celle que le portable attend.

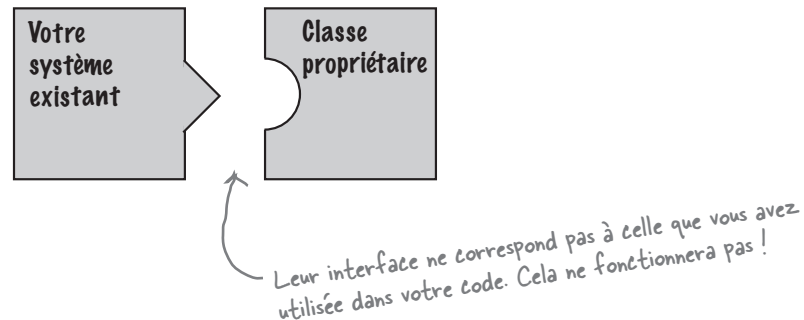
Combien d'autres adaptateurs du monde réel pouvez-vous citer ?

Certains adaptateurs CA sont simples : ils se bornent à modifier la forme de la prise pour qu'elle corresponde à celle de votre fiche et laissent passer directement le courant alternatif. D'autres sont plus complexes et peuvent augmenter ou diminuer le voltage en fonction des besoins de votre machine.

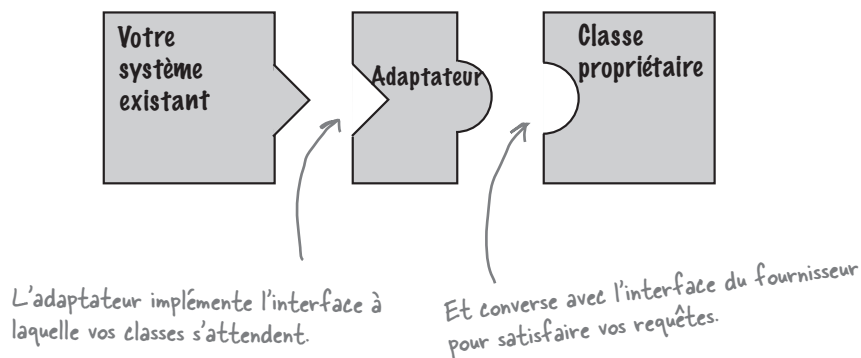
Voilà pour le monde réel, mais qu'en est-il des adaptateurs orientés objet ? Eh bien nos adaptateurs OO jouent le même rôle que leurs homologues du monde réel : ils prennent une interface et l'adaptent de manière à ce qu'elle corresponde à celle que le client attend.

Adaptateurs orientés objet

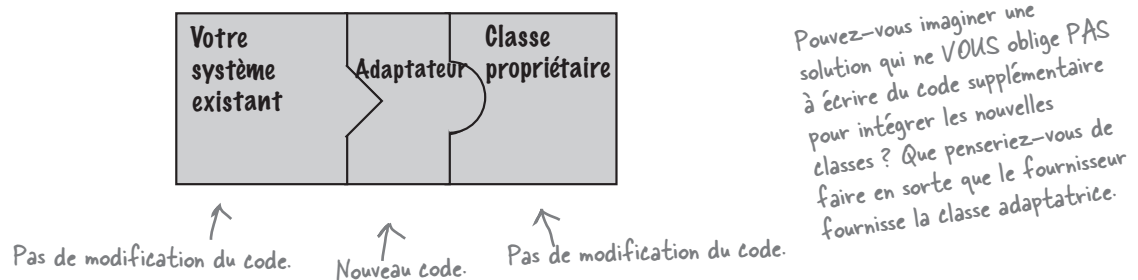
Disons que vous possédez un système logiciel dans lequel vous voulez charger une nouvelle bibliothèque de classe d'un fournisseur donné, mais que ce nouveau fournisseur a conçu ses interfaces différemment du précédent :



Bien. Vous vous refusez à résoudre le problème en modifiant votre code existant (et vous ne pouvez pas récrire celui du fournisseur). Que faites-vous ? Eh bien vous pouvez écrire une classe qui adapte l'interface du nouveau fournisseur pour qu'elle corresponde à celle dont vous avez besoin.



L'adaptateur joue le rôle d'intermédiaire : il reçoit les requêtes du client et les convertit en requêtes compréhensibles par les classes du fournisseur.



Si ça marche comme un canard et si ça cancanne comme un canard, ce doit ~~peut~~ être un canard dindon enveloppé dans un adaptateur pour canard...

Il est temps de voir un adaptateur en action. Vous souvenez-vous de nos canards du chapitre 1 ? Examinons une version légèrement simplifiée des interfaces et des classes Canard :



```
public interface Canard {  
    public void cancaner();  
    public void voler();  
}
```

Cette fois, nos canards implémentent une interface Canard qui leur permet de voler et de cancaner.

Voici une sous-classe de Canard, le Colvert.

```
public class Colvert implements Canard {  
    public void cancaner() {  
        System.out.println("Coincoin");  
    }  
  
    public void voler() {  
        System.out.println("Je vole");  
    }  
}
```

Implémentations simples : le canard se contente d'afficher ce qu'il fait.

Il est temps de faire connaissance avec notre tout dernier volatile :

```
public interface Dindon {  
    public void glouglouter();  
    public void voler();  
}
```

Les dindons ne cancanent pas, ils glougloutent.

Les dindons ne volent pas, sauf parfois sur de courtes distances.


```

public class DindonSauvage implements Dindon {
    public void glouglouter() {
        System.out.println("Glouglou");
    }
    public void voler() {
        System.out.println("Je ne vole pas loin");
    }
}

```

Voici une implémentation concrète de Dindon ; comme le canard, ce dindon se contente d'afficher ses actions.

Disons maintenant que vous êtes à court d'objets Canard et que vous aimeriez utiliser des objets Dindon à la place. De toute évidence, il est impossible d'utiliser les dindons directement parce que leur interface est différente.

Nous allons donc écrire un Adaptateur :



Code à la loupe

```

public class AdaptateurDindon implements Canard {
    Dindon dindon;

    public AdaptateurDindon(Dindon dindon) {
        this.dindon = dindon;
    }

    public void cancaner() {
        dindon.glouglouter();
    }

    public void voler() {
        for(int i=0; i < 5; i++) {
            dindon.voler();
        }
    }
}

```

Vous devez d'abord implémenter l'interface du type auquel vous vous adaptez. C'est l'interface que votre client s'attend à voir.

Puis vous devez obtenir une référence à l'objet que vous adaptez ; pour ce faire, nous utilisons le constructeur.

Maintenant, nous devons implémenter toutes les méthodes de l'interface. La traduction de `cancaner()` entre les classes est facile : il suffit d'appeler la méthode `glouglouter()`.

Même si les deux interfaces possèdent une méthode `voler()`, les Dindons ne volent que sur de courtes distances – ils ne peuvent pas voler très longtemps comme les Canards. Pour assurer la correspondance entre la méthode `voler()` d'un Canard et celle d'un Dindon, nous devons appeler cinq fois la méthode `voler()` du Dindon pour simuler son comportement.

Testons l'adaptateur

Il ne nous faut plus qu'un peu de code pour tester notre adaptateur : créons un Canard... et un Dindon.

```
public class TestCanard {
    public static void main(String[] args) {
        Colvert canard = new Colvert();

        DindonSauvage dindon = new DindonSauvage();
        Canard adaptateurDindon = new AdaptateurDindon(dindon);

        System.out.println("Dindon dit...");
        dindon.glouglouter();
        dindon.voler();

        System.out.println("\nCanard dit...");
        testerCanard(canard);

        System.out.println("\nAdaptateurDindon dit...");
        testerCanard(adaptateurDindon);
    }

    static void testerCanard(Canard canard) {
        canard.cancaner();
        canard.voler();
    }
}
```

Créons un Canard...

et un Dindon.

Puis enveloppons le dindon dans un AdaptateurDindon qui le fait ressembler à un Canard.

Testons maintenant le canard en appelant la méthode testerCanard() qui attend un objet Canard.

Puis testons le Dindon : qu'il glougloute et qu'il vole.

Et maintenant le grand test : nous essayons de faire passer le dindon pour un canard...

Voici notre méthode testerCanard() : elle reçoit un canard et appelle ses méthodes cancaner() et voler().

Exécution du test

```
Fichier Fenêtre Édition Aide Canarder
%java TestCanard
Dindon dit...
Glouglou
Je ne vole pas loin
Canard dit...
Coincoin
Je vole
AdaptateurDindon dit...
Glouglou
Je ne vole pas loin
Je ne vole pas loin
Je ne vole pas loin
Je ne vole pas loin
Je ne vole pas loin
```

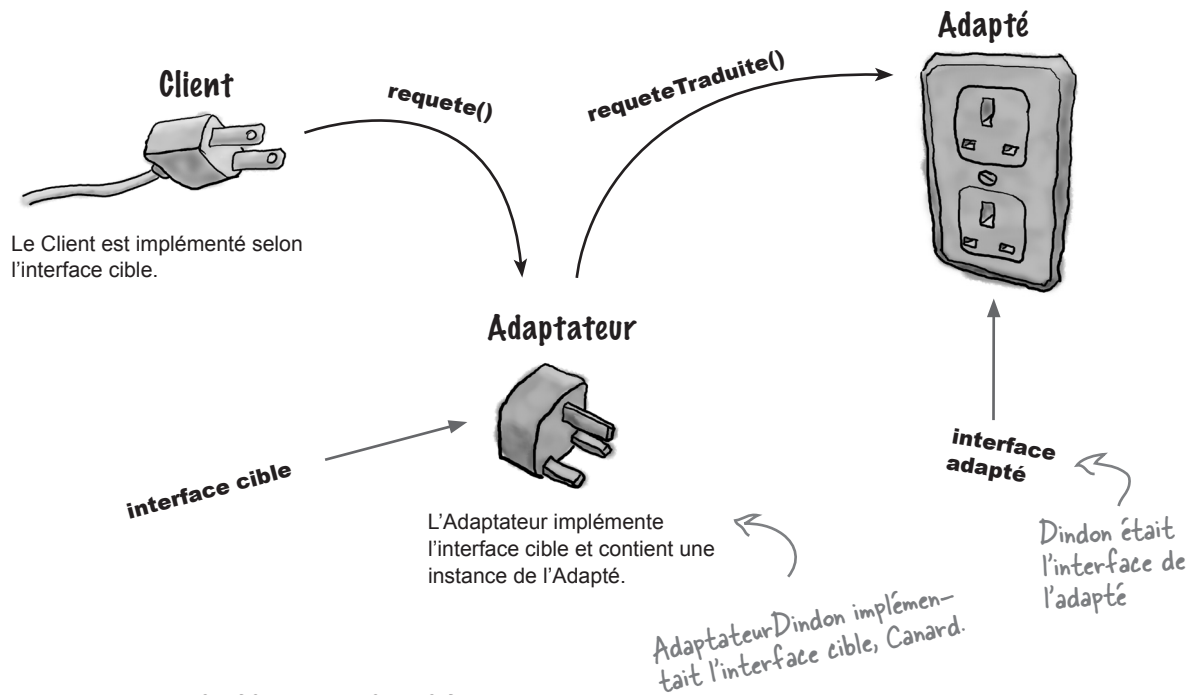
Le Dindon glougloute et vole sur une courte distance.

Le Canard cancaner et vole exactement comme prévu.

Et l'adaptateur glougloute quand cancaner() est appelée et vole cinq fois quand voler() est appelée. La méthode testerCanard() ne s'est pas rendu compte qu'elle avait un dindon déguisé en canard !

Le pattern Adaptateur expliqué

Maintenant que nous avons une idée de ce qu'est un Adaptateur, revenons en arrière et regardons de nouveau tous ses constituants.



Voici comment le Client utilise l'Adaptateur

- ❶ Le client envoie une requête à l'adaptateur en appelant dessus une méthode en utilisant l'interface cible.
- ❷ L'adaptateur traduit cette requête en un ou plusieurs appels à l'adapté en utilisant l'interface de l'adapté.
- ❸ Le client reçoit les résultats de l'appel et ne sait jamais que c'est un adaptateur qui effectue la traduction.

Notez que le Client et l'Adapté sont découplés – aucun des deux ne connaît l'autre.

À vos crayons



Imaginez que nous ayons également besoin d'un Adaptateur qui convertit un Canard en Dindon. Appelons-le AdaptateurCanard. Écrivez cette classe :

Comment avez-vous traité la méthode voler() (après tout nous savons que les canards volent plus loin que les dindons ? Regardez notre solution à la fin de ce chapitre. Avez-vous trouvé un meilleur moyen ?

Il n'y a pas de questions stupides

Q: Quel volume d'« adaptation » un adaptateur doit-il effectuer ? On dirait que si je dois implémenter une interface Cible importante, cela implique une somme **IMPORTANTE** de travail.

R: Certainement. Le travail d'implémentation d'un adaptateur est exactement proportionnel à la taille de l'interface que vous devez prendre en charge en tant qu'interface Cible. Mais vous disposez de deux options. Vous pouvez retravailler tous vos appels à l'interface côté client, ce qui entraînerait beaucoup de recherches et de modifications du code. Ou bien vous pouvez fournir bien proprement une seule classe qui encapsule tous les changements.

Q: Un adaptateur enveloppe-t-il toujours une classe et une seule ?

R: Le rôle du pattern Adaptateur consiste à convertir une interface en une autre. Si la plupart des exemples du pattern montrent un adaptateur enveloppant un adapté, nous savons tous les deux que les choses sont un peu plus complexes dans la réalité. Vous pouvez donc rencontrer des situations dans laquelle un adaptateur nécessite deux ou plusieurs adaptés pour implémenter l'interface Cible. Ceci est lié à un autre pattern nommé Façade et il est fréquent de confondre les deux. Rappelez-moi d'en reparler quand nous aborderons les façades plus loin dans ce chapitre.

Q: Que se passe-t-il si mon système comprend plusieurs parties et que les plus anciennes attendent l'interface de l'ancien fournisseur mais que nous en ayons déjà écrit d'autres qui utiliseront celle du nouveau ? Cela va poser des problèmes si nous utilisons un adaptateur ici et l'interface non enveloppée là. Ne vaudrait-il pas mieux que je réécrive mon ancien code et que j'oublie l'adaptateur ?

R: Pas nécessairement. Vous pouvez par exemple créer un Adaptateur bidirectionnel qui prenne en charge les deux interfaces. Pour créer un Adaptateur bidirectionnel, il suffit d'implémenter les deux interfaces concernées, afin que l'adaptateur puisse se comporter comme l'ancienne interface ou comme la nouvelle.

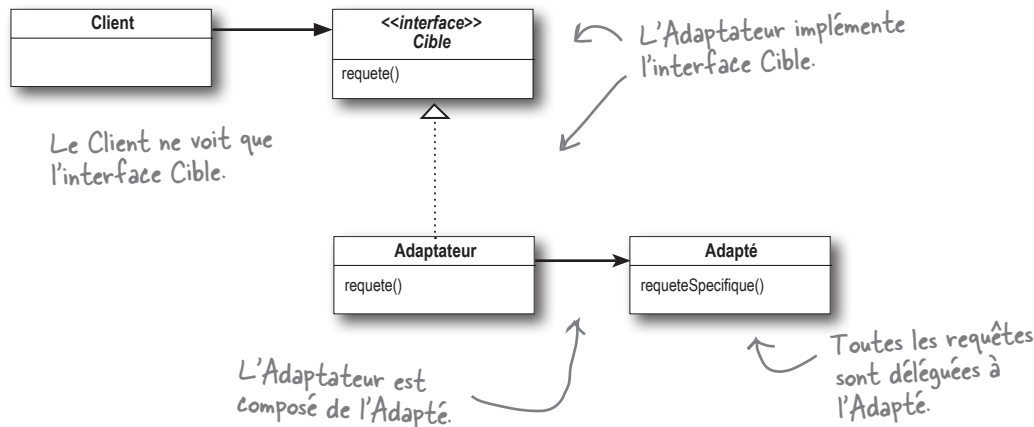
Le pattern Adaptateur : définition

Assez de canards, de dindons et d'adaptateurs électriques pour aujourd'hui. Revenons à la réalité et voyons la définition officielle du pattern Adaptateur :

Le pattern Adaptateur convertit l'interface d'une classe en une autre conforme à celle du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.

Maintenant, nous savons que ce pattern nous permet d'utiliser un client dont l'interface est incompatible en créant un Adaptateur qui exécute la conversion, ce qui a pour effet de découpler le client de l'interface implémentée. Si nous nous attendons à ce que l'interface doive changer, l'adaptateur encapsule ce changement pour qu'on n'ait pas besoin de modifier le client chaque fois qu'il doit fonctionner avec une interface différente.

Après avoir eu un aperçu du comportement de ce pattern lors de l'exécution, jetons également un coup d'œil au diagramme de classes :



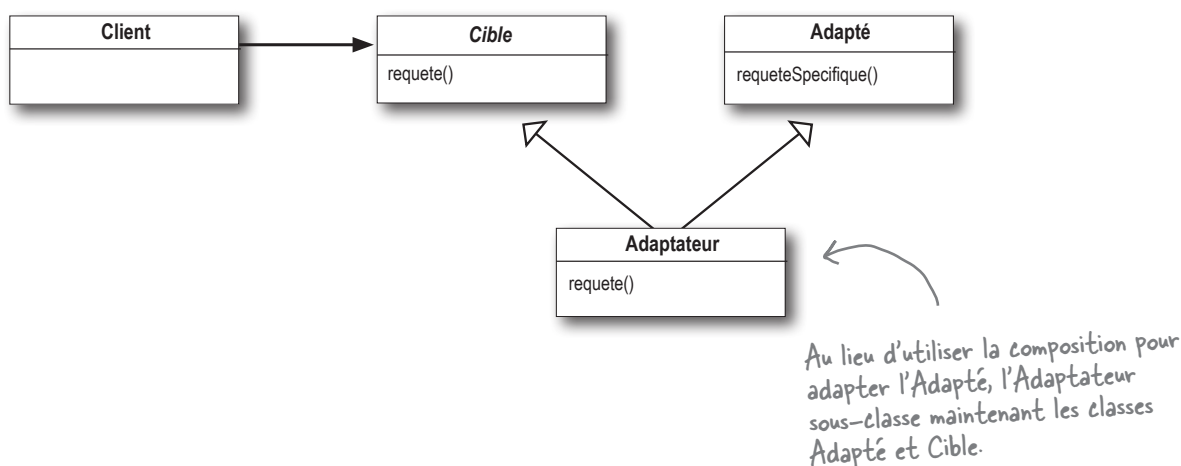
Le pattern Adaptateur est plein de bons principes de conception OO : remarquez l'usage de la composition des objets pour envelopper l'adapté dans une interface modifiée. Cette approche présente l'avantage supplémentaire de permettre d'utiliser un adaptateur avec n'importe quelle sous-classe de l'adapté.

Observez également comment le pattern lie le client à une interface, non à une implémentation : nous pourrions utiliser plusieurs adaptateurs, chacun deux convertissant un ensemble de classes différent. Ou bien nous pourrions ajouter de nouvelles implémentations après, tant qu'elles adhèrent à l'interface Cible.

Adaptateurs d'objet et adaptateurs de classe

Si nous avons bien défini le pattern, nous ne vous avons pas encore tout dit. Il existe en réalité deux sortes d'adaptateurs : les adaptateurs d'objet et les adaptateurs de classe. Nous n'avons vu jusqu'ici que des adaptateurs d'objet, et le diagramme de classes de la page précédente est le diagramme d'un adaptateur d'objet.

Qu'est-ce donc qu'un adaptateur de classe et pourquoi ne pas en avoir parlé ? Parce que l'héritage multiple est nécessaire pour l'implémenter, ce qui est impossible en Java. Mais cela ne veut pas dire que vous n'aurez pas besoin un jour d'adaptateurs de classe si votre langage favori autorise l'héritage multiple ! Voici le diagramme de classes pour l'héritage multiple :



Cela vous rappelle quelque chose ? Exactement. La seule différence est que l'adaptateur de classe sous-classe la Cible et l'Adapté, tandis que, dans le cas de l'adaptateur d'objet, nous utilisons la composition pour transmettre des requêtes à un Adapté.



MUSCLEZ VOS NEURONES

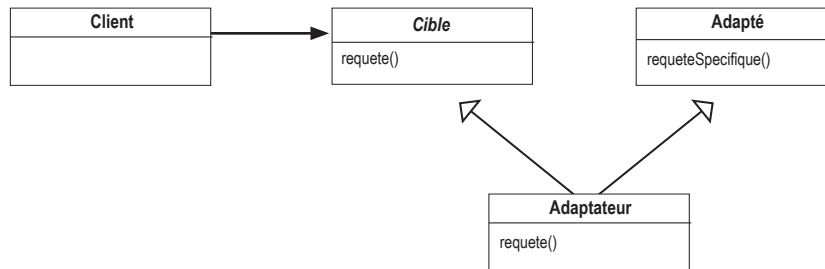
Les adaptateurs d'objet et les adaptateurs de classe utilisent deux techniques différentes pour adapter l'adapté (composition vs. héritage). Comment ces différences d'implémentation affectent-elles la souplesse de l'adaptateur ?



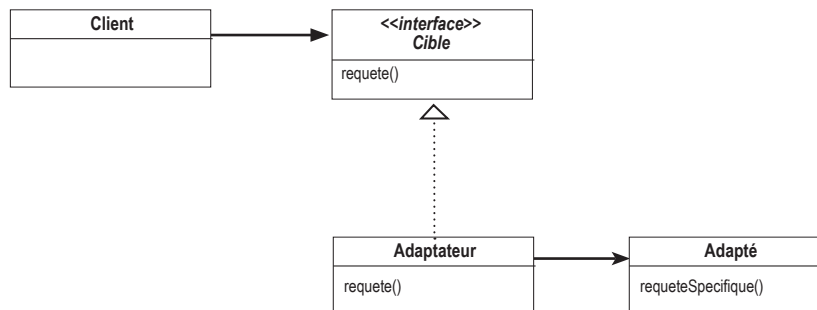
Le frigo

Votre tâche consiste à prendre les canards et les dindons magnétiques et à les poser sur la partie du diagramme qui décrit le rôle joué par chaque volatile dans notre précédent exemple. (Essayez de ne pas tourner les pages). Puis ajoutez vos propres annotations pour documenter le fonctionnement.

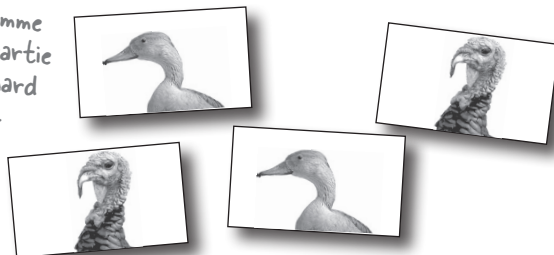
Adaptateur de classe



Adaptateur d'objet



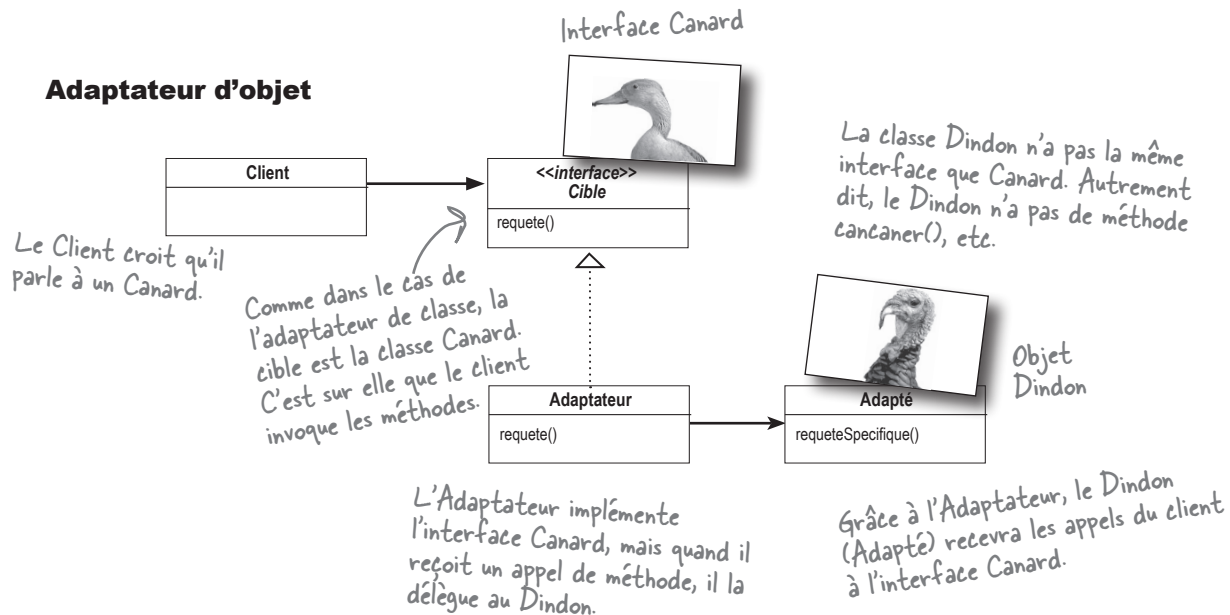
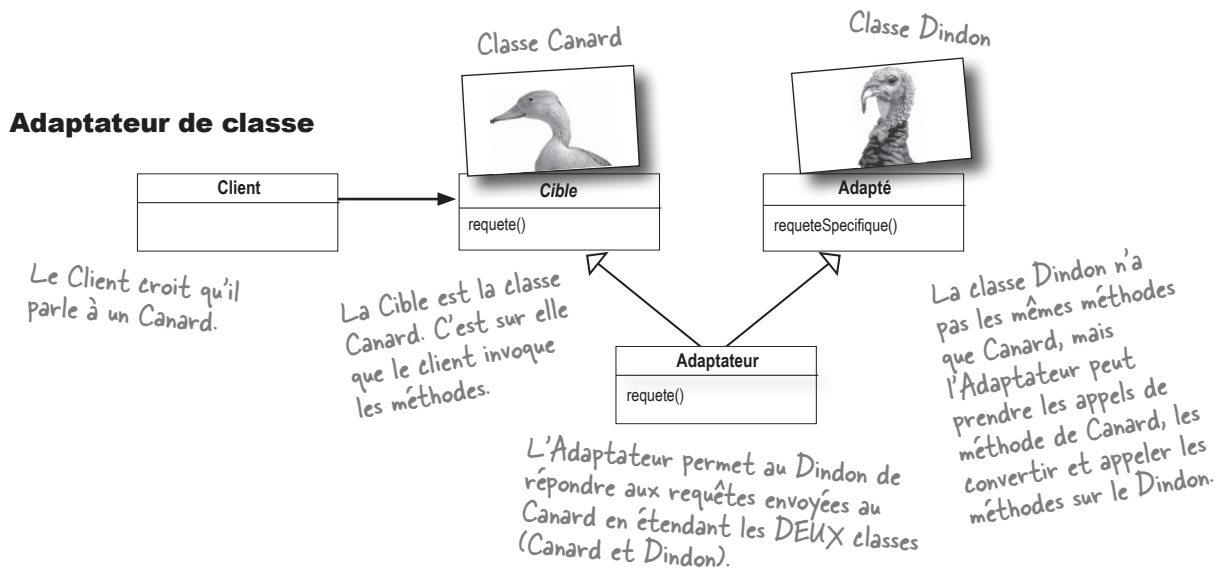
Posez ces aimants sur le diagramme de classes pour indiquer quelle partie du diagramme représente le Canard et laquelle représente le Dindon.



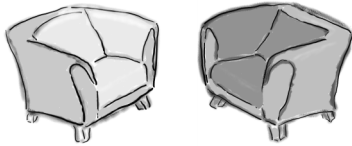


Solution du jeu du frigo

Note : l'adaptateur de classe s'appuyant sur l'héritage multiple, vous ne pouvez pas l'implémenter en Java...



Face à face :



Le face à face de ce soir : **l'Adaptateur d'objet et l'Adaptateur de classe.**

Adaptateur d'objet

Comme j'utilise la composition, j'ai une longueur d'avance. Je suis capable d'adapter une classe Adapté, mais aussi n'importe laquelle de ses sous-classes.

Dans le monde où je vis, nous préférons la composition à l'héritage. Vous économisez peut-être quelques lignes de code, mais je ne fais rien d'autre qu'en ajouter un petit peu pour déléguer à l'adapté. Nous aimons bien conserver de la souplesse.

Parce que vous souciez d'un seul petit objet ? Vous pourriez être capable de redéfinir rapidement une méthode, mais tout comportement que j'ajoute au code adaptateur fonctionne avec ma classe Adapté et toutes ses sous-classes.

Oh, ça va, lâchez-moi les baskets, je n'ai qu'à composer avec la sous-classe pour que ça fonctionne.

Vous voulez voir quelque chose de pas terrible ? Regardez-vous dans une glace !

Adaptateur de classe

C'est vrai, cela me pose des problèmes parce que je suis lié à une classe Adapté spécifique. Mais j'ai un énorme avantage : je ne suis pas obligé de réimplémenter mon Adapté entier. Je peux aussi redéfinir son comportement si j'en ai besoin, parce que je me contente de sous-classer.

Souple, peut-être. Efficace, sûrement pas. Quand on utilise un adaptateur de classe, il n'y a que moi. Il n'y a pas un adaptateur ET un adapté.

Ouais, et si une sous-classe de l'adapté ajoute un nouveau comportement. Qu'est-ce qui se passe ?

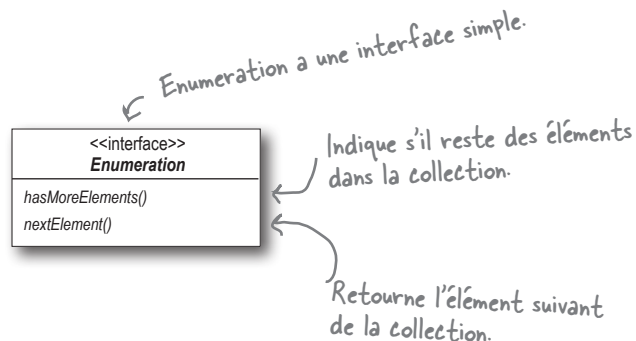
Pas terrible...

Adaptateurs du monde réel

Jetons un coup d'œil à une utilisation d'un Adaptateur simple dans le monde réel (quelque chose de plus sérieux que les canards en tous cas)...

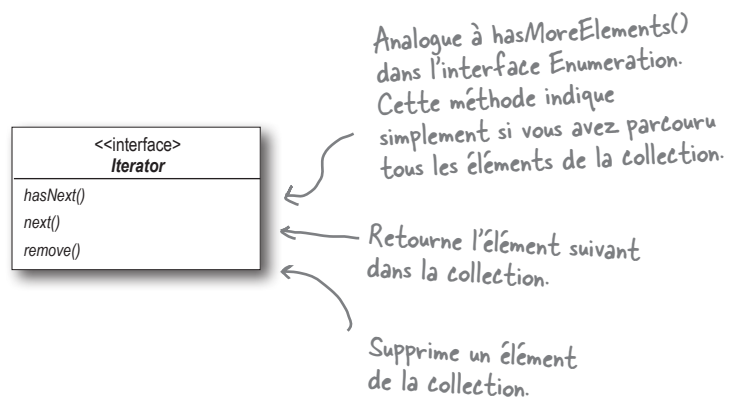
L'ancien monde : Enumeration

Si vous programmez en Java depuis un moment, vous vous souvenez probablement que les premiers types collection (Vector, Stack, Hashtable et quelques autres) implémentent une méthode `elements()`, qui retourne une Enumeration. L'interface Enumeration vous permet de parcourir les éléments d'une collection sans connaître en détail la façon dont ils sont gérés au sein de cette collection.



Le nouveau monde : Iterator

Lorsque Sun a créé ses nouveaux types collections, nous avons commencé à utiliser une interface Iterator qui, comme Enumeration, sert à opérer des itérations sur les éléments d'une collection, mais qui permet de plus d'en supprimer..

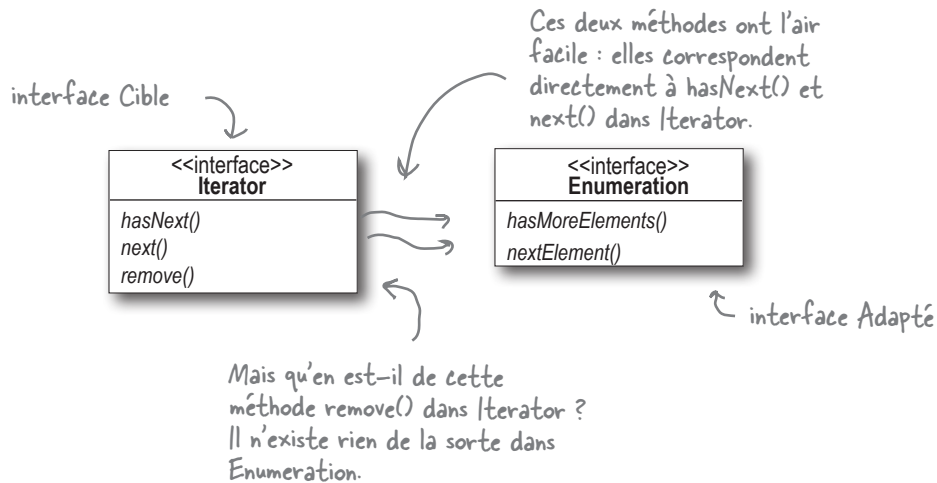


Et aujourd'hui...

Nous sommes souvent confrontés à du code hérité qui expose l'interface Enumeration, mais nous voudrions que notre nouveau code n'utilise qu'Iterator. On dirait que nous avons avoir besoin d'un adaptateur.

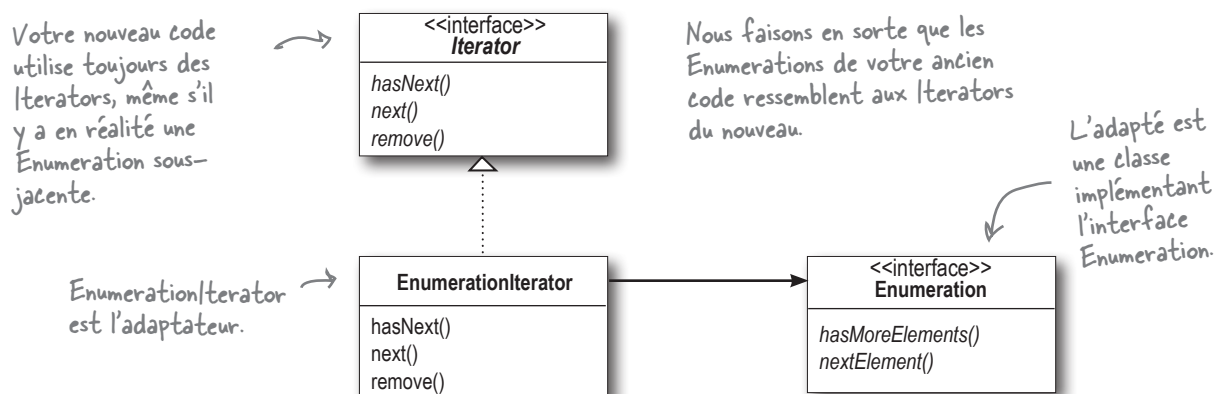
Adapter une Enumeration à un Iterator

Nous allons d'abord étudier les deux interfaces pour voir comment les méthodes se correspondent. Autrement dit, nous allons déterminer quoi appeler sur l'interface Adapté quand le client invoque une méthode sur la cible.



Concevoir l'Adaptateur

Voici à quoi nos classes doivent ressembler : il nous faut un adaptateur qui implémente l'interface `Cible` et qui soit composé avec un adapté. La correspondance entre les méthodes `hasNext()` et `next()` va être facile à réaliser entre la cible et l'adaptateur : nous les transposons directement. Mais que faites-vous de `remove()` ? Réfléchissez-y un moment (nous allons nous en occuper à la page suivante). Pour l'instant, voici le diagramme de classes :



Traiter la méthode remove()

Bien. Nous savons qu'Enumeration est purement et simplement incapable de prendre en charge remove(). C'est une interface « en lecture seule ». Il n'existe aucun moyen d'implémenter une méthode remove() pleinement fonctionnelle dans l'adaptateur. Au mieux, nous pouvons lancer une exception au moment de l'exécution. Heureusement, les concepteurs de l'interface Iterator ont prévu ce cas et ont défini remove() pour qu'elle supporte une UnsupportedOperationException.

C'est l'une des situations dans lesquelles l'adaptateur n'est pas parfait. Les clients devront gérer les exceptions potentielles, mais tant qu'ils font attention et que l'adaptateur est bien documenté, c'est une solution parfaitement raisonnable.

Écrire l'adaptateur EnumerationIterator

Voici un code simple mais efficace pour toutes ces classes héritées qui continuent à produire des Enumerations :

```
public class EnumerationIterator implements Iterator
{
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Comme nous adaptons Enumeration à Iterator, notre Adaptateur implémente l'interface Iterator... Il doit ressembler à un Iterator.

L'Enumeration que nous adaptons. Comme nous utilisons la composition, nous la plaçons dans une variable d'instance.

La méthode hasNext() d'Iterator est déléguée à la méthode hasMoreElements() d'Enumeration...

... et la méthode next() d'Iterator est déléguée à la méthode nextElement() d'Enumeration.

Malheureusement, nous ne pouvons pas prendre en charge la méthode remove() d'Iterator, et nous laissons tomber ! Nous nous contentons de lancer une exception.



Si Java a pris la direction de l'Iterator, il existe néanmoins beaucoup de **code client** hérité qui dépend de l'interface Enumeration, et un Adaptateur qui convertit un Iterator en Enumeration est également très utile.

Écrivez un Adaptateur qui convertit un Iterator en Enumeration. Vous pouvez tester votre code en adaptant une ArrayList. La classe ArrayList supporte l'interface Iterator mais pas Enumeration (enfin, pas encore en tous cas).

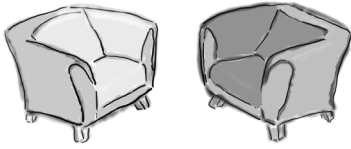


MUSCLEZ VOS NEURONES

Non contents de modifier l'interface, certains adaptateurs électriques ajoutent des fonctionnalités : protection contre les surtensions, voyants indicateurs et autres accessoires possibles et imaginables.

Si vous deviez implémenter ce genre de fonctionnalités, quel pattern appliqueriez-vous ?

Face à face :



L'entretien de ce soir : **Le pattern Décorateur et le pattern Adaptateur analysent leurs différences.**

Décorateur

Je suis très important. Ma vocation, c'est la responsabilité – quand un Décorateur est impliqué, vous savez que votre conception va comprendre de nouvelles responsabilités ou de nouveaux comportements.

C'est possible, mais n'allez pas croire qu'on se tourne les pouces. Quand il faut décorer une grosse interface, oh là là ! Cela peut demander des quantités de code.

Malin. Mais ne pensez pas que nous récoltons toute la gloire. Parfois, je ne suis qu'un décorateur enveloppé par je ne sais combien d'autres décorateurs. Quand une méthode m'est déléguée, je n'ai aucune idée du nombre de décorateurs qui l'ont déjà traitée, et je ne sais même pas si les efforts que j'ai déployés pour satisfaire la requête seront remarqués.

Adaptateur

Mais vous, vous voulez toute la gloire pendant que nous autres adaptateurs pataugeons dans les tranchées à faire le sale boulot : convertir des interfaces. Notre tâche n'est peut être pas prestigieuse, mais nos clients apprécient à coup sûr qu'on leur facilite la vie.

Essayez donc d'être un adaptateur, quand vous devez faire appel à plusieurs classes pour fournir l'interface que votre client attend. Voilà qui est dur. Mais nous avons un dicton : « un client découplé est un client heureux ».

Hé, si les adaptateurs font leur travail, nos clients ne remarquent même pas notre présence. Ce peut être un travail ingrat.

Décorateur

Nous aussi, seulement nous permettons d'ajouter de nouveaux comportements aux classes sans modifier le code existant. Je continue à prétendre que les adaptateurs ne sont que des sortes de décorateurs. Je veux dire que vous enveloppez un objet, tout comme nous.

Euh, non. Notre tâche dans la vie consiste à étendre les comportements ou les responsabilités des objets que nous enveloppons. Nous ne sommes que de simples intermédiaires.

Nous devrions peut-être convenir que nous sommes différents. Nous avons sans doute l'air quelque peu similaires sur le papier, mais nous sommes à des *lieues* l'un de l'autre en termes de *motivation*.

Adaptateur

Mais ce qui est génial avec nous autres adaptateurs, c'est que nous permettons aux clients d'utiliser de nouvelles bibliothèques et autres sous-ensembles sans avoir à changer une ligne de code, et de s'en remettre à nous pour effectuer la conversion à leur place. Bien sûr, c'est une niche, mais nous y excellons.

Non, non, non, pas du tout. Nous convertissons *toujours* l'interface de ce que nous enveloppons, vous *jamais*. Je dirais qu'un décorateur est comme un adaptateur, sauf que vous ne modifiez pas l'interface !

Hé, qui appelez-vous de *simples intermédiaires* ? Venez ici et nous verrons combien de temps vous durerez à convertir quelques interfaces !

Oh oui, là je vous rejoins.

Et maintenant pour changer...

Il y a un autre pattern dans ce chapitre.

Vous avez vu comment le pattern Adaptateur convertit l'interface d'une classe en celle que le client attend. Vous savez également que Java permet de mettre en œuvre ce mécanisme en enveloppant l'objet qui a une interface incompatible dans un objet qui implémente la bonne.

Nous allons maintenant étudier un pattern qui modifie une interface, mais pour une autre raison : pour simplifier ladite interface. On l'appelle à point nommé le pattern Façade parce qu'il masque toute la complexité d'une ou plusieurs classes derrière une façade bien propre et bien éclairée.

Qui fait quoi ?

Faites correspondre chaque pattern à sa motivation :

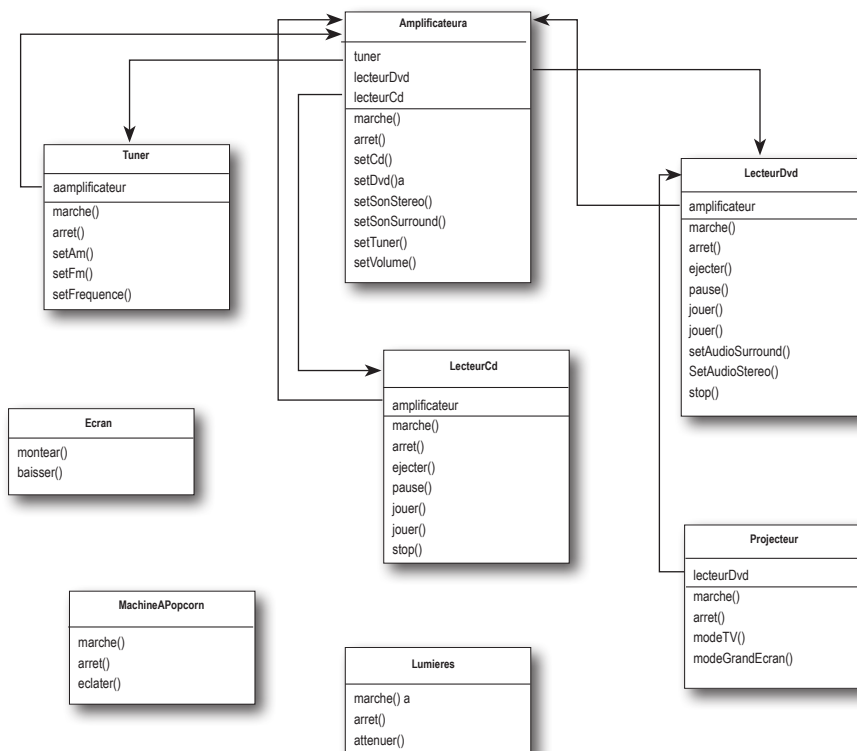
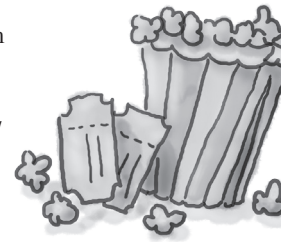
Pattern	Motivation
Décorateur	Convertit une interface en une autre
Adaptateur	Ne modifie pas l'interface mais ajoute une responsabilité
Façade	Simplifie une interface

Home Cinéma

Avant de plonger dans les détails du pattern Façade, jetons un coup d'œil à une obsession nationale qui ne cesse de prendre de l'ampleur : construire son propre *home cinéma*.

Vous avez fait des recherches et vous avez assemblé un système d'enfer, avec un lecteur de DVD, un projecteur vidéo, un écran automatisé, le son *surround* et même une machine à pop-corn.

Vérifions tous les composants que vous avez assemblés :



Cela fait beaucoup de classes, beaucoup d'interactions et un ensemble important d'interfaces à apprendre et à utiliser.

Vous avez passé des semaines à tirer des fils, à monter le projecteur, à réaliser toutes les connexions et à effectuer tous les réglages. Maintenant, il est temps de tout mettre en marche et d'apprécier un bon film...

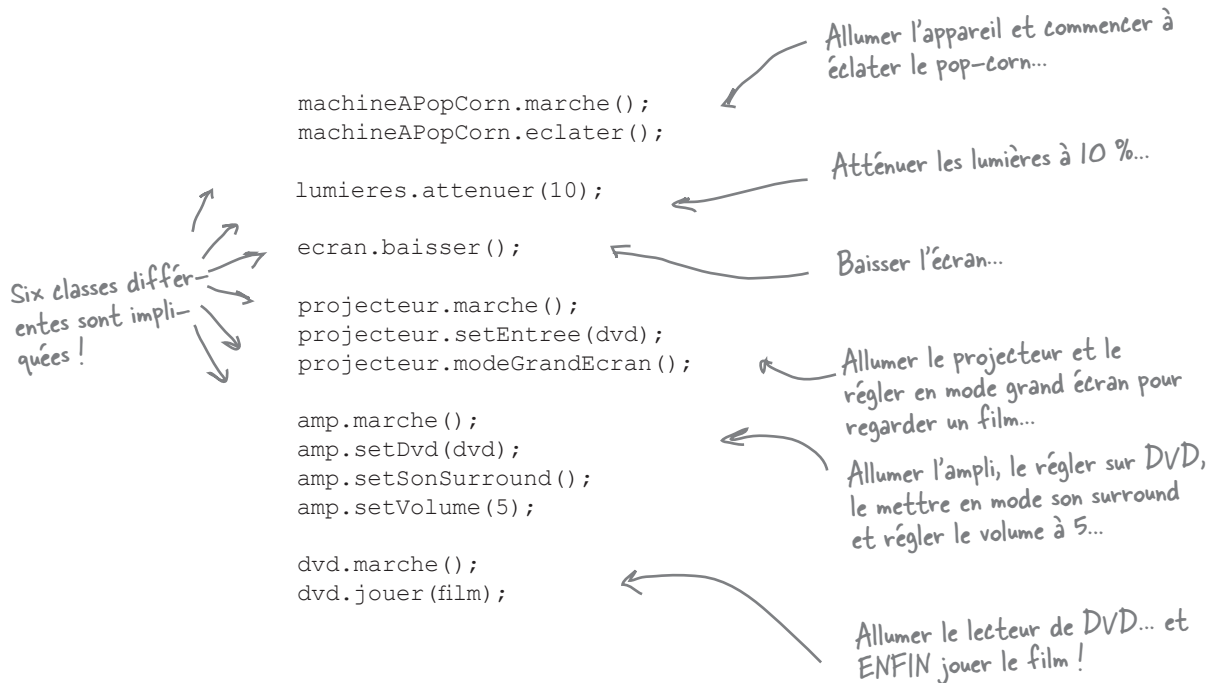
Regarder un film (à la dure)

Choisissez un DVD, relaxez-vous et préparez vous à contempler la magie du cinéma. Oh, juste une chose, pour regarder un film, vous devez effectuer un certain nombre de tâches :

- 1 Allumer la machine à pop-corn**
- 2 Commencer à faire éclater le pop-corn**
- 3 Atténuer les lumières**
- 4 Baisser l'écran**
- 5 Allumer le projecteur**
- 6 Régler le projecteur sur DVD**
- 7 Régler le projecteur en mode grand écran**
- 8 Allumer l'amplificateur de son**
- 9 Régler l'amplificateur sur DVD**
- 10 Régler le son de l'amplificateur à surround**
- 11 Régler le volume de l'amplificateur sur moyen (5)**
- 12 Allumer le lecteur de DVD**
- 13 Jouer le DVD**



Voyons ces mêmes tâches en termes de classes et d'appels de méthode nécessaires pour les réaliser :



Mais ce n'est pas tout...

- Quand le film est terminé, comment faites-vous pour tout éteindre ?
Ne faut-il pas tout refaire, mais cette fois à l'envers ?
- Ne serait-ce pas aussi complexe d'écouter un CD ou la radio ?
- Et si vous décidez d'acquérir un nouveau composant, vous devrez probablement apprendre une procédure légèrement différente.

Alors, que faire ? La complexité d'utilisation de votre home cinéma commence à apparaître !

Voyons comment le pattern Façade peut nous sortir de ce pétrin et nous permettre d'apprécier le film...

Lumières, Caméra, Façade !

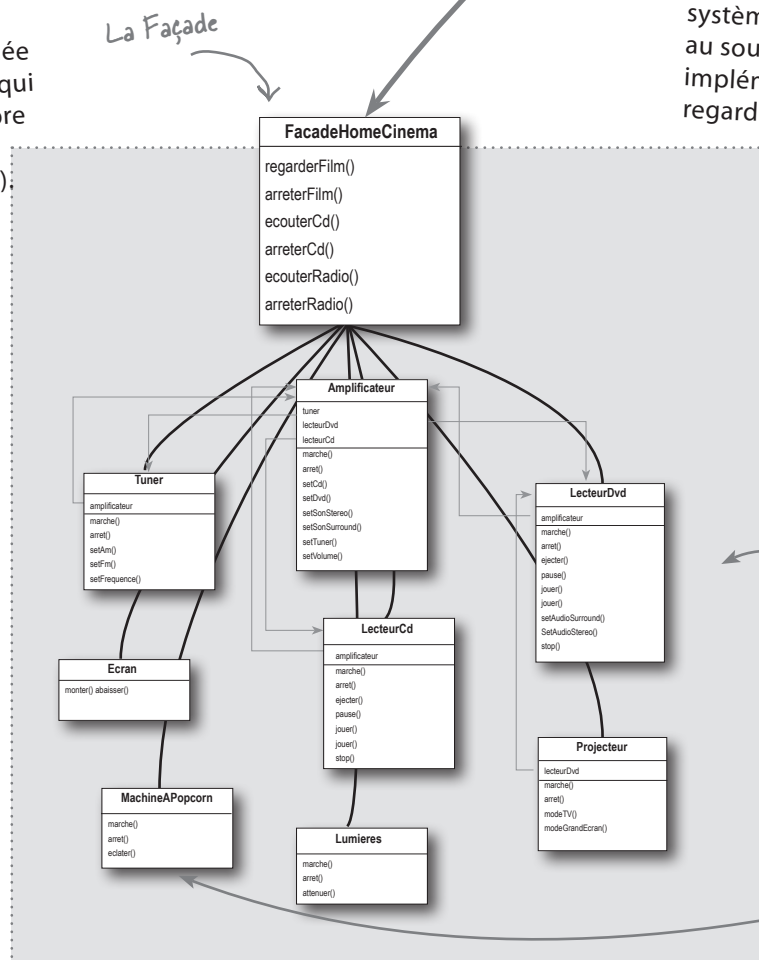
Une Façade est exactement ce qu'il vous faut : le pattern Façade vous permet de prendre un sous-système complexe et d'en simplifier l'utilisation en implémentant une classe Façade qui fournit une interface unique et plus raisonnable. Ne vous inquiétez pas : si vous avez besoin du sous-système complexe, il est toujours à votre disposition, mais si vous n'avez besoin que d'une interface simple, la Façade est là pour vous.

Jetons un coup d'œil au fonctionnement de la Façade :

- 1 Bien. Il est temps de créer une Façade pour le système de home cinéma. Pour ce faire, nous créons une nouvelle classe nommée `FacadeHomeCinema`, qui expose un petit nombre de méthodes simples comme `regarderFilm()`.

- 2 La classe Façade traite les composants du home cinéma comme un sous-système, et fait appel au sous-système pour implémenter sa méthode `regarderFilm()`.

Le sous-système que la Façade simplifie.



jouer()

marche()

regarderFilm()

Un client du sous-système
de la façade

- 3** Le code de votre client appelle maintenant les méthodes sur la Façade, pas sur le sous-système. Désormais, pour regarder un film, il suffit d'appeler une seule méthode, regarderFilm(), et celle-ci communique à notre place avec les lumières, le lecteur de DVD, le projecteur, l'amplificateur, l'écran et la machine à pop-corn.

Il faut que j'aie mon propre accès de bas niveau !



Ancien président du ciné-club du collège de Villedieu-les-Poêles.

- 4** La Façade permet toujours d'utiliser directement le sous-système qui demeure accessible. Si vous avez besoin des fonctionnalités avancées des classes du sous-système, elles sont disponibles.

il n'y a pas de questions stupides

Q: Si la Façade encapsule les classes du sous-système, comment un client qui a besoin d'une fonctionnalité de plus bas niveau peut-il y accéder ?

R: Les Façades n'« encapsulent » pas les classes du sous-système. Elles fournissent simplement une interface simplifiée à leurs fonctionnalités. Les classes du sous-système demeurent toujours accessibles et directement utilisables par les clients qui ont besoin d'interfaces plus spécifiques. C'est une propriété sympathique du pattern Façade : il fournit une interface simplifiée tout en continuant à exposer toutes les fonctionnalités du système à ceux qui pourraient en avoir besoin.

Q: La façade ajoute-t-elle une fonctionnalité quelconque ou se contente-t-elle de transmettre chaque requête au sous-système ?

R: Une façade est libre d'ajouter ses propres « plus » à l'utilisation du sous-système. Par exemple, si la façade de notre home cinéma n'implémente aucun nouveau comportement, elle est suffisamment intelligente pour savoir qu'une machine à pop-corn doit être allumée avant que le maïs ne puisse éclater (de même qu'elle connaît les détails des opérations pour démarrer et arrêter un film).

Q: Est-ce que chaque sous-système n'a qu'une seule façade ?

R: Pas nécessairement. Le pattern permet sans équivoque de créer un nombre quelconque de façades pour un sous-système donné.

Q: Quel est l'avantage d'une façade en dehors du fait qu'elle simplifie l'interface ?

R: Le pattern Façade vous permet également de découpler l'implémentation de votre client de tout sous-système. Disons par exemple que vous avez eu une grosse augmentation et que vous décidez d'acheter tout un tas de nouveaux composants qui ont des interfaces différentes. Eh bien, si vous avez codé votre client en fonction de la façade et non du sous-système, ce code client n'a pas besoin de changer : seule la façade est modifiée (et avec un peu de chance le fournisseur s'en occupe !).

Q: Donc la différence entre le pattern Adaptateur et le pattern Façade est que l'adaptateur enveloppe une classe et que la façade peut représenter plusieurs classes ?

R: Non ! Souvenez-vous : le pattern Adaptateur transforme l'interface d'une ou plusieurs classes en une interface à laquelle le client s'attend. Si dans la plupart des ouvrages les exemples montrent un adaptateur adaptant une seule classe, vous pouvez avoir besoin d'adapter plusieurs classes pour fournir l'interface en fonction de laquelle le client est codé. De même, une Façade peut fournir une interface simplifiée à une seule classe dotée d'une interface très complexe.

La différence entre les deux ne réside pas dans le nombre de classes qu'ils peuvent « envelopper » mais dans leur motivation. La motivation du pattern Adaptateur est de modifier une interface pour qu'elle corresponde à celle qu'un client attend. Celle du pattern Façade est de fournir une interface simplifiée à un sous-système.

Une façade ne se limite pas à simplifier une interface : elle découple un client d'un sous-système de composants.

Façades et adaptateurs peuvent envelopper plusieurs classes, mais la finalité d'une façade est de simplifier une interface, tandis que celle d'un adaptateur est de la convertir en quelque chose de différent.

Construire la façade de votre home cinéma

Voyons les étapes de la construction de la FacadeHomeCinema:

La première consiste à utiliser la composition afin que la façade ait accès à tous les composants du sous-système :

```
public class FacadeHomeCinema {  
    Amplificateur amp;  
    Tuner tuner;  
    LecteurDvd dvd;  
    LecteurCd cd;  
    Projecteur projecteur;  
    Lumieres lumieres;  
    Ecran ecran;  
    MachineAPopcorn machineAPopCorn;  
  
    public FacadeHomeCinema(Amplificateur amp,  
    Tuner tuner,  
        LecteurDvd dvd,  
        LecteurCd cd,  
        Projecteur projecteur,  
        Ecran ecran,  
        Lumieres lumieres,  
        MachineAPopcorn machineAPopCorn) {  
  
        this.amp = amp;  
        this.tuner = tuner;  
        this.dvd = dvd;  
        this.cd = cd;  
        this.projecteur = projecteur;  
        this.ecran = ecran;  
        this.lumieres = lumieres;  
        this.machineAPopCorn = machineAPopCorn;  
    }  
    // autres méthodes  
}
```

Voici la composition : ce sont tous les composants du sous-système que nous allons utiliser.

On transmet au constructeur de la façade une référence à chaque composant du sous-système. Puis la façade affecte chacun d'eux à la variable d'instance correspondante.

Nous allons les voir dans un instant...

Implémenter l'interface simplifiée

C'est maintenant le moment de rassembler les composants du sous-système dans une interface unifiée. Implémentons les méthodes `regarderFilm()` et `arreterFilm()` :

```
public void regarderFilm(String film) {
    System.out.println("Vous allez voir un bon film...");
    machineAPopCorn.marche();
    machineAPopCorn.eclater();
    lumieres.attenuer(10);
    ecran.baisser();
    projecteur.marche();
    projecteur.modeGrandEcran();
    amp.marche();
    amp.setDvd(dvd);
    amp.setSonSurround();
    amp.setVolume(5);
    dvd.marche();
    dvd.jouer(film);
}
```

regarderFilm() observe la même séquence que lorsque nous faisons tout à la main, mais elle l'enveloppe dans une méthode pratique qui fait tout le travail. Remarquez que nous déléguons la responsabilité de chaque tâche au composant correspondant dans le sous-système.

```
public void arreterFilm() {
    System.out.println("C'est la fin du film...");
    machineAPopCorn.arret();
    lumieres.marche();
    ecran.monter();
    projecteur.arret();
    amp.arret();
    dvd.stop();
    dvd.ejecter();
    dvd.arret();
}
```

Et arreterFilm() s'occupe de tout arrêter à votre place. Encore une fois, chaque tâche est déléguée au composant approprié dans le sous-système.



MUSCLEZ VOS NEURONES

Pensez aux façades que vous avez rencontrées dans l'API Java.
Où aimeriez-vous en avoir quelques unes de plus ?

Regardons un film (sans peine)

C'est l'heure du SPECTACLE !



```
public class TestHomeCinema {
    public static void main(String[] args) {
        // instantiation des composants

        FacadeHomeCinema homeCinema =
            new FacadeHomeCinema(amp, tuner, dvd, cd,
                projecteur, ecran, lumieres, machineAPopCorn);

        homeCinema.regarderFilm("Hôtel du Nord");
        homeCinema.arreterFilm();
    }
}
```

Ici nous créons les composants directement dans le test. Normalement, le client voit une façade et n'a pas besoin de la construire lui-même.

On commence par instancier la Façade avec tous les composants du sous-système.

On utilise l'interface simplifiée pour démarrer le film, puis pour l'arrêter.

Voici le résultat.

L'appel de la méthode regarderFilm() de la Façade fait tout le travail à notre place...

... ici, nous avons fini de regarder le film et arrêter-Film() arrête tous les appareils.

```
Fichier Fenêtre Édition Aide AtmosphèreAtmosphère?
%java TestHomeCinema

Vous allez voir un bon film...
Machine à pop-corn en marche
Machine à pop-corn fait éclater le pop-corn !
Lumières du Home Cinéma atténuées à 10%
Écran du Home Cinéma descendu
Projecteur en marche
Projecteur en mode grand écran (aspect 16x9)
Magnifique Amplificateur en marche
Magnifique Amplificateur positionné sur le lecteur DVD Super Lecteur DVD
Magnifique Amplificateur réglé sur son surround (5 enceintes, 1 caisson de basses)
Magnifique Amplificateur volume réglé sur 5
Super Lecteur DVD en marche
Super Lecteur DVD joue «Hôtel du Nord»
C'est la fin du film...
Machine à pop-corn arrêtée
Lumières du Home Cinéma allumées
Écran du Home Cinéma remonté
Projecteur arrêté
Magnifique Amplificateur éteint
Super Lecteur DVD arrêté sur «Hôtel du Nord»
Super Lecteur DVD éjection
Super Lecteur DVD arrêté
%
```

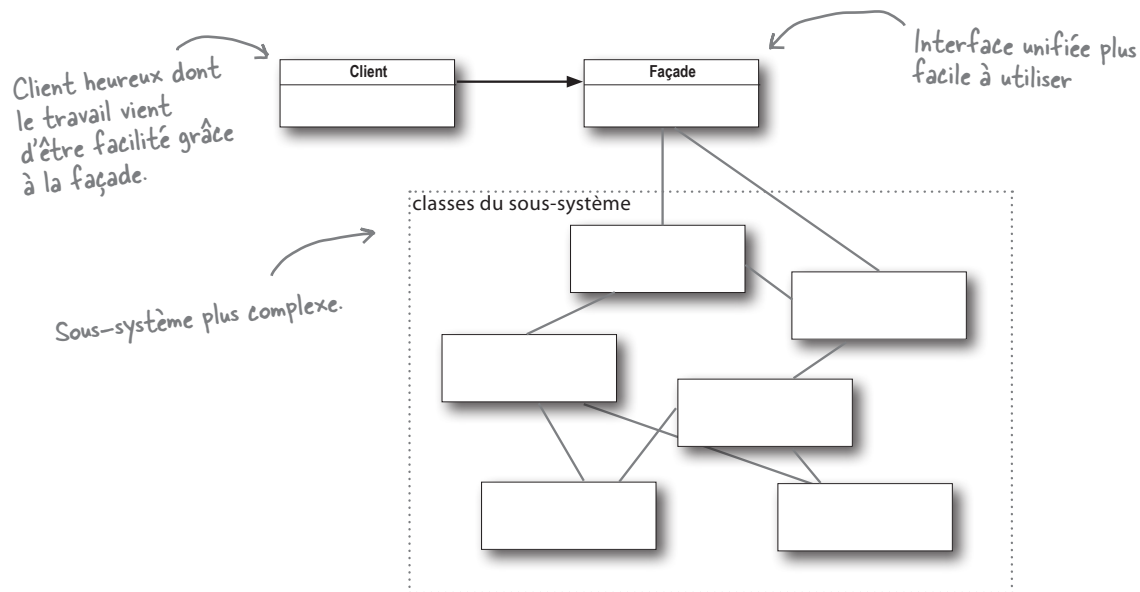

Le pattern *Façade* : définition

Pour utiliser le pattern *Façade*, nous créons une classe qui simplifie et unifie un ensemble de classes plus complexes qui appartiennent à un sous-système donné. Contrairement à beaucoup de patterns, *Façade* est relativement simple et ne contient aucune de ces abstractions compliquées qui vous font tourner la tête. Mais cela n'ôte rien à sa puissance : le pattern *Façade* vous permet d'éviter de coupler fortement les clients et les sous-systèmes et, comme vous allez le voir bientôt, vous aide également à observer un autre principe orienté objet.

Mais avant de présenter ce nouveau principe, regardons la définition officielle du pattern :

Le pattern *Façade* fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.

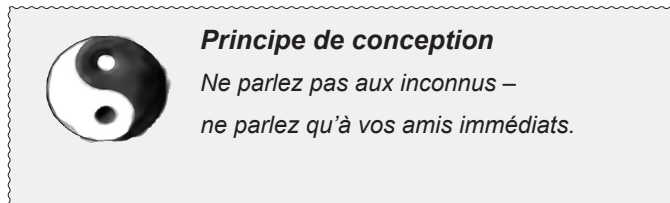
Il n'y a rien là que vous ne sachiez déjà, mais l'une des choses les plus importantes à mémoriser à propos d'un pattern est sa motivation. Cette définition clame haut et fort que l'objectif d'une façade est de rendre un sous-système plus facile à utiliser via une interface simplifiée. Vous pouvez le constater dans son diagramme de classes :



Et voilà ! Vous avez un autre pattern en poche ! Maintenant, il est temps de voir ce nouveau principe OO. Attention, celui-ci peut remettre en question quelques idées reçues !


Ne parlez pas aux inconnus

Ne parlez pas aux inconnus est un principe qui nous aide à restreindre les interactions entre objets à quelques « amis » proches. Il s'énonce habituellement ainsi :



Mais quel est le sens de ce principe dans la réalité ? Lors de la conception d'un système, il signifie que vous devez être attentif pour chaque objet au nombre de classes avec lesquelles il interagit et à la façon dont il entre en interaction avec elles.

Ce principe nous empêche de créer des systèmes constitués d'un grand nombre de classes fortement couplées dans lesquels les modifications d'une composante se propagent en cascade aux autres parties. Lorsqu'il existe trop de dépendances entre de nombreuses classes, vous construisez un système fragile dont la maintenance sera coûteuse et que les autres auront du mal à comprendre en raison de sa complexité.

**MUSCLEZ
VOS NEURONES**

À combien de classes ce code est-il couplé ?

```
public float getTemp() {  
    return station.getThermometre().getTemperature();  
}
```


Comment ne PAS se faire des amis et influencer les objets

D'accord, mais comment procéder concrètement ?

Voici quelques lignes directrices : considérez un objet quelconque ; maintenant, à partir de n'importe quelle méthode de cet objet, le principe nous enjoint de n'appeler que des méthodes qui appartiennent :

- À l'objet lui-même
- Aux objets transmis en arguments à la méthode
- Aux objets que la méthode crée ou instancie
- Aux composants de l'objet

Remarquez que ces lignes directrices nous disent de ne pas appeler de méthodes appartenant à des objets retournés par d'autres appels de méthodes !!

Représentez-vous un « composant » comme un objet qui est référencé par une variable d'instance. Autrement dit, il s'agit d'une relation A-UN.

Cela semble un peu draconien, n'est-ce pas ? Quel mal y a-t-il à appeler la méthode d'un objet qu'un autre appel nous a retourné ? Eh bien, si nous le faisons, ce serait une requête d'une sous-partie d'un autre objet (et nous augmenterions le nombre d'objets que nous connaissons directement). Dans de tels cas, le principe nous force à demander à l'objet d'émettre la requête à notre place. Ainsi, nous n'avons pas à connaître les objets qui le composent (et notre cercle d'amis demeure restreint). Par exemple :

Sans le Principe

```
public float getTemp() {
    Thermometre thermometre = station.getThermometre();
    return thermometre.getTemperature();
}
```

Ici, nous obtenons l'objet thermomètre de la station, puis nous appelons la méthode getTemperature() nous-mêmes.

Avec le Principe

```
public float getTemp() {
    return station.getTemperature();
}
```

Quand nous appliquons le principe, nous ajoutons à la classe Station une méthode qui demande le thermomètre à notre place. Cette technique réduit le nombre de classes dont nous dépendons.

Fixer des limites aux appels de méthodes...

Voici une classe Voiture qui illustre toutes les façons dont vous pouvez appeler des méthodes tout en observant le principe Ne parlez pas aux inconnus :

```
public class Voiture {
    Moteur moteur;
    // autres variables d'instance

    public Voiture() {
        // initialiser moteur, etc.
    }

    public void demarrer(Cle cle) {
        Portes portes = new Portes();

        boolean autorise = cle.tourner();

        if (autorise) {
            moteur.demarrer();
            majTableauDeBord();
            portes.fermer();
        }
    }

    public void majTableauDeBord() {
        // mettre à jour l'affichage
    }
}
```

Voici un composant de cette classe. Nous pouvons appeler ses méthodes.

Ici, nous créons un nouvel objet : l'appel de ses méthodes est légal.

Vous pouvez appeler une méthode d'un objet transmis en argument.

Vous pouvez appeler une méthode sur un composant de l'objet.

Vous pouvez appeler une méthode locale à l'objet.

Vous pouvez appeler une méthode d'un objet que vous créez ou que vous instanciez.

Il n'y a pas de questions stupides

Q: Il y a un autre principe nommé **Loi de Déméter**. Comment sont-ils liés ?

R: Il s'agit d'une seule et même chose et vous rencontrerez les deux termes indifféremment. Nous préférons Ne parlez pas aux inconnus pour deux raisons : (1) il se comprend mieux intuitivement et (2) l'emploi du mot « loi » laisse entendre que ce principe doit toujours être appliqué.

En fait, aucun principe n'est une loi et tous doivent être utilisés quand et où ils sont utiles. Toute conception implique des compromis (abstractions vs. rapidité, espace vs. temps, etc.). Les principes peuvent nous guider, mais nous devons prendre en compte tous les facteurs avant de les appliquer.

Q: Y a-t-il un inconvénient quelconque à appliquer **Ne parlez pas aux inconnus** ?

R: Oui. Si ce principe diminue les dépendances entre les objets et si des études ont démontré que la maintenance s'en trouve réduite, il arrive également que son application conduise à écrire plus de classes « enveloppes » pour gérer les appels de méthodes à d'autres composants. Cela peut entraîner une augmentation de la complexité et du temps de développement, ainsi qu'une dégradation des performances lors de l'exécution.

À vos crayons

L'une ou l'autre de ces classes enfreint-elle le principe Ne parlez pas aux inconnus ? Pourquoi ?

```
public Maison {
    StationMeteo station;

    // autres méthodes et constructeur

    public float getTemp() {
        return station.getThermometre().getTemperature();
    }
}
```

```
public Maison {
    StationMeteo station;

    // autres méthodes et constructeur

    public float getTemp() {
        Thermometre thermometre = station.getThermometre();
        return getAuxiliaireTemp(thermometre);
    }

    public float getAuxiliaireTemp(Thermometre thermometre) {
        return thermometre.getTemperature();
    }
}
```



**Attention ! Chutes d'hypothèses.
Port du casque obligatoire.**

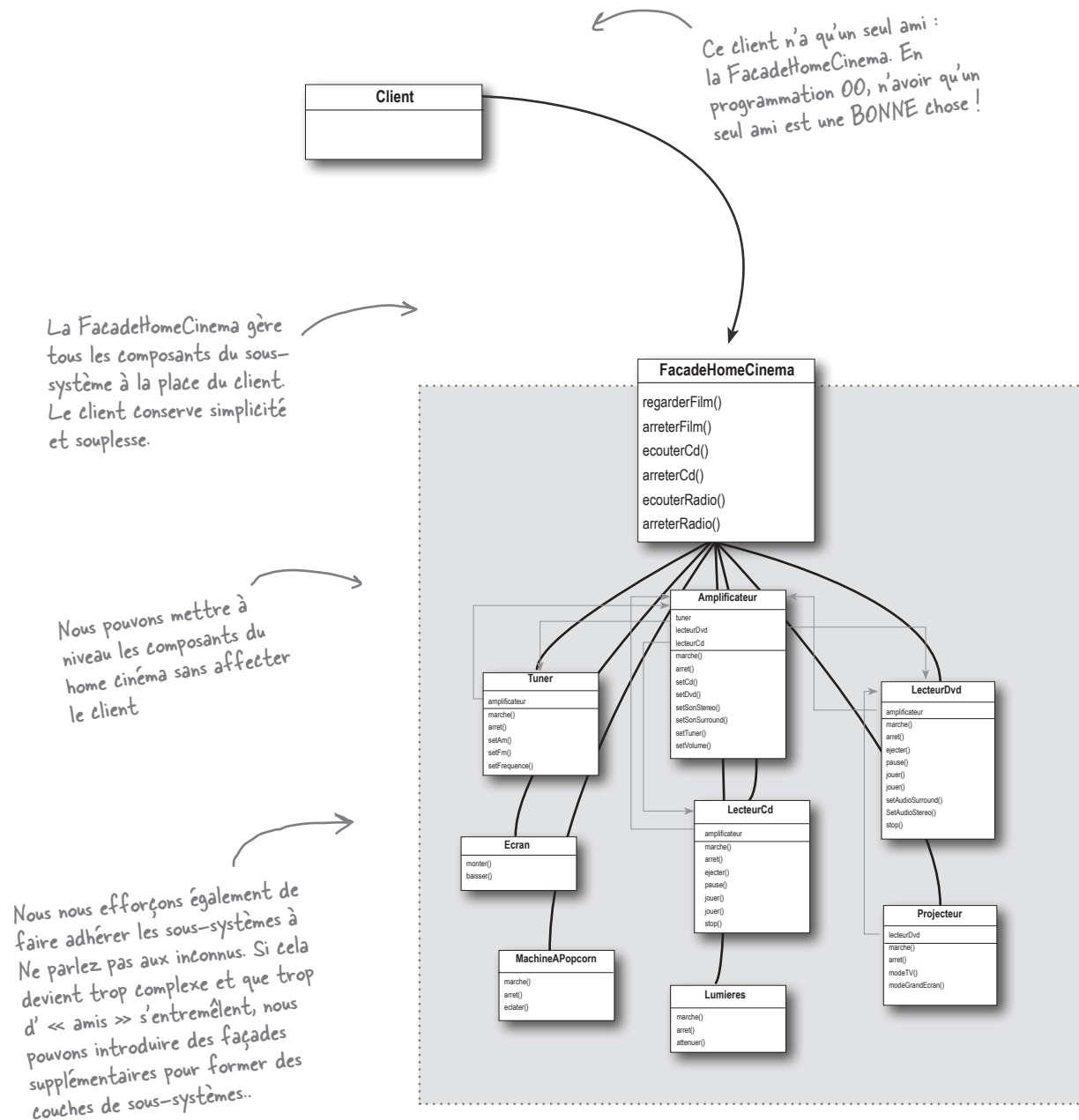


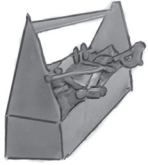
MUSCLEZ VOS NEURONES

Connaissez-vous un emploi courant de Java qui viole le principe Ne parlez pas aux inconnus ?
Est-ce important ?

Réponse : Que pensez-vous de `System.out.println()` ?

Façade et Ne parlez pas aux inconnus





Votre boîte à outils de concepteur

Votre boîte à outils commence à s'alourdir ! Dans ce chapitre, nous lui avons ajouté deux patterns qui permettent de modifier des interfaces et de réduire le couplage entre les clients et les systèmes qu'ils utilisent.

Principes OO

Encapsulez ce qui varie.
Préférez la composition à l'héritage.
Programmez des interfaces, non des implémentations.
Efforcez-vous de coupler faiblement les objets qui interagissent.
Les classes doivent être ouvertes à l'extension mais fermées à la modification.
Dépendez des abstractions. Ne dépendez pas des classes concrètes.
Ne parlez qu'à vos amis.

Bases de l'OO

abstraction
encapsulation
polymorphisme
héritage

Nous disposons d'une nouvelle technique pour maintenir le couplage faible. (Souvenez-vous : ne parlez qu'à vos amis)...

et de DEUX nouveaux patterns. Chacun d'eux modifie une interface, l'adaptateur pour convertir et la façade pour unifier et simplifier.

Patterns OO

Adaptateur – convertit l'interface d'une classe en une autre conforme à celle du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.

Façade – fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.

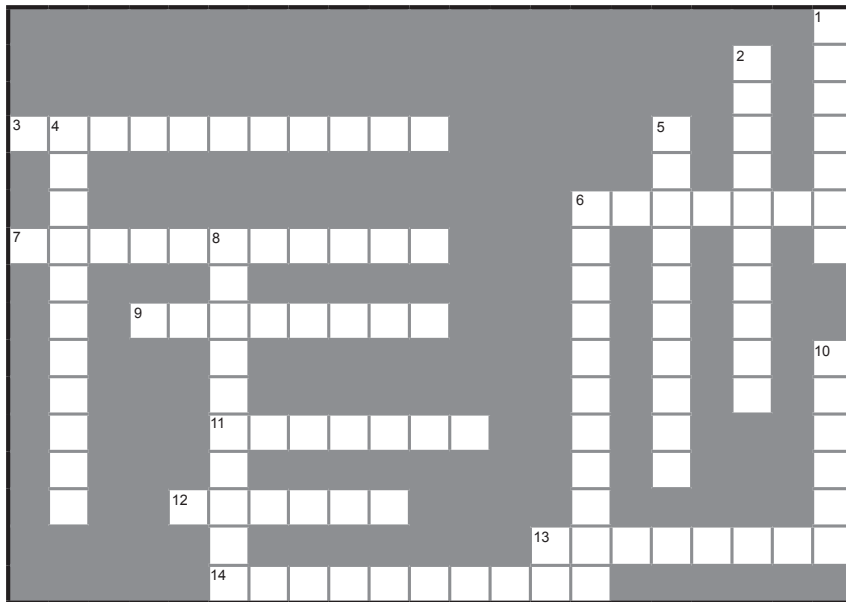


POINTS D'IMPACT

- Quand vous devez utiliser une classe existante et que son interface n'est pas celle dont vous avez besoin, employez un adaptateur.
- Quand vous devez simplifier et unifier une grosse interface ou un ensemble d'interfaces complexe, employez une façade.
- Un adaptateur transforme une interface en celle que le client attend.
- Une façade découple un client d'un sous-système complexe.
- L'implémentation d'un adaptateur peut demander plus ou moins de travail selon la taille et la complexité de l'interface cible.
- L'implémentation d'une façade nécessite de composer la façade avec son sous-système et d'utiliser la délégation pour effectuer le travail de la façade.
- Le pattern Adaptateur possède deux formes : les adaptateurs d'objet et les adaptateurs de classe. Les adaptateurs de classe font appel à l'héritage multiple.
- Vous pouvez implémenter plus d'une façade pour un même sous-système.
- Un adaptateur enveloppe un objet pour modifier son interface, un décorateur enveloppe un objet pour ajouter de nouveaux comportements et de nouvelles responsabilités, et une façade « enveloppe » un ensemble d'objets afin de simplifier.



Oui, c'est un autre mots-croisés. Toutes les solutions sont dans ce chapitre.



Horizontalement

3. Le monde réel en est plein.
6. C'est une loi.
7. Le fil que nous avons regardé.
9. Il ne faut pas leur parler.
11. Viole le principe précédent (System.out._____).
12. Simplifie une interface.
13. L'adaptateur de classe nécessite l'_____ multiple.
14. Wrappers.

Verticalement

1. Le home cinéma a même une machine à _____.
2. Attention, chute d'_____.
4. Ajoute de nouveaux comportements.
5. L'application sur laquelle nous avons travaillé.
6. Entre classes, point trop n'en faut.
8. L'un des avantages de Façade.
10. Il existe des adaptateurs d'objet et des adaptateurs de _____.
13. Interjection.



solutions des exercices

À vos crayons

Imaginez que nous ayons également besoin d'un Adaptateur qui convertit un Canard en Dindon. Appelons-le AdaptateurCanard. Écrivez cette classe :

```
public class AdaptateurCanard implements Dindon {
    Canard canard;
    Random rand;
```

Comme nous adaptons maintenant des Dindons à des Canards, nous implémentons l'interface Dindon.

```
    public AdaptateurCanard(Canard canard) {
        this.canard = canard;
        rand = new Random();
    }
```

Nous mémorisons une référence au Canard que nous adaptons.

Nous créons également un objet aléatoire ; regardez la méthode voler() pour voir comment elle est utilisée.

```
    public void glouglouter() {
        canard.cancaner();
    }
```

Un glouglou qui se transforme en cancan.

```
    public void voler() {
        if (rand.nextInt(5) == 0) {
            canard.voler();
        }
    }
```

Puisque les canards volent beaucoup plus loin que les dindons, nous avons décidé de ne faire voler le canard qu'un fois sur cinq en moyenne.

À vos crayons

L'une ou l'autre de ces classes enfreint-elle Ne parlez pas aux inconnus ? Pourquoi ?

```
public Maison {
    StationMeteo station;
    // autres méthodes et constructeur

    public float getTemp() {
        return station.getThermometre().getTemperature();
    }
}
```

```
public Maison {
    StationMeteo station;
    // autres méthodes et constructeur

    public float getTemp() {
        Thermometre thermometre = station.getThermometre();
        return getAuxiliaireTemp(thermometre);
    }
```

```
    public float getAuxiliaireTemp(Thermometre thermometre) {
        return thermometre.getTemperature();
    }
}
```

Enfreint le Ne parlez pas aux inconnus !
Vous appelez la méthode d'un objet retourné par un autre appel.

N'enfreint pas le Ne parlez pas aux inconnus !
On dirait que nous nous sommes débrouillés pour contourner ce principe. Quelque chose a-t-il réellement changé depuis que nous avons transféré l'appel à une autre méthode ?



Solutions des exercices

Vous avez vu comment écrire un adaptateur qui convertit une Enumeration en Iterator ; écrivez maintenant un adaptateur qui convertit un Iterator en Enumeration.

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

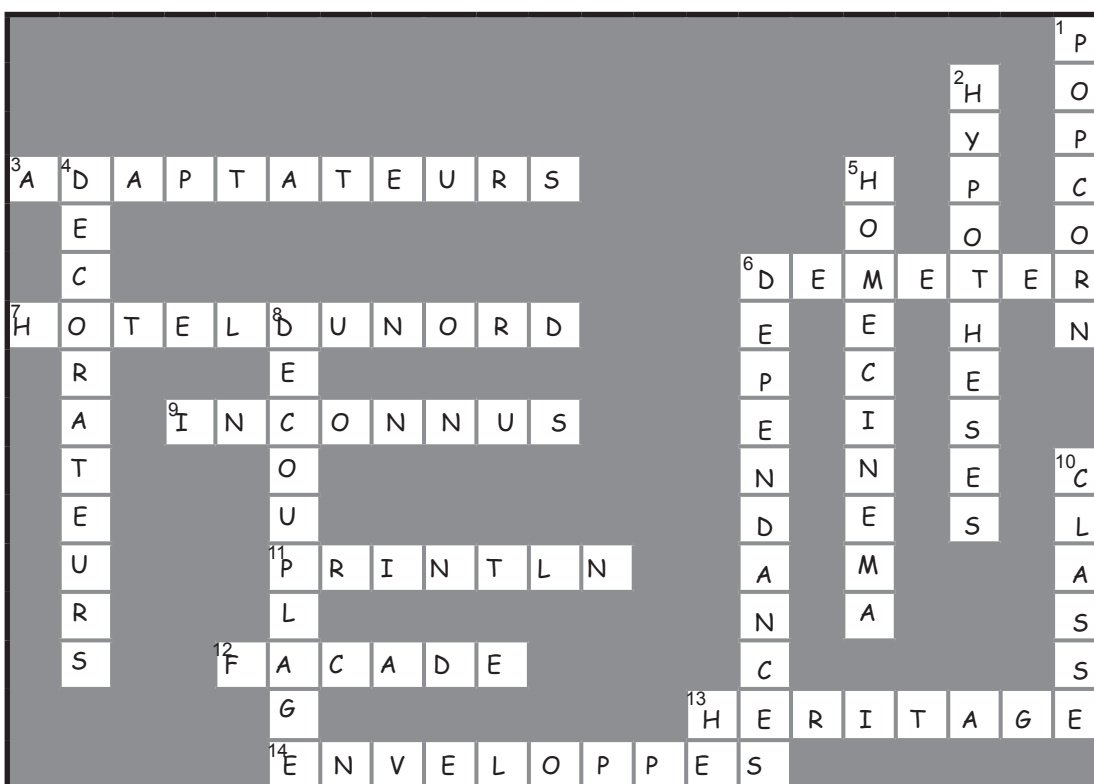
Qui fait quoi ?

Faites correspondre chaque pattern à sa motivation :

Pattern	Motivation
Décorateur	Convertit une interface en une autre
Adaptateur	Ne modifie pas l'interface mais ajoute une responsabilité
Façade	Simplifie une interface



Solutions des mots-croisés



8 le pattern Patron de méthode

Encapsuler les algorithmes

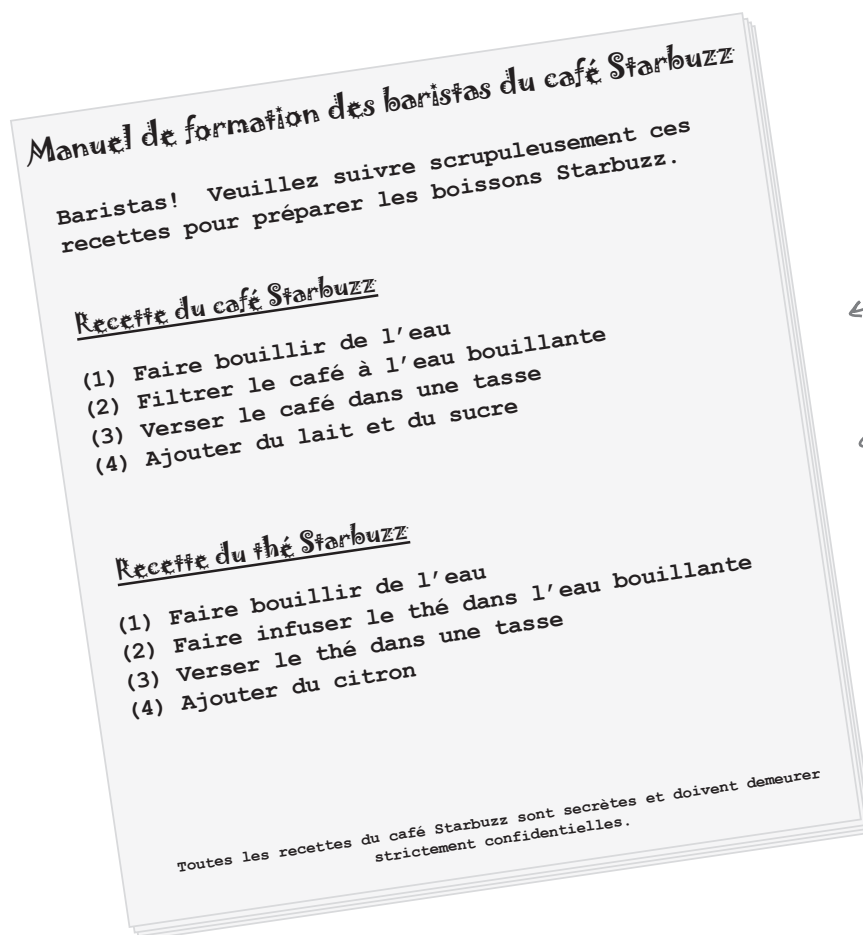


Nous n'arrêtons pas d'encapsuler. Nous avons encapsulé la création d'objets et l'invocation de méthodes. Nous avons encapsulé des interfaces complexes, des canards, des pizzas... Par quoi pourrions-nous continuer ? Nous allons entreprendre d'encapsuler des fragments d'algorithmes afin que les sous-classes puissent s'y adapter au moment de leur choix. Nous allons même découvrir un principe de conception inspiré par Hollywood.

Il est temps d'absorber un peu plus de caféine

Certaines personnes ne peuvent pas vivre sans leur café, d'autres ne peuvent pas vivre sans thé. L'ingrédient commun ? La caféine, bien sûr !

Mais ce n'est pas tout : on prépare le thé et le café de manière très similaire. C'est ce que nous allons vérifier.



La recette du café ressemble beaucoup à celle du thé, n'est-ce pas ?

Écrivons rapidement des classes pour le café et le thé (en Java)



Jouons au « barista programmeur » et écrivons du code pour faire du café et du thé.

Voici pour le café :

Voici notre classe Cafe qui fait du café.

```
public class Cafe {
```

```
    void suivreRecette() {
        faireBouillirEau();
        filtrerCafe();
        verserDansTasse();
        ajouterLaitEtSucre();
    }
```

Voici notre recette de café. Elle sort tout droit du manuel de formation. Chacune des étapes est implémentée au moyen d'une méthode distincte.

```
    public void faireBouillirEau() {
        System.out.println("Portage de l'eau à ébullition");
    }
```

```
    public void filtrerCafe() {
        System.out.println("Passage du café");
    }
```

```
    public void verserDansTasse() {
        System.out.println("Remplissage de la tasse");
    }
```

```
    public void ajouterLaitEtSucre() {
        System.out.println("Ajout du lait et du sucre");
    }
}
```

Chacune de ces méthodes implémente une étape de l'algorithme. Il y a une méthode pour faire bouillir l'eau, une pour passer le café, une pour le verser dans la tasse et une pour ajouter du lait et du sucre.

Et maintenant, le thé...



```
public class The {  
  
    void suivreRecette() {  
        faireBouillirEau();  
        tremperSachet();  
        verserDansTasse();  
        ajouterCitron();  
    }  
  
    public void faireBouillirEau() {  
        System.out.println("Portage de l'eau à ébullition");  
    }  
  
    public void tremperSachet() {  
        System.out.println("Infusion du thé");  
    }  
  
    public void verserDansTasse() {  
        System.out.println("Remplissage de la tasse");  
    }  
  
    public void ajouterCitron() {  
        System.out.println("Ajout du citron");  
    }  
}
```

Ceci ressemble beaucoup à ce que nous venons d'écrire dans la classe Cafe. La deuxième et la quatrième méthode sont différentes, mais il s'agit essentiellement de la même recette.

Ces deux méthodes sont spécifiques à la classe The.

Remarquez que ces deux méthodes sont exactement les mêmes que dans la classe Cafe ! Décidément, il y a de la duplication de code dans l'air.

Quand nous avons du code dupliqué, c'est le signe qu'il faut réviser notre conception. Ici, on dirait qu'il faut extraire les éléments communs pour les placer dans une classe de base puisque les recettes du café et du thé sont tellement similaires.



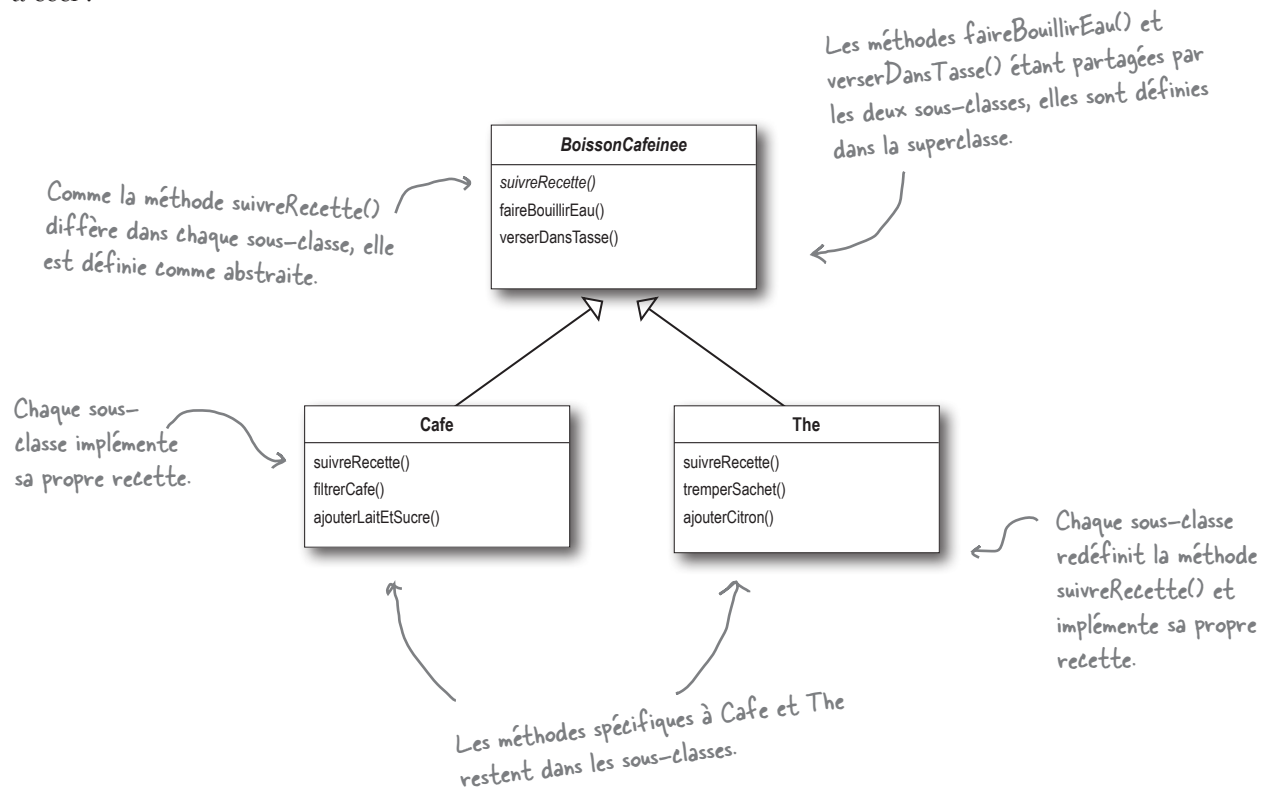


Problème de conception

Vous avez constaté que les classes Cafe et The contenaient pas mal de code dupliqué. Regardez de nouveau leur code et tracez un diagramme de classes décrivant comment vous pourriez revoir la conception de ces classes afin d'éliminer les redondances.

Monsieur, puis-je transformer votre café (thé) en abstraction ?

On dirait que nous sommes en présence d'un problème de conception plutôt facile avec ces classes Café et Thé. Votre premier essai ressemble peut-être à ceci :



MUSCLEZ VOS NEURONES

Avons-nous fait du bon travail avec cette nouvelle conception ? Mmmmm, jetons encore un coup d'œil. Ne sommes-nous pas en train d'oublier un autre point commun ? En quoi les classes Café et Thé sont-elles également similaires ?

Poursuivons notre raisonnement...

Quel est donc l'autre élément commun à Cafe et à The ?
Commençons par les recettes.

Recette du café Starbuzz

- (1) Faire bouillir de l'eau
- (2) Filtrer le café à l'eau bouillante
- (3) Verser le café dans une tasse
- (4) Ajouter du lait et du sucre

Recette du thé Starbuzz

- (1) Faire bouillir de l'eau
- (2) Faire infuser le thé dans l'eau bouillante
- (3) Verser le thé dans une tasse
- (4) Ajouter du citron

Remarquez que les deux recettes appliquent le même algorithme :

- ❶ Faire bouillir de l'eau.
- ❷ Utiliser l'eau chaude pour extraire le café ou le thé.
- ❸ Verser la boisson résultante dans une tasse.
- ❹ Ajouter les suppléments appropriés à la boisson.

Ces deux méthodes ne sont pas abstraites, mais elles sont similaires. Simplement, elles s'appliquent à des boissons différentes.

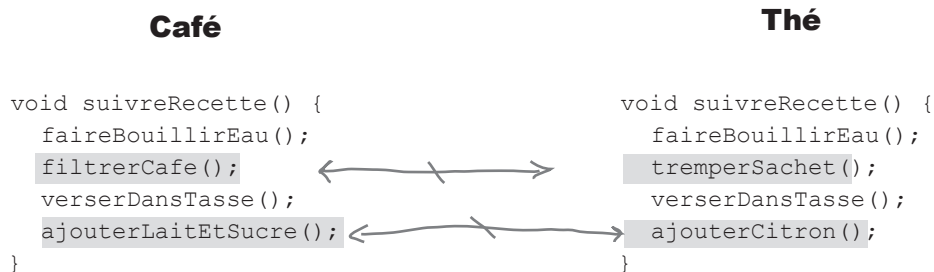
Ces deux-là sont déjà abstraites dans la classe de base.

Alors ? Pouvons-nous trouver un moyen d'abstraire également suivreRecette() ?
Oui. Réfléchissons...

Abstraire suivreRecette()

Voyons les étapes pour extraire suivreRecette() de chaque sous-classe (autrement dit, les classes Café et Thé)...

- 1 Premier problème : Café utilise les méthodes filtrerCafe() et ajouterLaitEtSucre() tandis que Thé utilise les méthodes tremperSachet() et ajouterCitron().



Réfléchissons : il n'y a pas grande différence entre faire passer du café et faire infuser du thé. C'est même pratiquement la même chose. Créons donc un nouveau nom de méthode, par exemple préparer(), et nous utiliserons le même nom, que nous fassions du café ou du thé.

De même, l'ajout de lait et de sucre n'est pas très différent de l'ajout de citron : les deux ajoutent des suppléments à la boisson. Nous pouvons donc créer un nouveau nom de méthode, ajouterSupplements(), pour résoudre le problème. Notre nouvelle méthode suivreRecette() va donc ressembler à ceci :

```
void suivreRecette() {
    faireBouillirEau();
    préparer();
    verserDansTasse();
    ajouterSupplements();
}
```

- 2 Nous avons maintenant une méthode suivreRecette(), mais il nous faut l'adapter au code. Pour ce faire, nous allons commencer par la superclasse, BoissonCaféinée :



BoissonCafeinee est abstraite, tout comme dans le modèle de classes.

```

public abstract class BoissonCafeinee {
    final void suivreRecette() {
        faireBouillirEau();
        preparer();
        verserDansTasse();
        ajouterSupplements();
    }

    abstract void preparer();

    abstract void ajouterSupplements();

    void faireBouillirEau() {
        System.out.println("Portage de l'eau à ébullition");
    }

    void verserDansTasse() {
        System.out.println("Remplissage de la tasse");
    }
}

```

Maintenant, la même méthode `suivreRecette()` sera utilisée pour préparer du café ou du thé. `suivreRecette()` est déclarée finale parce que nous ne voulons pas que les sous-classes puissent redéfinir cette méthode et modifier la recette ! Nous avons généralisé les étapes 2 et 4 pour `preparer()` la boisson et `ajouterSupplements()`.

Comme `Cafe` et `The` traitent ces méthodes différemment, il va falloir les déclarer abstraites et laisser les sous-classes faire le travail.

Souvenez-vous : nous avons transféré ces méthodes dans la classe `BoissonCafeinee` (voir le diagramme de classes).

- 3 Enfin, nous devons nous occuper des classes `Cafe` et `The`. Comme elles s'en remettent maintenant à `BoissonCafeinee` pour gérer la recette, elles n'ont plus qu'à traiter la préparation et les suppléments :

```

public class The extends BoissonCafeinee {
    public void preparer() {
        System.out.println("Infusion du thé");
    }
    public void ajouterSupplements() {
        System.out.println("Ajout du citron");
    }
}

public class Coffee extends BoissonCafeinee {
    public void preparer() {
        System.out.println("Passage du café");
    }
    public void ajouterSupplements() {
        System.out.println("Ajout du lait et du sucre");
    }
}

```

Comme dans notre conception, `The` et `Cafe` étendent maintenant `BoissonCafeinee`.

`The` doit définir les méthodes `preparer()` et `ajouterSupplements()` – les deux méthodes abstraites de la boisson

`Cafe` également, sauf que `Cafe` utilise du café, du lait et du sucre et non des sachets de thé et du citron.

À vos crayons

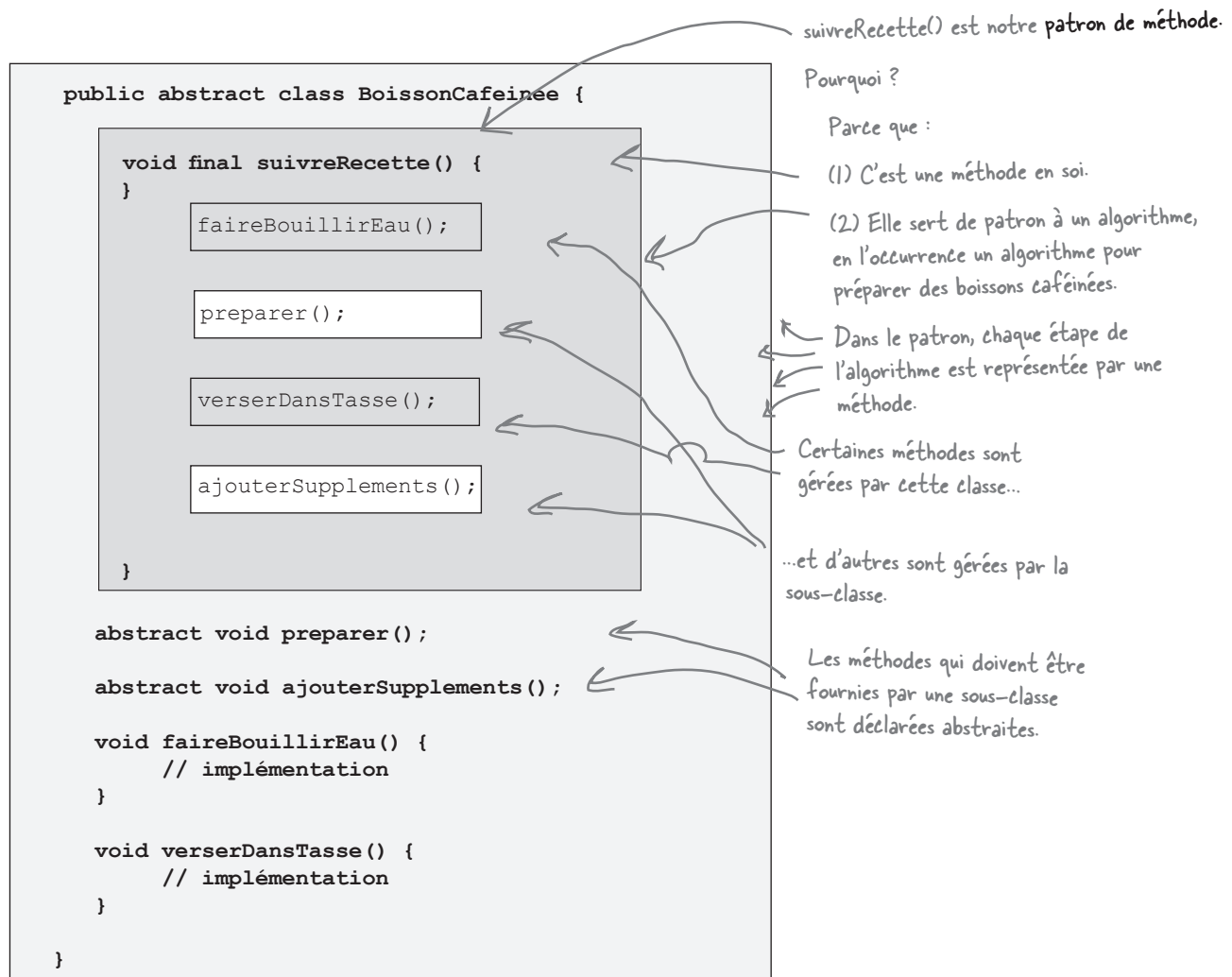
Tracez le nouveau diagramme de classes, maintenant que nous avons transféré l'implémentation de `sivreRecette()` dans la classe `BoissonCafeinee`.

Qu'avons-nous fait ?



Faites connaissance avec Patron de méthode

En substance, nous venons d'implémenter le pattern Patron de méthode. De quoi s'agit-il ?
Observons la structure de la classe `BoissonCafeinée` : elle contient le « patron de méthode ».



Le Patron de méthode définit les étapes d'un algorithme et permet aux sous-classes de fournir l'implémentation d'une ou plusieurs de ses étapes.

Préparons un peu de thé...

Voyons les étapes de la préparation d'un thé et observons le fonctionnement du patron de méthode. Vous allez voir que le patron de méthode contrôle l'algorithme. À certains points de l'algorithme, il laisse la sous-classe fournir l'implémentation des étapes...

Dans
les coulisses



- 1 Bien. D'abord, il nous faut un objet The...

```
The monThe = new The();
```

```
faireBouillirEau();
preparer();
verserDansTasse();
ajouterSupplements();
```

- 2 Puis nous appelons le patron de méthode :

```
monThe.suivreRecette();
```

qui applique l'algorithme pour préparer des boissons caféinées...

La méthode `suivreRecette()` contrôle l'algorithme, personne ne peut le modifier et elle compte sur les sous-classes pour fournir tout ou partie de l'implémentation.

- 3 Premièrement, on fait bouillir de l'eau :

```
faireBouillirEau();
```

ce qui se passe dans `BoissonCafeinee`.

- 4 Puis nous devons faire infuser le thé. Seule la sous-classe sait comment faire :

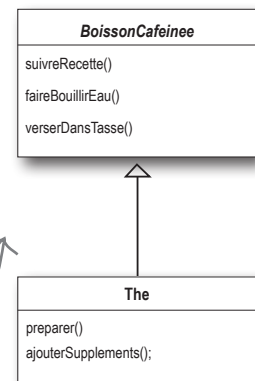
```
preparer();
```

- 5 Maintenant, nous versons le thé dans la tasse. Comme on verse toutes les boissons de la même façon, cela se passe dans la classe `BoissonCafeinee` :

```
verserDansTasse();
```

- 6 Enfin, nous ajoutons les suppléments. Comme ils sont spécifiques à chaque boisson, c'est la sous-classe qui implémente l'opération :

```
ajouterSupplements();
```



Que nous a apporté le patron de méthode ?



Implémentation de The et Cafe manquant de puissance

Cafe et The mènent la danse : ils contrôlent l'algorithme.

Le code est dupliqué entre Cafe et The.

Changer l'algorithme nécessite d'ouvrir les sous-classes et d'apporter de multiples modifications.

Les classes sont organisées selon une structure qui demande beaucoup de travail lorsqu'il faut ajouter une nouvelle boisson.

La connaissance de l'algorithme et la façon de l'implémenter est répartie entre plusieurs classes.



Nouvelle BoissonCafeinee branchée, boostée par le Patron de méthode

La classe BoissonCafeinee mène la danse : elle possède l'algorithme et elle le protège.

La classe BoissonCafeinee augmente la possibilité de réutilisation entre les sous-classes.

L'algorithme réside à un seul endroit, et c'est uniquement là que les modifications du code ont lieu.

La version Patron de méthode fournit un cadre dans lequel on peut insérer d'autres boissons caféinées. Celles-ci n'auront qu'une paire de méthodes à implémenter.

La classe BoissonCafeinee centralise la connaissance de l'algorithme et s'en remet aux sous-classes pour fournir les implémentations complètes.

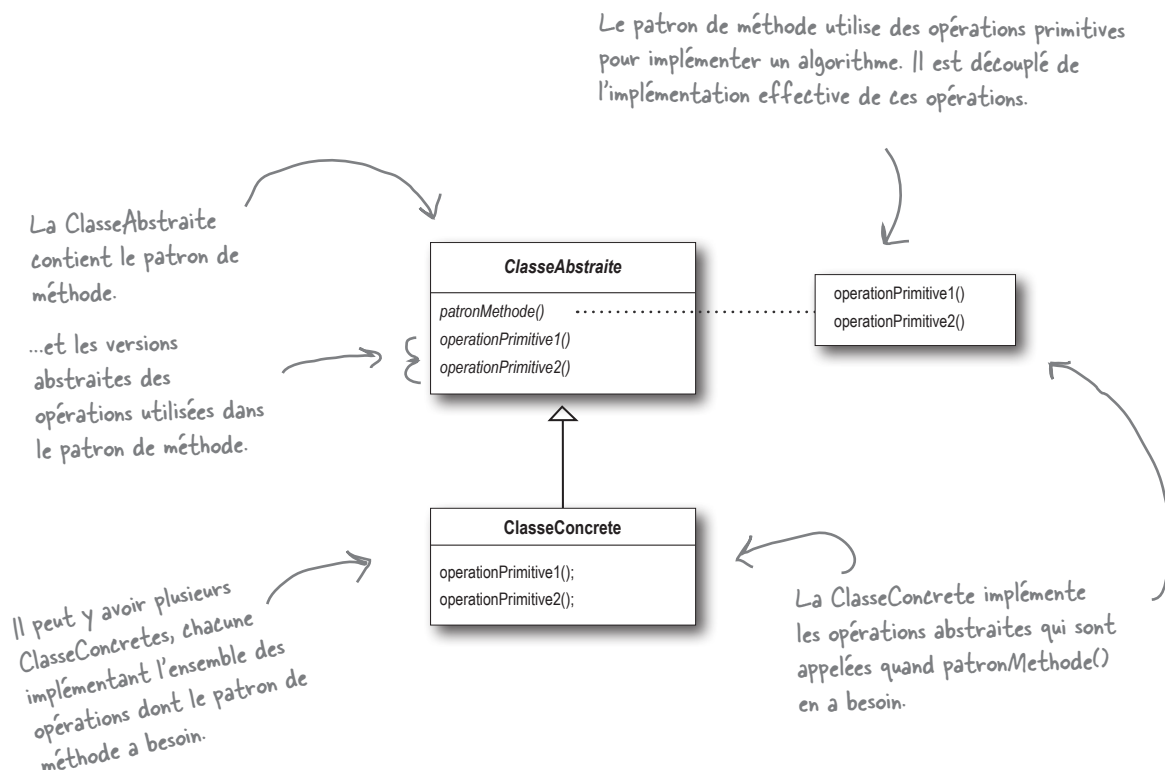
Le pattern Patron de méthode : définition

Vous avez vu comment le pattern Patron de méthode fonctionne dans notre exemple de The et de Cafe. Voyons maintenant la définition officielle et étudions tous les détails :

Le pattern Patron de méthode définit le squelette d'un algorithme dans une méthode, en déléguant certaines étapes aux sous-classes. Patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de celui-ci.

Ce pattern est entièrement dédié à la création d'un patron d'algorithme. Qu'est-ce qu'un patron ? Comme vous l'avez constaté, c'est simplement une méthode. Plus spécifiquement, c'est une méthode qui définit un algorithme sous la forme d'une suite d'étapes. Une ou plusieurs de ces étapes sont définies abstraites et sont implémentées par une sous-classe. Cela permet à la structure de l'algorithme de demeurer inchangée, tandis que les sous-classes fournissent une partie de l'implémentation.

Observons le diagramme de classes :





Code à la loupe

Regardons de plus près comment la ClasseAbstraite est définie, notamment le patron de méthode et les opérations primitives.

Voici notre classe abstraite. Elle est déclarée abstraite et conçue pour être sous-classée par les classes qui fournissent les implémentations des opérations.

```
abstract class ClasseAbstraite {  
  
    final void patronMethode() {  
        operationPrimitive1();  
        operationPrimitive2();  
        operationConcrete();  
    }  
  
    abstract void operationPrimitive1();  
    abstract void operationPrimitive2();  
  
    void operationConcrete() {  
        // implémentation de l'opération  
    }  
}
```

Voici le patron de méthode. La méthode est déclarée finale pour empêcher les sous-classes de modifier la séquence d'étapes de l'algorithme.

Le patron de méthode définit la séquence d'étapes, chacune d'elles étant représentée par une méthode.

Dans cet exemple, deux des opérations primitives doivent être implémentées par les sous-classes concrètes.

Nous avons également une opération concrète définie dans la classe abstraite. Des détails sur ces types de méthodes dans un moment...



Code au microscope

Nous allons maintenant regarder d'encore plus près les types de méthode que peut contenir la classe abstraite :

Nous avons modifié `patronMethode()` qui inclut maintenant un nouvel appel de méthode.

```
abstract class ClasseAbstraite {

    final void patronMethode() {
        operationPrimitive1();
        operationPrimitive2();
        operationConcrete();
        adapter();
    }

    abstract void operationPrimitive1();

    abstract void operationPrimitive2();

    final void operationConcrete() {
        // implémentation de l'opération
    }

    void adapter() {}

}
```

Nous avons toujours nos méthodes primitives. Elles sont abstraites et implémentées par les sous-classes concrètes.

Une opération concrète est définie dans la classe abstraite. Celle-ci est déclarée finale afin que les sous-classes ne puissent pas la redéfinir. Elle peut être utilisée directement par le patron de méthode, ou bien être utilisée par les sous-classes.

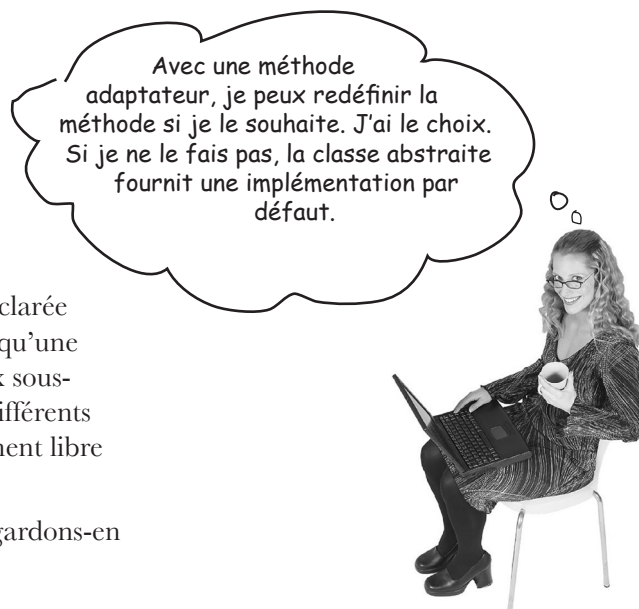
Une méthode concrète, mais qui ne fait rien !

Nous pouvons également avoir des méthodes concrètes qui ne font rien par défaut. Nous les appelons « adaptateurs ». Les sous-classes sont libres de les redéfinir, mais elles n'y sont pas obligées. Nous allons comprendre leur utilité page suivante.

Méthodes adaptateurs et Patron de méthode...

Une méthode adaptateur est une méthode qui est déclarée dans la classe abstraite, mais à laquelle on ne donne qu'une implémentation vide ou par défaut. Cela confère aux sous-classes la capacité de « s'adapter » à l'algorithme à différents endroits si elles le veulent ; une sous-classe est également libre d'ignorer l'adaptateur.

Les méthodes adaptateurs ont plusieurs emplois. Regardons-en une maintenant. Nous en verrons d'autres plus tard :



```
public abstract class BoissonAvecAdaptateur {

    void suivreRecette() {
        faireBouillirEau();
        preparer();
        verserDansTasse();
        if (clientVeutSupplements()) {
            ajouterSupplements();
        }
    }

    abstract void preparer();

    abstract void ajouterSupplements();

    void faireBouillirEau() {
        System.out.println("Portage de l'eau à ébullition");
    }

    void verserDansTasse() {
        System.out.println("Remplissage de la tasse");
    }

    boolean clientVeutSupplements() {
        return true;
    }
}
```

Nous avons ajouté une petite instruction conditionnelle dont le succès dépend d'une méthode concrète, `clientVeutSupplements()`. Si et seulement si le client VEUT des suppléments, nous appelons `ajouterSupplements()`.

Ici, nous avons défini une méthode dont l'implémentation par défaut est (pratiquement) vide. Cette méthode se contente de retourner `true` et ne fait rien d'autre.

C'est une méthode adaptateur parce que la sous-classe peut la redéfinir mais qu'elle n'y est pas obligée.

Utiliser la méthode adaptateur

Pour utiliser la méthode adaptateur, nous la redéfinissons dans notre sous-classe. Ici, l'adaptateur contrôle si la BoissonCafeinee évalue une certaine partie de l'algorithme ; autrement dit, si elle ajoute un supplément à la boisson.

Comment savons-nous si le client veut un supplément ? Il suffit de demander !

```
public class CafeAvecAdaptateur extends BoissonAvecAdaptateur {
```

```
    public void preparer() {
        System.out.println("Passage du café");
    }
```

```
    public void ajouterSupplements() {
        System.out.println("Ajout du lait et du sucre");
    }
```

```
    public boolean clientVeutSupplements() {
        String reponse = getReponseUtilisateur();

        if (reponse.toLowerCase().startsWith("o")) {
            return true;
        } else {
            return false;
        }
    }
```

```
    private String getReponseUtilisateur() {
        String reponse = null;
```

```
        System.out.print("Voulez-vous du lait et du sucre (o/n)? ");
```

```
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

```
        try {
            reponse = in.readLine();
        } catch (IOException ioe) {
            System.err.println("Erreur d'ES. Choix non proposé");
        }
```

```
        if (reponse == null) {
            return "non";
        }
```

```
        return reponse;
    }
```

```
}
```

Voilà où vous redéfinissez la méthode adaptateur et où vous fournissez votre propre fonctionnalité

Lire la décision de l'utilisateur sur les suppléments et retourner true ou false, en fonction de l'entrée.

Ce code demande à l'utilisateur s'il veut du lait et du sucre et lit l'entrée sur la ligne de commande.

Exécuter le test

Bien. L'eau est en train de bouillir... Voici le code de test dans lequel nous créons un thé et un café bien chauds.

```
public class TestBoisson {
    public static void main(String[] args) {

        TheAvecAdaptateur theAdapt = new TheAvecAdaptateur();
        CafeAvecAdaptateur cafeAdapt = new CafeAvecAdaptateur();

        System.out.println("\nPréparation du thé...");
        theAdapt.suivreRecette();

        System.out.println("\nPréparation du café...");
        cafeAdapt.suivreRecette();
    }
}
```

← Créer un thé.
← Un café.

Et appeler suivreRecette() sur les deux !

Exécutons-le...

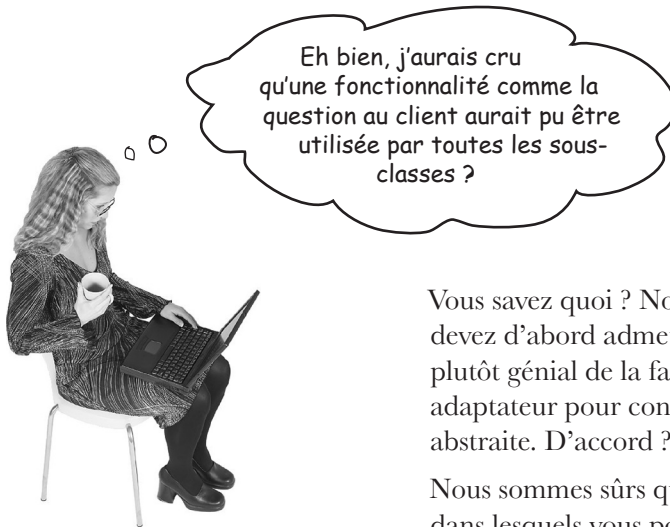
```
Fichier Édition Fenêtre Aide LibertéEgalitéFraternité
%java TestBoisson

Préparation du thé...
Portage de l'eau à ébullition
Infusion du thé
Remplissage de la tasse
Voulez-vous du citron (o/n)? o
Ajout du citron

Préparation du café...
Portage de l'eau à ébullition
Passage du café
Remplissage de la tasse
Voulez-vous du lait et du sucre (o/n)? n
%
```

Une tasse de thé fumante, et oui, bien sûr, nous voulons du citron !

Et une bonne tasse de café bien chaud, mais nous surveillons notre ligne et nous le prenons noir.



Vous savez quoi ? Nous sommes d'accord avec vous. Mais vous devez d'abord admettre que vous pensiez que c'était un exemple plutôt génial de la façon dont on peut utiliser une méthode adaptateur pour contrôler le flot d'un algorithme dans la classe abstraite. D'accord ?

Nous sommes sûrs que vous voyez bien d'autres scénarios réalistes dans lesquels vous pourriez utiliser le patron de méthode et des méthodes adaptateurs dans votre propre code.

Il n'y a pas de questions stupides

Q: Quand on crée un patron de méthode, comment savoir quand utiliser des méthodes abstraites et quand utiliser des méthodes adaptateurs ?

R: Utilisez des méthodes abstraites quand votre sous-classe DOIT fournir une implémentation de méthode ou une étape de l'algorithme. Utilisez des adaptateurs quand cette partie de l'algorithme est optionnelle. Une sous-classe peut choisir d'implémenter ces méthodes adaptateurs, mais elle n'y est pas obligée.

Q: À quoi les méthodes adaptateurs sont-elles censées réellement servir ?

R: Il y a plusieurs emplois des méthodes adaptateurs. Comme nous venons de le dire, elles peuvent fournir à une sous-classe un moyen d'implémenter une partie optionnelle d'un algorithme, ou de la sauter si elle n'est pas importante pour l'implémentation de la sous-classe.

Un autre emploi consiste à donner à la sous-classe une chance de réagir à une étape du patron de méthode qui est sur le point de se produire ou qui vient de se produire. Par exemple, une méthode adaptateur comme `listeReordonnée()` permet à la sous-classe d'exécuter une activité (comme afficher de nouveau une représentation à l'écran) après qu'une liste interne a été réordonnée. Comme vous l'avez constaté, une méthode adaptateur peut également conférer à une sous-classe la capacité de prendre une décision pour la classe abstraite.

Q: Une sous-classe doit-elle implémenter toutes les méthodes abstraites de la `ClasseAbstraite` ?

R: Oui. Chaque sous-classe concrète définit l'ensemble des méthodes abstraites et fournit une implémentation complète des étapes non définies de l'algorithme du patron de méthode.

Q: On dirait que je ferais mieux d'avoir un petit nombre de méthodes abstraites si je ne veux pas avoir trop de travail pour les implémenter dans la sous-classe.

R: Il est bon d'en être conscient lorsqu'on écrit des patrons de méthodes. On y parvient parfois en évitant que la granularité des étapes de l'algorithme soit trop fine. Mais il s'agit de toute évidence d'un compromis : moins la granularité est fine, moins il y a de souplesse. N'oubliez pas non plus que certaines étapes seront optionnelles : vous pouvez donc les implémenter sous forme de méthodes adaptateurs au lieu de classes abstraites, ce qui allège le fardeau imposé aux sous classes de votre classe abstraite.

Le principe d'Hollywood

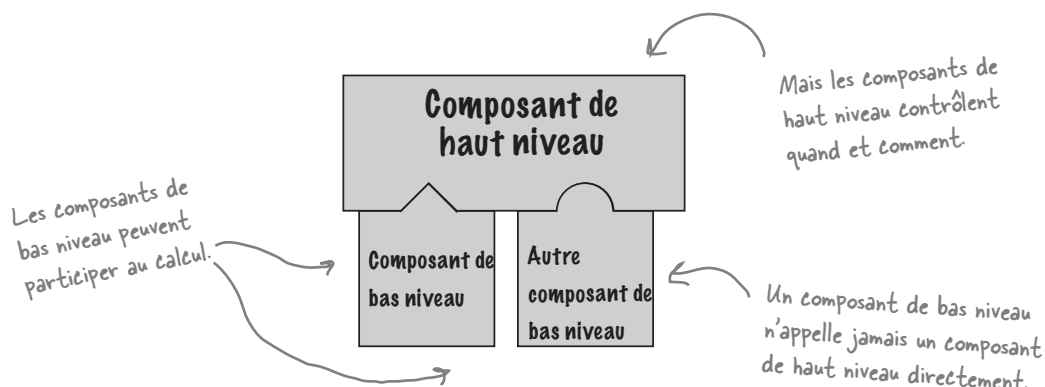
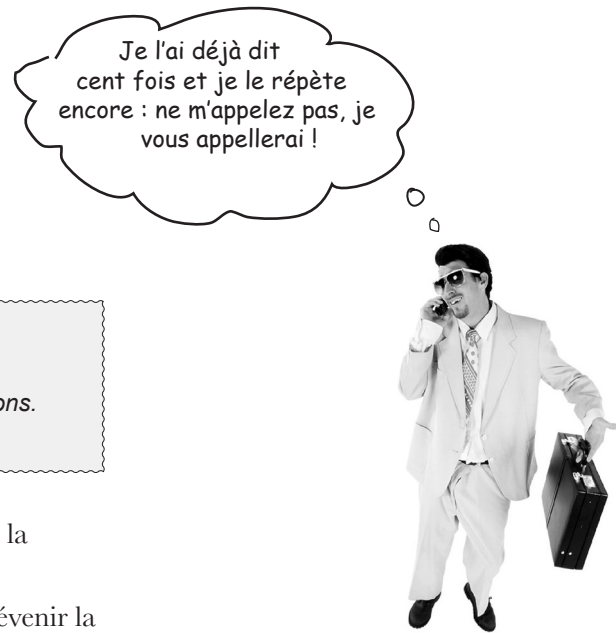
Nous avons un autre principe de conception pour vous. Il s'appelle le Principe d'Hollywood :



Facile à mémoriser, non ? Mais quel est le rapport avec la conception OO ?

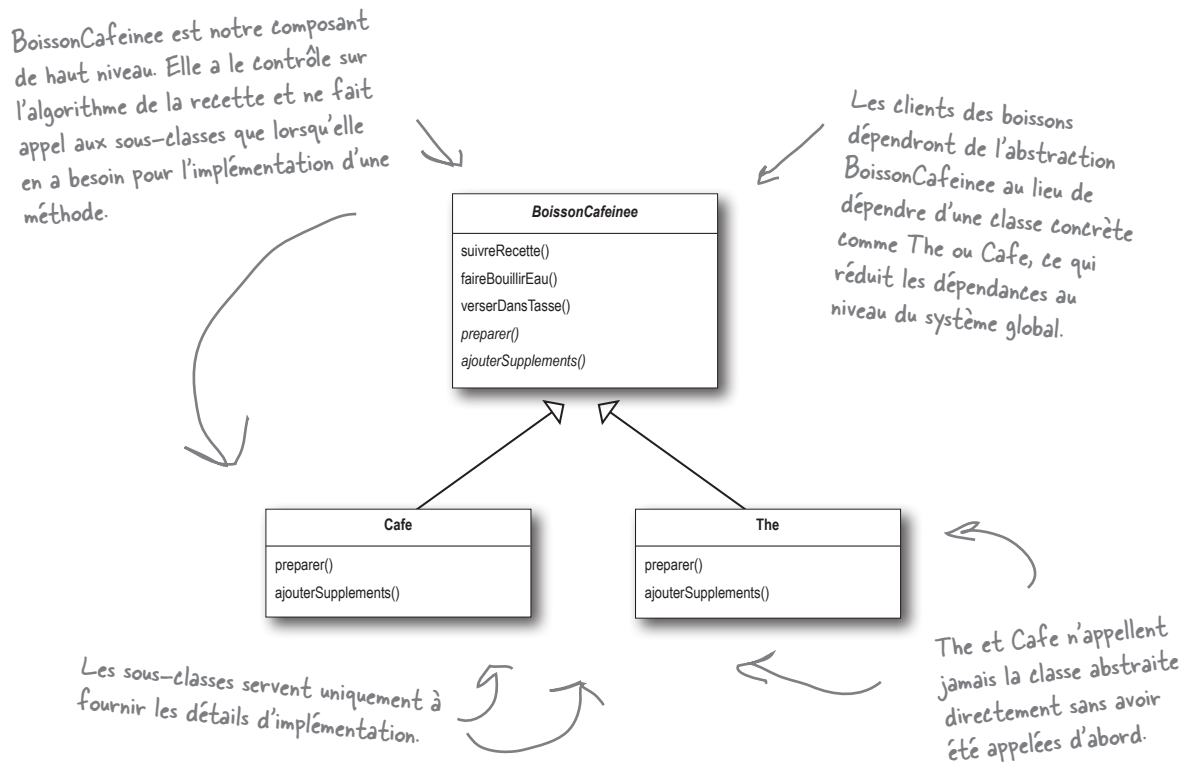
Le principe d'Hollywood nous fournit un moyen de prévenir la « gangrène des dépendances ». Il y a gangrène des dépendances lorsqu'on a des composants de haut niveau qui dépendent de composants de bas niveau qui dépendent de composants de haut niveau qui dépendent de composants transverses qui dépendent de composants de haut niveau, et ainsi de suite. Quand la gangrène s'installe, plus personne ne parvient à comprendre la façon dont le système a été conçu.

Avec le principe d'Hollywood, nous permettons aux composants de bas niveau de s'adapter à un système, mais les composants de haut niveau déterminent quand on en a besoin et comment. Autrement dit, les composants de haut niveau traitent les composants de bas niveau à la mode d'Hollywood : « ne nous appelez pas, nous vous appellerons ».



Le principe d'Hollywood et le Patron de méthode

La connexion entre le principe d'Hollywood et le pattern Patron de méthode est probablement assez apparente : quand nous appliquons le pattern Patron de méthode, nous disons aux sous-classes : « ne nous appelez pas, nous vous appellerons ». Comment ? Observons de nouveau la conception de notre BoissonCafeinee :



MUSCLEZ VOS NEURONES

Quels autres patterns utilisent le principe d'Hollywood ?

Fabrication, Observation... En voyez-vous d'autres ?

Il n'y a pas de questions stupides

Q: Y a-t-il un rapport entre le principe d'Hollywood et le principe d'inversion des dépendances que nous avons vu il y a quelques chapitres ?

R: Le principe d'inversion des dépendances nous apprend comment éviter l'emploi de classes concrètes et travailler autant que possible avec des abstractions. Le principe d'Hollywood est une technique pour construire des structures et des composants qui permet d'adapter les

composants de bas niveau sans créer de dépendance entre ceux-ci et les couches supérieures. La finalité de ces deux principes est donc le découplage, mais le principe d'inversion énonce de façon beaucoup plus forte et plus générale le moyen d'éviter des dépendances dans la conception. Le principe d'Hollywood nous offre une technique pour créer des conceptions qui permettent aux structures de bas niveau d'interopérer tout en empêchant les autres classes de devenir trop dépendantes d'elles.

Q: Un composant de bas niveau a-t-il l'interdiction d'appeler une méthode d'un composant de plus haut niveau ?

R: Pas vraiment. En réalité, un composant de bas niveau finira par appeler une méthode définie au-dessus de lui dans la hiérarchie d'héritage, par la simple vertu de l'héritage. Mais nous voulons d'éviter des dépendances circulaires explicites entre les composants de bas niveau et de haut niveau.

Qui fait quoi ?

Faites correspondre chaque pattern à sa description :

Pattern	Description
Patron de méthode	Encapsule des comportements interchangeables et utilise la délégation pour décider quel comportement utiliser
Stratégie	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
Fabrication	Les sous-classes décident des classes concrètes à créer

Patrons de méthodes à l'état sauvage

Le pattern Patron de méthode est un pattern très courant et vous en trouverez des quantités dans le monde réel. Mais vous devrez avoir l'œil, parce qu'il en existe de nombreuses implémentations qui ne ressemblent pas vraiment à sa définition officielle.

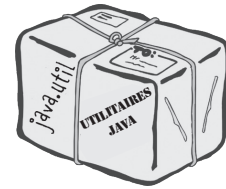
La fréquence d'apparition de ce pattern s'explique par le fait que c'est un excellent outil pour concevoir des structures : la structure contrôle l'exécution de l'algorithme, mais elle laisse son utilisateur spécifier les détails de ce qui se passe réellement à chaque étape de l'algorithme.

Un petit safari va nous permettre d'observer quelques emplois du pattern en pleine nature (bon, d'accord, dans l'API Java)...

En formation, nous étudions les patterns classiques. Mais dehors, dans le monde réel, nous devons apprendre à reconnaître les patterns hors contexte. Nous devons également apprendre à reconnaître les variantes des patterns, parce que, dans le monde réel, un trou carré n'est pas toujours vraiment un trou carré.



Trier avec Patron de méthode



Qu'est-ce qu'on fait très souvent avec les tableaux ?

On les trie !

Conscients de ce fait, les concepteurs de la classe Java Arrays ont mis à notre disposition un patron de méthode bien pratique pour trier. Jetons un coup d'œil à la façon dont cette méthode opère :

Nous avons un peu abrégé ce code pour faciliter l'explication. Si vous voulez en voir la totalité, téléchargez le source sur le site de Sun et étudiez-le...

En réalité, nous avons là deux méthodes qui collaborent pour fournir la fonctionnalité de tri.

La première méthode, `sort()`, n'est qu'une méthode auxiliaire qui crée une copie du tableau et la transmet comme tableau de destination à la méthode `mergeSort()`. Elle transmet également la taille du tableau et dit à la méthode `sort` de commencer au premier élément.

```
public static void sort(Object[] a) {  
    Object aux[] = (Object[])a.clone();  
    mergeSort(aux, a, 0, a.length, 0);  
}
```

La méthode `mergeSort()` contient l'algorithme de tri et laisse le soin à une implémentation de la méthode `compareTo()` de terminer l'algorithme.

```
private static void mergeSort(Object src[], Object dest[],  
    int low, int high, int off)  
{  
    for (int i=low; i<high; i++){  
        for (int j=i; j>low &&  
            ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)  
        {  
            swap(dest, j, j-1);  
        }  
    }  
    return;  
}
```

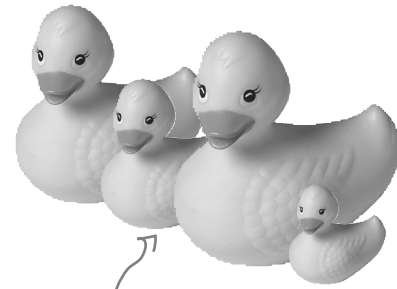
Voici le patron de méthode.

Ceci est une méthode concrète, déjà définie dans la classe Arrays.

`compareTo()` est la méthode que nous devons implémenter pour « remplir » le patron de méthode.

Nous avons des canards à trier...

Disons que vous avez un tableau de canards et que vous voulez le trier. Comment procédez-vous ? Eh bien, le patron de méthode sort de la classe Arrays nous fournit l'algorithme, mais vous devez lui indiquer comment comparer les canards, ce que vous faites en implémentant la méthode `compareTo()`... Logique, non ?



Nous avons un tableau de canards à trier.



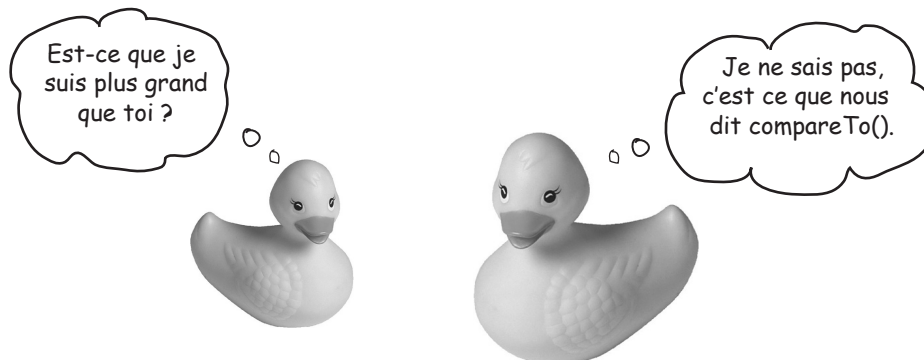
Non. Ne sommes-nous pas censé sous-classer quelque chose ? Je pensais que c'était le but de Patron de méthode. Un tableau ne sous-classant rien, je ne vois pas comment on pourrait utiliser `sort()`.

Très juste. Voilà le truc : comme les concepteurs de `sort()` voulaient qu'on puisse l'utiliser dans tous les tableaux, ils ont donc fait de `sort()` une méthode statique qu'on peut appeler de n'importe où. Mais, c'est vrai, elle fonctionne presque comme si elle était dans une superclasse. Maintenant, précisons un détail : puisque la méthode `sort()` n'est pas définie dans notre superclasse, elle a besoin de savoir que vous avez implémenté la méthode `compareTo()`, sinon il vous manque l'élément nécessaire pour compléter l'algorithme de tri.

Pour résoudre le problème, les concepteurs ont utilisé l'interface `Comparable`. Il vous suffit d'implémenter cette interface, qui contient une seule méthode nommée (surprise) : `compareTo()`.

Qu'est-ce que `compareTo()` ?

La méthode `compareTo()` compare deux objets et indique si l'un est supérieur, inférieur ou égal à l'autre. `sort()` s'en sert pour comparer les objets du tableau.



Comparer des canards et des canards

Bien. Vous savez donc que si vous voulez trier des Canards, vous devez implémenter cette méthode `compareTo()`. Ce faisant, vous fournissez à la classe `Arrays` ce dont elle a besoin pour compléter l'algorithme et trier vos canards.

Voici l'implémentation pour les canards :



```

public class Canard implements Comparable {
    String nom;
    int poids;

    public Canard(String nom, int poids) {
        this.nom = nom;
        this.poids = poids;
    }

    public String toString() {
        return nom + " pèse " + poids;
    }

    public int compareTo(Object object) {
        Canard autreCanard = (Canard)object;
        if (this.poids < autreCanard.poids) {
            return -1;
        } else if (this.poids == autreCanard.poids) {
            return 0;
        } else { // this.poids > autreCanard.poids
            return 1;
        }
    }
}

```

Souvenez-vous : puisque nous ne sous-classons pas réellement, nous devons implémenter l'interface `Comparable`.

Nos Canards ont un nom et un poids.

Restons simples : nos canards se contentent d'afficher leur nom et leur poids !

Voilà ce dont `sort()` a besoin...

`compareTo()` prend un autre Canard et le compare au Canard courant (`this`).

Voici où nous spécifions comment comparer les Canards. Si le Canard courant (`this`) pèse moins que l'autreCanard nous retournons `-1`, s'ils sont égaux nous retournons `0` ; et si le Canard courant pèse plus, nous retournons `1`.

Trions quelques canards...

Voici le test pour trier des Canards...

```

public class TestTriCanards {
    public static void main(String[] args) {
        Canard[] canards = {
            new Canard("Donald", 8),
            new Canard("Riri", 2),
            new Canard("Daisy", 7),
            new Canard("Fifi", 2),
            new Canard("Picsou", 10),
            new Canard("Loulou", 2)
        };

        System.out.println("Avant le tri :");
        afficher(canards);

        Arrays.sort(canards);

        System.out.println("\nAprès le tri :");
        afficher(canards);
    }

    public static void afficher(Canard[] canards) {
        for (int i = 0; i < canards.length; i++) {
            System.out.println(canards[i]);
        }
    }
}

```

Il nous faut un tableau de Canards. Ceux-ci n'ont pas l'air mal.

Remarquez que nous appelons la méthode statique sort() de la classe Arrays, et que nous lui transmettons nos Canards.

Affichons-les pour voir leur nom et leur poids.

C'est le moment de trier !

Affichons-les (encore) pour voir leur nom et leur poids.

Que le tri commence !

```

Fichier Edition Fenêtre Aide PicsouDoitSuivreUnRégime
%java TestTriCanards

Avant le tri :
Donald pèse 8
Riri pèse 2
Daisy pèse 7
Fifi pèse 2
Picsou pèse 10
Loulou pèse 2

Après le tri :
Riri pèse 2
Fifi pèse 2
Loulou pèse 2
Daisy pèse 7
Donald pèse 8
Picsou pèse 10
%

```

Les Canards en vrac

Les Canards triés

Les secrets de la machine à trier les canards



Observons pas à pas le fonctionnement du patron de méthode `sort()` de la classe `Arrays`. Nous verrons comment il contrôle l'algorithme, et, à certains points, comment il demande à nos Canards de fournir l'implémentation d'une étape...

- 1 Tout d'abord, il nous faut un tableau de Canards :

```
Canard[] canards = {new Canard("Donald", 8), ... };
```

- 2 Puis nous appelons le patron de méthode `sort()` de la classe `Arrays` et nous lui transmettons nos canards :

```
Arrays.sort(canards);
```

La méthode `sort()` et sa méthode auxiliaire `mergeSort()` contrôlent la procédure de tri.

- 3 Pour trier un tableau, il faut comparer tous les éléments deux par deux jusqu'à ce que toute la liste soit triée.

Pour comparer deux canards, la méthode `sort()` s'en remet à la méthode `compareTo()` du `Canard`, puisqu'elle sait comment faire. La méthode `compareTo()` est appelée sur le premier canard et on lui transmet le canard auquel le comparer :

```
canards[0].compareTo(canards[1]);
```

Premier Canard Canard auquel comparer

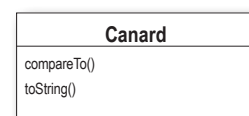
- 4 Si les Canards ne sont pas dans l'ordre, ils sont permutés grâce à la méthode concrète `swap()` de la classe `Arrays` :

```
swap()
```

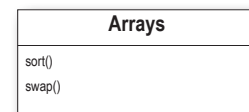
- 5 La méthode `sort()` continue à comparer et à permuter les Canards jusqu'à ce que le tableau soit dans l'ordre correct !

```
for (int i=low; i<high; i++){
    ... compareTo() ...
    ... swap() ...
}
```

La méthode `sort()` contrôle l'algorithme, aucune classe ne peut le changer. `sort()` compte sur une classe implémentant `Comparable` pour fournir l'implémentation de `compareTo()`.



Pas d'héritage, contrairement au patron de méthode typique.



Il n'y a pas de questions stupides

Q: Est-ce que c'est vraiment le Patron de méthode, ou est-ce que vous cherchez la petite bête ?

R: Le pattern permet d'implémenter un algorithme et de laisser les sous-classes fournir l'implémentation les étapes. Il est clair que ce n'est pas le rôle de la méthode `sort()` de la classe `Arrays`! Mais, comme nous le savons, les patterns du monde réel ne ressemblent pas toujours exactement à ceux des manuels. Il faut modifier ces derniers pour qu'ils s'adaptent au contexte et respectent les contraintes d'implémentation. Les concepteurs de `sort()` étaient confrontés à certaines contraintes. En général, on ne peut pas sous-classer un tableau Java et ils voulaient que `sort()` soit utilisable sur tous les tableaux (et chaque tableau est une classe différente). Ils ont donc défini une méthode statique en laissant aux éléments à trier la partie « comparaison » de l'algorithme.

Donc, si ce n'est pas un patron de méthode classique, cette implémentation est toujours dans l'esprit du pattern Patron de méthode. De plus, en éliminant la nécessité de sous-classer `Arrays` pour utiliser cet algorithme, ils ont d'une certaine façon rendu le tri plus souple et plus pratique.

Q: Cette implémentation du tri ressemble plutôt au pattern Stratégie qu'au pattern Patron de méthode. Pourquoi le considère-t-on comme un Patron de méthode ?

R: Vous pensez sans doute cela parce que le pattern Stratégie utilise la composition des objets. Dans un sens, vous avez raison : comme nous utilisons l'objet `Arrays` pour trier note tableau, cela ressemble à Stratégie. Mais souvenez-vous : dans Stratégie, la classe avec laquelle vous composez implémente la totalité de l'algorithme. Celui que `Arrays` implémente pour trier

est incomplet ; il a besoin d'une classe pour « remplir » la méthode `compareTo()` manquante. C'est en cela que c'est plutôt un Patron de méthode.

Q: Y a-t-il d'autres exemples de patrons de méthode dans l'API Java ?

R: Oui, vous en trouverez quelques uns. Par exemple, `java.io` possède dans `InputStream` une méthode `read()` que les sous-classes doivent implémenter et elle est utilisée par le patron de méthode `read(byte b[], int off, int len)`.



MUSCLEZ VOS NEURONES

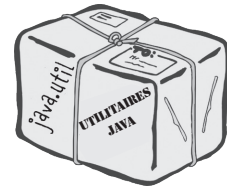
Nous savons que nous devons préférer la composition à l'héritage, d'accord ? Mais les concepteurs du patron de méthode `sort()` ont décidé de ne pas utiliser l'héritage et d'implémenter `sort()` comme une méthode statique qui est composée avec un `Comparable` au moment de l'exécution. En quoi est-ce mieux ? En quoi est-ce pire ? Comment traiteriez-vous ce problème ? Les tableaux Java le rendent-ils particulièrement délicat ?



MUSCLEZ² VOS NEURONES

Pensez à un autre pattern qui est une spécialisation du patron de méthode. Dans cette spécialisation, on utilise des opérations primitives pour créer et retourner des objets.

Les adaptateurs de Swing



Continuons notre safari... Soyez vigilant, vous allez voir des **JFrames** !

Si vous n'avez jamais rencontré de **JFrame**, c'est le conteneur Swing le plus élémentaire, un cadre, et il hérite d'une méthode `paint()`. Par défaut, `paint()` ne fait rien parce que c'est une méthode adaptateur ! En redéfinissant `paint()`, vous pouvez vous insérer dans l'algorithme du **JFrame** pour afficher sa surface à l'écran et y incorporer vos propres éléments graphiques. Voici un exemple affreusement simple d'utilisation d'un **JFrame** et de redéfinition de la méthode `paint()` :

```
public class MonCadre extends JFrame {  
  
    public MonCadre(String titre) {  
        super(titre);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        this.setSize(300,300);  
        this.setVisible(true);  
    }  
  
    public void paint(Graphics graphics) {  
        super.paint(graphics);  
        String msg = "Je suis le maître du monde !!";  
        graphics.drawString(msg, 100, 100);  
    }  
  
    public static void main(String[] args) {  
        MonCadre monCadre = new MonCadre("Design Patterns Tête la Première");  
    }  
}
```

Nous étendons **JFrame** qui contient une méthode `update()` qui contrôle l'algorithme pour mettre à jour l'écran.
Nous pouvons nous adapter à cet algorithme en redéfinissant la méthode adaptateur `paint()`.

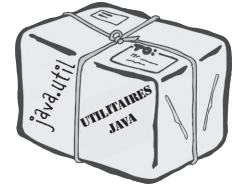
Ne regardez pas derrière le rideau !
Ce n'est qu'un peu d'initialisation...

L'algorithme de mise à jour du **JFrame** appelle `paint()`. Par défaut, `paint()` ne fait rien... c'est un adaptateur. Nous redéfinissons `paint()` et disons au **JFrame** d'afficher un message dans la fenêtre.



Voici le message qui est affiché dans le cadre puisque nous avons utilisé la méthode `paint()`.

Applets



Dernière étape du safari : l'applet.

Vous savez probablement qu'une applet est un petit programme qui s'exécute dans une page web. Toute applet doit sous-classer Applet, et cette classe fournit plusieurs méthodes adaptateurs. Regardons-en quelques-unes :

```
public class MonApplet extends Applet {
    String message;

    public void init() {
        message = "Bonjour, me voilà!";
        repaint();
    }

    public void start() {
        message = "Je démarre...";
        repaint();
    }

    public void stop() {
        message = "Oh, on m'arrête...";
        repaint();
    }

    public void destroy() {
        // l'applet s'en va...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

La méthode adaptateur `init()` permet à l'applet de faire ce qu'elle veut pour s'initialiser la première fois.

`repaint()` est une méthode concrète de la classe `Applet` qui permet aux composants de plus haut niveau de savoir que l'applet doit être redessinée.

La méthode adaptateur `start()` permet à l'applet de faire quelque chose quand elle est sur le point d'être affichée sur la page web.

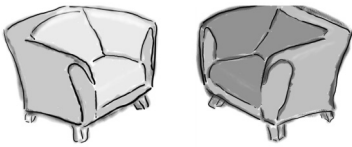
Si l'utilisateur va à une autre page, la méthode adaptateur `stop()` est appelée et l'applet peut faire ce qu'il faut pour arrêter ses actions.

Et l'adaptateur `destroy()` est utilisé quand l'applet va être détruite, par exemple quand on ferme le panneau du navigateur.

Nous pourrions afficher quelque chose, mais quel serait l'intérêt ? Hé, regardez ! Notre vieille amie la méthode `paint()` ! Applet l'emploie également comme méthode adaptateur.

Les applets concrètes font un usage intensif des méthodes adaptateurs pour fournir des comportements spécifiques. Puisque ces méthodes sont des adaptateurs, l'applet n'est pas tenue de les implémenter.

Face à Face



Le face à face de ce soir : **Patron de méthode et stratégie comparent leurs méthodes.**

Patron de méthode

Hé, Stratégie, qu'est-ce que vous faites dans mon chapitre ? Je pensais qu'on allait me coller quelqu'un de barbant comme Fabrication.

Je plaisantais ! Mais sérieusement, qu'est-ce que vous faites là ? Il y a huit chapitres qu'on n'a pas entendu parler de vous !

Vous pourriez peut-être rappeler au lecteur ce que vous faites, depuis le temps.

Hé, voilà qui ressemble beaucoup à ce que je fais. Mais mon rôle est un peu différent du vôtre. Ma tâche consiste à définir les grandes lignes d'un algorithme et à laisser mes sous-classes faire le reste du travail. De cette façon, je peux avoir différentes implémentations des étapes individuelles d'un algorithme tout en gardant le contrôle de la structure. On dirait que vous, vous devez abandonner le contrôle de vos algorithmes.

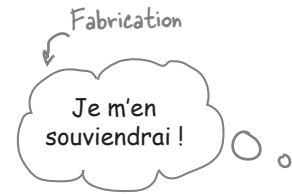
Stratégie

Nan, c'est moi. Mais faites attention, vous êtes parents, Fabrication et vous, si je ne m'abuse ?

J'ai appris que vous mettiez la touche finale à votre chapitre, et je me suis dit que j'allais faire une apparition pour voir où vous en étiez. Nous avons beaucoup de points communs, et je me suis dit que je pourrais peut-être vous aider...

Je ne sais pas, mais, depuis le chapitre 1, les gens m'arrêtent dans la rue en disant « Vous ne seriez pas ce pattern... ». Je pense donc qu'ils savent qui je suis. Mais, pour vous faire plaisir, je définis une famille d'algorithmes et je les rends interchangeable. Comme chacun d'eux est encapsulé, le client peut utiliser facilement des algorithmes différents.

Je ne suis pas sûr qu'« abandonner » soit le bon terme... Et, en tout cas, je ne suis pas obligé d'utiliser l'héritage pour implémenter les algorithmes. Grâce à la composition, je peux offrir aux clients un grand choix d'implémentations.



Patron de méthode

Je m'en souviens. Mais j'ai plus de contrôle sur mon algorithme et je ne duplique pas le code. En fait, si toutes les parties de mon algorithme sont identiques à l'exception de, disons une ou deux lignes, mes classes sont beaucoup plus efficace que les autres. Tout le code commun étant placé dans la superclasse, toutes les sous-classes peuvent le partager.

Ouais, eh bien j'en suis *vraiment* heureux pour vous, mais n'oubliez pas que je suis le *pattern* le plus utilisé. Pourquoi ? Parce que je fournis une méthode fondamentale à la réutilisation du code qui permet aux sous-classes de spécifier des comportements. Je suis sûr que vous voyez que c'est parfait pour créer des frameworks.

Comment donc ? Ma superclasse est abstraite.

Comme je l'ai dit, Stratégie, J'en suis vraiment heureux pour vous. Merci d'être passé, mais j'ai le reste de ce chapitre à terminer.

Compris. Ne nous appelez pas, nous vous appellerons.

Stratégie

Vous *pourriez* être un peu plus efficace (juste un peu) et nécessiter moins d'objets. Et vous pourriez également être un peu moins compliqué par rapport à mon modèle de délégation, mais je suis plus souple parce que j'utilise la composition. Avec moi, les clients peuvent changer leurs algorithmes au moment de l'exécution en utilisant simplement un autre objet Stratégie. Allons, ce n'est pas pour rien qu'on m'a choisi pour le chapitre 1 !

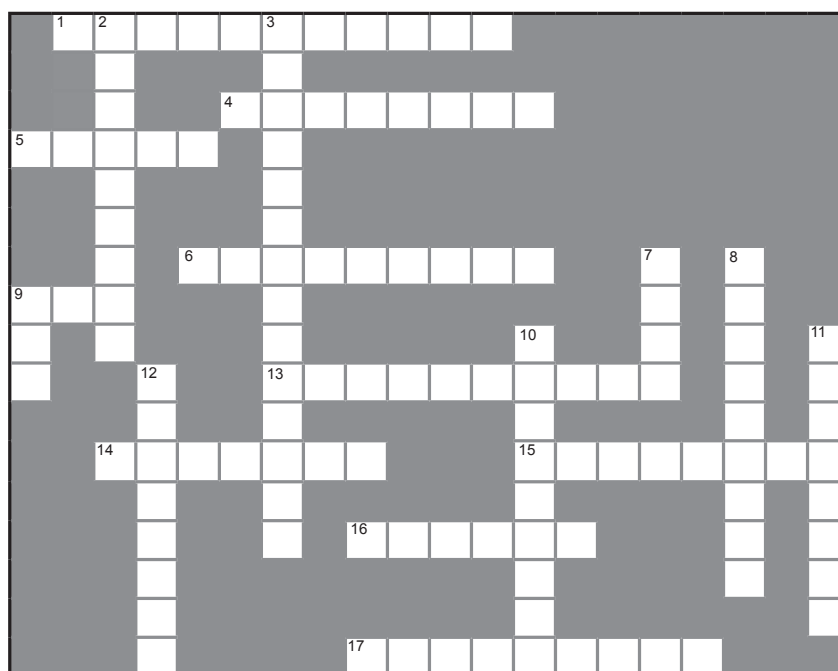
Ouais, j'imagine... Mais qu'en est-il de la dépendance ? Vous êtes beaucoup plus dépendant que moi.

Mais vous dépendez forcément des méthodes implémentées dans votre superclasse, qui font partie de votre algorithme. Moi, je ne dépends de personne : je peux me charger de tout l'algorithme moi-même !

Bon, bon, ne soyez pas susceptible. Je vous laisse travailler, mais si jamais vous avez besoin de mes techniques spéciales, faites-le moi savoir. Je suis toujours ravi de donner un coup de main.



Il est de nouveau temps...

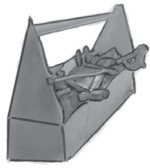


Horizontalement

1. Le pattern Stratégie l'utilise à la place de l'héritage.
4. Le pattern Patron de méthode l'utilise pour déléguer l'implémentation à d'autres classes.
5. La méthode de JFrame que nous redéfinissons pour afficher « Je suis le maître du monde ».
6. « Ne nous appelez pas, nous vous appellerons » est le principe d'_____.
9. On y met du citron.
13. Un patron de méthode définit les étapes d'un _____.
14. Dans ce chapitre, les boissons contiennent de la _____.
15. Mot-clé Java pour l'abstraction.
16. Classe qui aime les pages web.
17. Type de tri utilisé dans la classe Arrays.

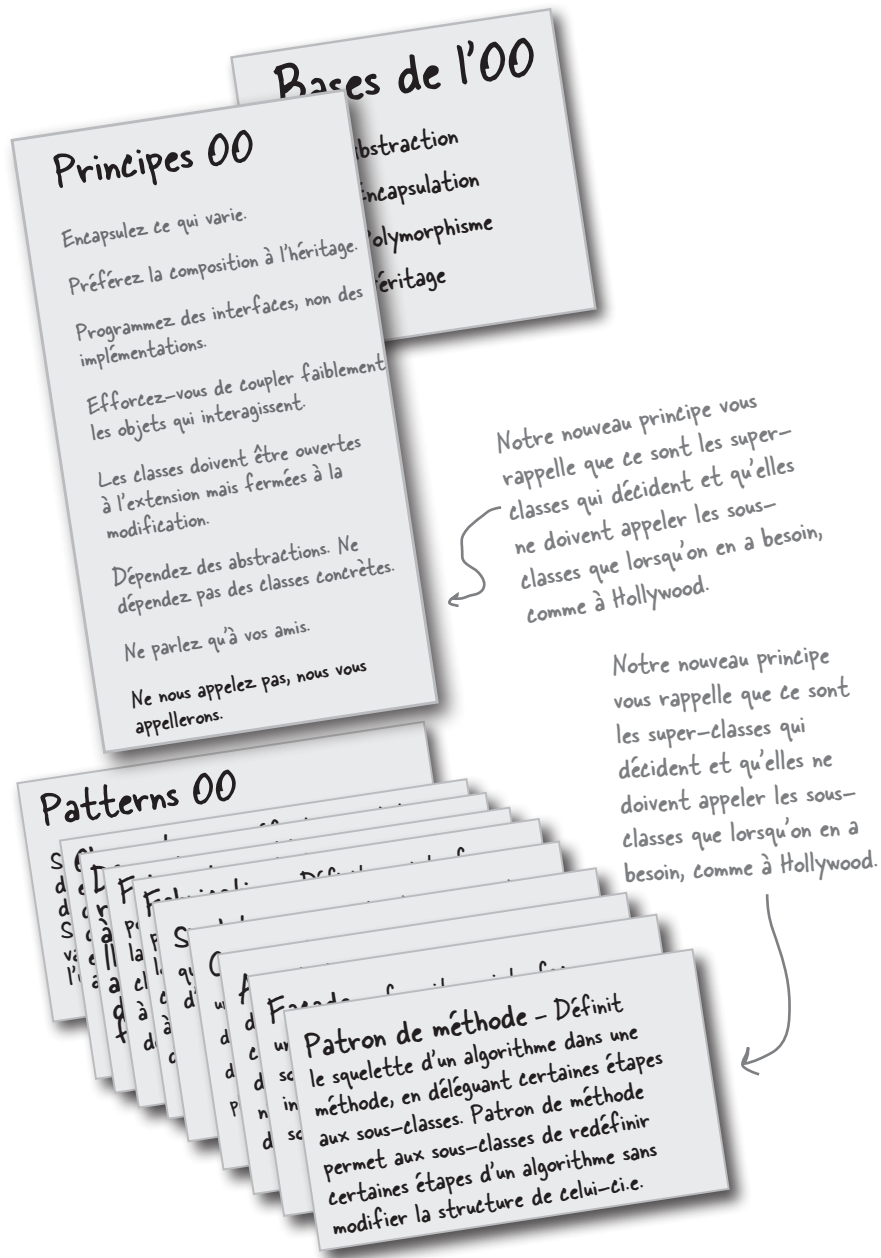
Verticalement

2. Facultatif.
3. Fabrication est une _____ de Patron de méthode.
7. On y met du lait.
8. Le patron de méthode est généralement défini dans une classe _____.
9. sort() est une méthode de _____.
10. Ce pattern a la grosse tête !
11. Dans la classe Arrays, les méthodes réalisant le tri sont des méthodes _____.
12. Notre café préféré à Objectville.



Votre boîte à outils de concepteur

Nous avons ajouté **Patron de méthode** à votre boîte à outils. **Patron de méthode** vous permet de réutiliser du code comme une pro tout en gardant le contrôle de vos algorithmes.



POINTS D'IMPACT

- Un « patron de méthode » définit les étapes d'un algorithme, déléguant aux sous-classes l'implémentation de ces étapes.
- Le pattern Patron de méthode nous fournit une technique importante pour la réutilisation du code.
- La classe abstraite du patron de méthode peut définir des concrètes, des méthodes abstraites et des méthodes adaptateurs.
- Les méthodes abstraites sont implémentées par les sous-classes.
- Les méthodes adaptateurs sont des méthodes qui ne font rien ou qui ont un comportement par défaut dans la classe abstraite, mais qui peuvent être redéfinies dans une sous-classe.
- Pour empêcher les sous-classes de modifier l'algorithme du patron de méthode, déclarez le patron de méthode final.
- Le principe d'Hollywood nous conseille de placer les prises de décision dans les modules de haut niveau, lesquels décident quand et comment appeler les modules de bas niveau.
- Vous rencontrerez beaucoup d'emplois du pattern Patron de méthode dans le code du monde réel, mais n'espérez pas (comme pour tout pattern) qu'ils soient entièrement conçus « dans les règles ».
- Les patterns Stratégie et Patron de méthode Patterns encapsulent tous deux les algorithmes, l'un au moyen de l'héritage l'autre de la composition.
- Fabrication est une spécialisation de Patron de méthode.

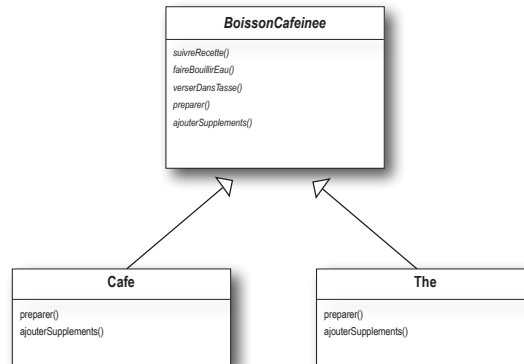


Solutions des exercices

À vos crayons



Tracez le nouveau diagramme de classes maintenant que nous avons transféré l'implémentation de suivreRecette() dans la classe BoissonCafeinee.



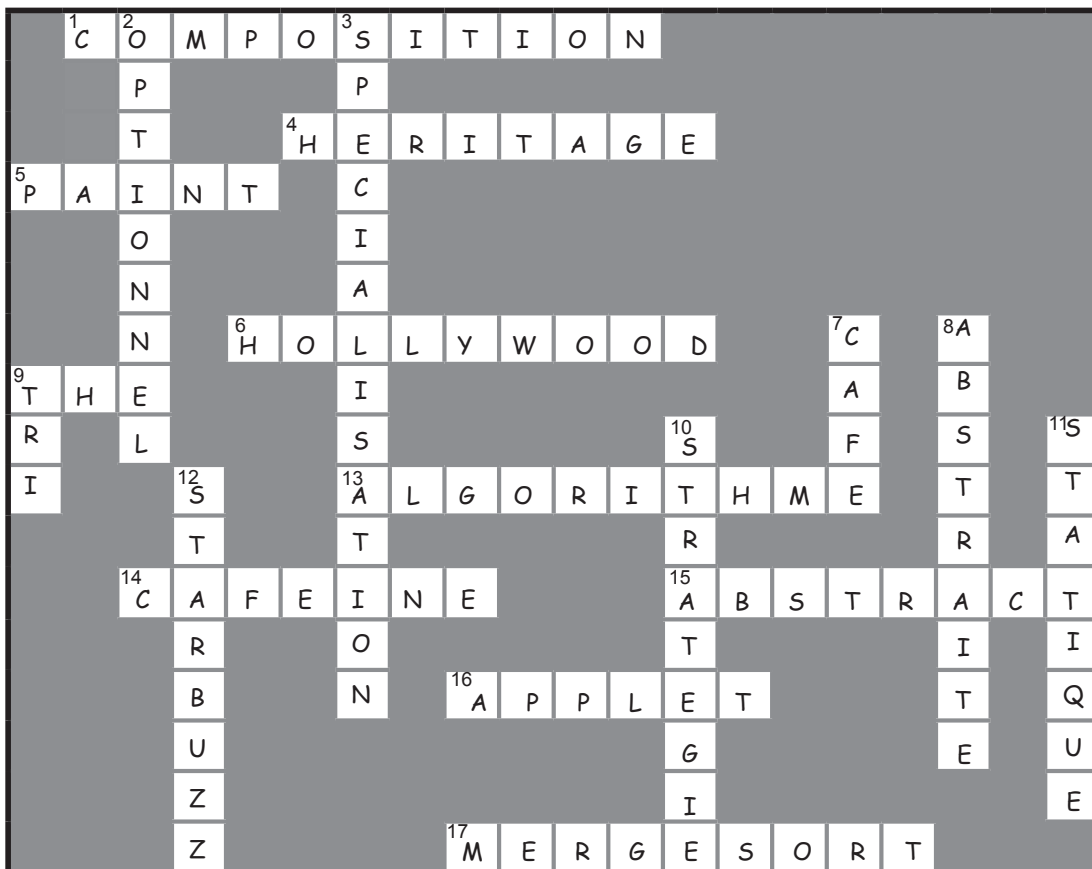
* Qui fait quoi ? *

Faites correspondre chaque pattern à sa description :

Pattern	Description
Patron de méthode	Encapsule des comportements interchangeables et utilise la délégation pour décider quel comportement utiliser
Stratégie	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme.
Fabrication	Les sous-classes décident des classes concrètes à créer.



Solutions des mots-croisés



9 les patterns Itérateur et Composite

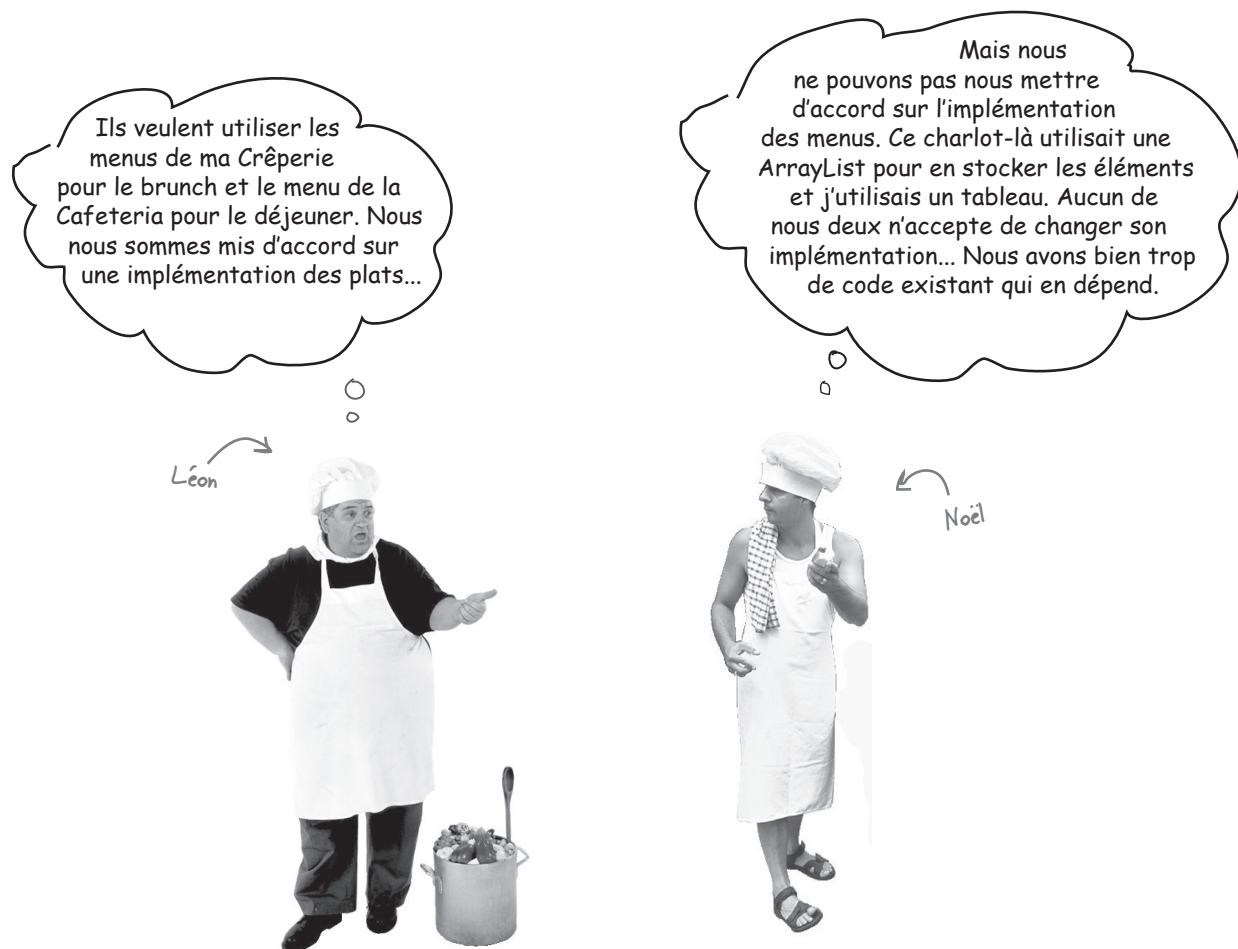
Des collections bien gérées



Il y a des quantités de façon de placer des objets dans une collection. Tableaux, piles, listes, tables de hachage : vous avez le choix. Chacune a ses avantages et ses inconvénients. Mais il y aura toujours un moment où votre client voudra opérer des itérations sur ces objets. Allez-vous alors lui montrer votre implémentation ? Espérons que non ! Ce serait un manque de professionnalisme absolu. Eh bien vous n'avez pas besoin de risquer votre carrière. Vous allez voir comment autoriser vos clients à procéder sans même jeter un coup d'œil à la façon dont vous stockez vos objets. Vous apprendrez également à créer des super collections d'objets qui peuvent parcourir des structures de données impressionnantes d'un seul trait. Et si cela ne suffit pas, vous allez découvrir une chose ou deux sur les responsabilités des objets.

Le scoop de l'année : fusion de la Cafeteria et de la Crêperie d'Objectville

Voilà une excellente nouvelle ! Maintenant, nous allons pouvoir trouver les délicieuses crêpes de la crêperie d'Objectville et les succulents plats de sa Cafeteria au même endroit. Mais on dirait qu'il y a comme un léger problème...



Observons les plats

Au moins, Léon et Noël sont d'accord sur l'implémentation des éléments des menus, les Plats. Observons les plats de chaque menu et jetons également un coup d'œil à l'implémentation.

Le menu de la cafeteria a toutes sortes de plats pour le dîner alors que celui de la crêperie a plein de plats pour le brunch. Chaque plat a un nom, une description et un prix



```
public class Plat {
    String nom;
    String description;
    boolean vegetarien;
    double prix;

    public Plat(String nom,
                String description,
                boolean vegetarien,
                double prix)
    {
        this.nom = nom;
        this.description = description;
        this.vegetarien = vegetarien;
        this.prix = prix;
    }

    public String getNom() {
        return nom;
    }

    public String getDescription() {
        return description;
    }

    public double getPrix() {
        return prix;
    }

    public boolean estVegetarien() {
        return vegetarien;
    }
}
```

Un Plat comprend un nom, une description, un statut pour indiquer s'il convient aux végétariens et un prix. On transmet toutes ces valeurs au constructeur pour initialiser le Plat.

Ces méthodes get vous permettent d'accéder aux champs de l'élément de menu.

Les implémentations de Léon et Noël

Voyons maintenant l'objet de la dispute de Léon et Noël. Ils ont tous deux investi beaucoup de temps et de code dans la façon de stocker les plats, et d'énormes quantités de code dépendent de celle-ci.

J'ai utilisé une ArrayList pour pouvoir ajouter facilement des plats à mon menu.



Voici comment Léon a implémenté le menu de la crêperie.

```
public class MenuCreperie implements Menu {
    ArrayList plats;

    public MenuCreperie() {
        plats = new ArrayList();

        ajouterPlat("Crêpe à l'oeuf",
            "Crêpe avec oeuf au plat ou brouillé",
            true,
            2.99);

        ajouterPlat("Crêpe complète",
            "Crêpe avec oeuf au plat et jambon",
            false,
            2.99);

        ajouterPlat("Crêpe forestière",
            "Myrtilles fraîches et sirop de myrtilles",
            true,
            3.49);

        ajouterPlat("Crêpe du chef",
            "Crème fraîche et fruits rouges au choix",
            true,
            3.59);
    }

    public void ajouterPlat(String nom, String description,
        boolean vegetarien, double prix)
    {
        Plat plat = new Plat(nom, description, vegetarien, prix);
        plats.add(plat);
    }

    public ArrayList getPlats() {
        return plats;
    }

    // autres méthodes
}
```

Léon utilise une ArrayList pour stocker ses plats.

Chaque plat est ajouté à l'ArrayList, ici, dans le constructeur.

Chaque Plat a un nom, une description, un flag indiquant s'il est végétarien ou non et un prix.

Pour ajouter un plat, Léon crée un nouvel objet Plat, lui transmet chaque argument puis l'insère dans l'ArrayList.

La méthode getPlats() retourne la liste des plats.

Léon a un tas d'autre code qui dépend de cette implémentation avec une ArrayList. Il ne veut pas devoir TOUT réécrire !



Haah ! Une ArrayList...
J'ai utilisé un VRAI tableau pour
pouvoir contrôler la taille maximale
de mon menu et accéder à mes
Plats sans coercion de type.

Et voici comment Noël implémente le menu de la Cafeteria.

```
public class MenuCafeteria implements Menu {
    static final int MAX_PLATS = 6;
    int nombreDePlats = 0;
    Plat[] plats;
```

Noël adopte une approche différente : il utilise un tableau
afin de contrôler la taille maximale de son menu et éviter
de devoir sous-typer ses objets.

```
public MenuCafeteria() {
    plats = new Plat[MAX_PLATS];
```

Comme Léon, Noël crée les éléments de son
menu dans le constructeur, grâce à la méthode
auxiliaire ajouterPlat().

```
    ajouterPlat("Salade printanière",
        "Salade verte, tomates, concombre, olives, pommes de terre", true, 2.99);
    ajouterPlat("Salade parisienne",
        "Salade verte, tomates, poulet, emmental", false, 2.99);
    ajouterPlat("Soupe du jour",
        "Soupe du jour et croûtons grillés", false, 3.29);
    ajouterPlat("Quiche aux fruits de mer",
        "Pâte brisée, crevettes, moules, champignons",
        false, 3.05);
    // ajouter ici d'autres plats
}
```

ajouterPlat() accepte tous les arguments
nécessaires pour créer un Plat et en instancie
un. Elle vérifie également que nous n'avons pas
atteint les limites du tableau.

```
public void ajouterPlat(String nom, String description,
    boolean vegetarien, double prix)
{
```

```
    Plat plat = new Plat(nom, description, vegetarien, prix);
    if (nombreDePlats >= MAX_PLATS) {
        System.err.println("Désolé, le menu est plein ! Impossible d'ajouter un plat.");
    } else {
        plats[nombreDePlats] = plat;
        nombreDePlats = nombreDePlats + 1;
    }
}
```

Noël veut explicitement que son menu ne dépasse
pas une taille donnée (probablement pour ne pas
avoir trop de recettes à mémoriser).

```
public Plat[] getPlats() {
    return plats;
}
// autres méthodes
}
```

getPlats() retourne le tableau de plats.

Comme Léon, Noël a un tas de code qui dépend de l'implémentation
de son menu sous forme de tableau. Il est bien trop occupé à faire
la cuisine pour tout réécrire.

Quel est le problème si les deux menus sont représentés différemment ?

Pour voir en quoi deux représentations différentes constituent un problème, essayons d'implémenter un client qui utilise les deux menus. Imaginez que vous ayez été embauché par la nouvelle société constituée par le nouveau propriétaire de la Cafeteria et de la Crêperie pour créer une serveuse compatible Java (nous sommes à Objectville, après tout). Les spécifications précisent que celle-ci peut afficher un menu à la demande des clients, et même vous indiquer si un plat est végétarien sans avoir à demander au chef. Quelle innovation !

Examinons les spécifications, puis nous parcourrons les étapes nécessaires pour les implémenter...



Les spécifications de la serveuse compatible Java

```
Serveuse compatible Java : -nom de code « Alice »

afficherMenu()
- affiche tous les plats au menu

afficherMenuBrunch()
- n'affiche que le menu du brunch

afficherMenuDejeuner()
- n'affiche que le menu du déjeuner

afficherMenuVegetarien()
- affiche tous les plats végétariens

estPlatVegetarien(nom)
- étant donné le nom d'un plat, retourne vrai
si le plat est végétarien, sinon, retourne faux
```


Commençons par voir comment nous implémenterions la méthode `afficherMenu()` :

- 1 Pour afficher tous les plats de chaque menu, vous aurez besoin d'appeler la méthode `getPlat()` de `MenuCreperie` et de `MenuCafeteria` pour extraire leurs plats respectifs. Notez que chacune d'elles retourne un type différent :

```
MenuCreperie menuCreperie = new MenuCreperie();
ArrayList platsBrunch = menuCreperie.getPlats();

MenuCafeteria menuCafeteria = new MenuCafeteria();
Plat[] platsDejeuner = menuCafeteria.getPlats();
```

La méthode semble identique, mais les appels retournent des types différents.

L'implémentation montre son nez : les plats du brunch sont dans une `ArrayList` et ceux du déjeuner sont dans un tableau.

- 2 Maintenant, pour afficher les plats du `MenuCreperie`, nous devons itérer sur les éléments de l'`ArrayList` nommée `platsBrunch`. Et pour afficher ceux de la `Cafeteria`, nous parcourrons le tableau.

```
for (int i = 0; i < platsBrunch.size(); i++) {
    Plat plat = (Plat)platsBrunch.get(i);
    System.out.print(plat.getNom() + " ");
    System.out.println(plat.getPrix() + " ");
    System.out.println(plat.getDescription());
}

for (int i = 0; i < platsDejeuner.length; i++) {
    Plat plat = platsDejeuner[i];
    System.out.print(plat.getNom() + " ");
    System.out.println(plat.getPrix() + " ");
    System.out.println(plat.getDescription());
}
```

Maintenant, nous devons écrire deux boucles différentes pour parcourir les deux implémentations des menus...

...une boucle pour l'`ArrayList`...

et une autre pour le tableau.

- 3 L'implémentation de toute autre méthode de la serveuse sera une variation sur ce thème. Nous devons toujours lire les deux menus et utiliser deux boucles pour parcourir leurs éléments. Si nous fusionnons avec un autre restaurant et que son implémentation est différente, il nous faudra alors *trois* boucles.



À vos crayons

Par rapport à notre implémentation d'afficherMenu(), quelles sont les affirmations qui sont vraies ?

- ☐ A. Nous codons pour des implémentations concrètes de MenuCreperie et de MenuCafeteria, non pour une interfaces.
- ☐ B. La Serveuse n'implémentant pas l'API Java Waitress, elle n'adhère pas à un standard.
- ☐ C. Si nous décidions de passer de l'emploi de MenuCafeteria à un autre type de menu qui implémenterait sa liste de plats sous forme de Hashtable, le code de Serveuse nécessiterait beaucoup de modifications.
- ☐ D. Pour chaque menu, la Serveuse doit connaître la représentation interne de sa collection de plats, ce qui viole l'encapsulation.
- ☐ E. Nous avons du code dupliqué : la méthode afficherMenu() a besoin de deux boucles séparées pour parcourir les deux types de menus. Et si nous ajoutions un troisième menu, il nous faudrait une nouvelle boucle.
- ☐ F. L'implémentation n'étant pas basée sur MXML (Menu XML), elle ne présente pas l'interopérabilité désirable.

Et maintenant ?

Noël et Léon nous mettent dans une position difficile. Ils ne veulent pas modifier leur implémentation parce que cela les obligerait à réécrire une grande quantité de code dans leur classe Menu respective. Mais si l'un des deux ne cède pas, nous aurons la tâche d'implémenter une Serveuse qui sera difficile à maintenir, sans parler d'extensibilité.

Ce serait vraiment génial si nous pouvions trouver un moyen qui leur permettrait d'implémenter la même interface pour leurs menus (leurs approches sont déjà assez semblables, sauf en ce qui concerne le type de retour de leur méthode getPlats()). Ainsi, nous pourrions minimiser les références concrètes dans le code de Serveuse, et probablement nous débarrasser du problème des deux boucles nécessaires pour parcourir les deux menus.

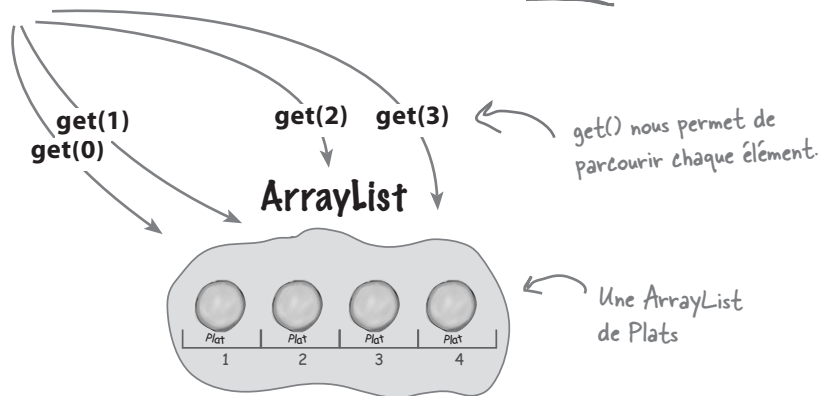
Bonne idée, non ? Mais comment procéder ?

Pouvons-nous encapsuler l'itération ?

Si nous avons appris une seule chose dans ce livre, c'est à encapsuler ce qui varie. Ici, la variation est évidente : c'est l'itération provoquée par la différence entre les collections d'objets retournées par les menus. Mais pouvons-nous l'encapsuler ? Approfondissons cette idée...

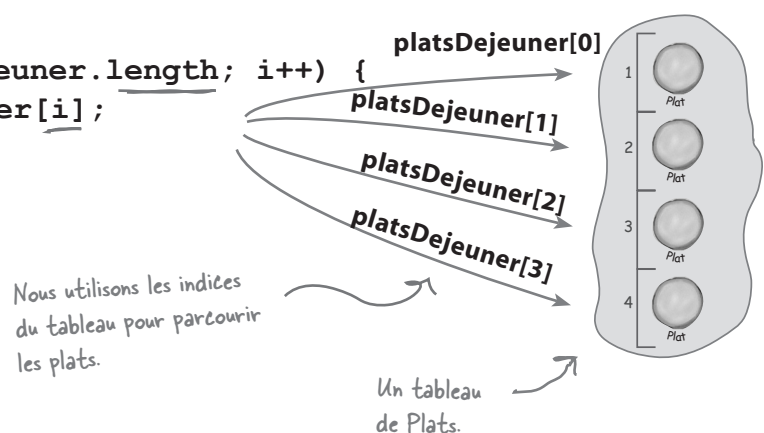
- 1 Pour parcourir les plats du brunch, nous appelons les méthodes `size()` et `get()` sur l'`ArrayList` :

```
for (int i = 0; i < platsBrunch.size(); i++) {
    Plat plat = (Plat)platsBrunch.get(i);
}
```



- 2 Et pour parcourir les plats du déjeuner, nous utilisons l'attribut `length` du tableau et l'opérateur `[]` pour accéder aux éléments du tableau `Plat`.

```
for (int i = 0; i < platsDejeuner.length; i++) {
    Plat plat = platsDejeuner[i];
}
```

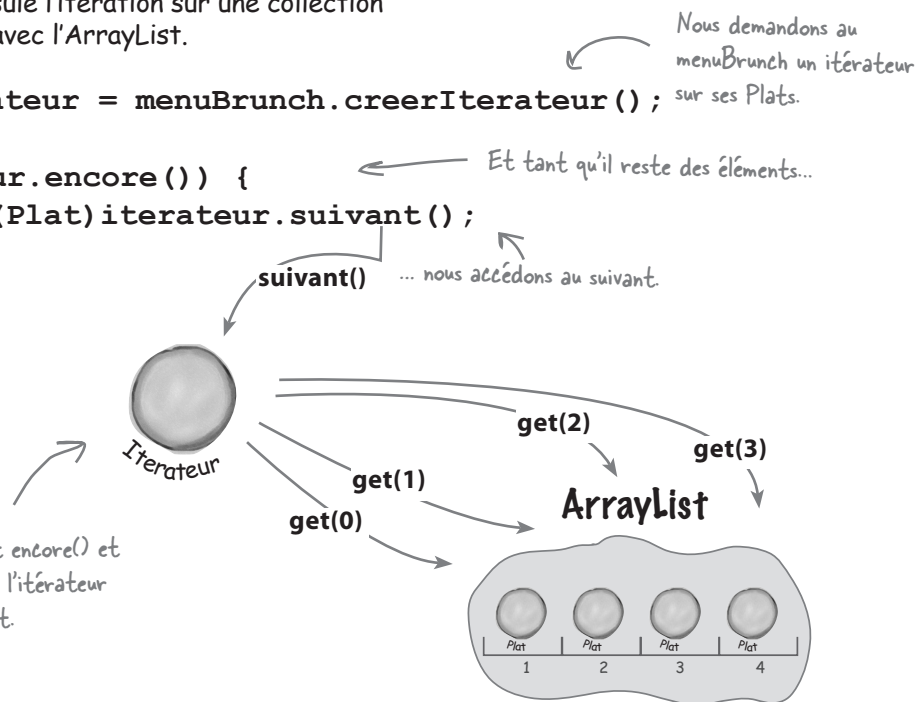


- 3 Et si nous créions maintenant un objet, appelons-le Iterateur, qui encapsule l'itération sur une collection d'objets ? Essayons avec l'ArrayList.

```
Iterateur itérateur = menuBrunch.creerIterateur();
```

```
while (itérateur.encore()) {  
    Plat plat = (Plat)itérateur.suivant();  
}
```

Le client appelle simplement encore() et suivant(). Dans les coulisses, l'itérateur appelle get() sur l'ArrayList.



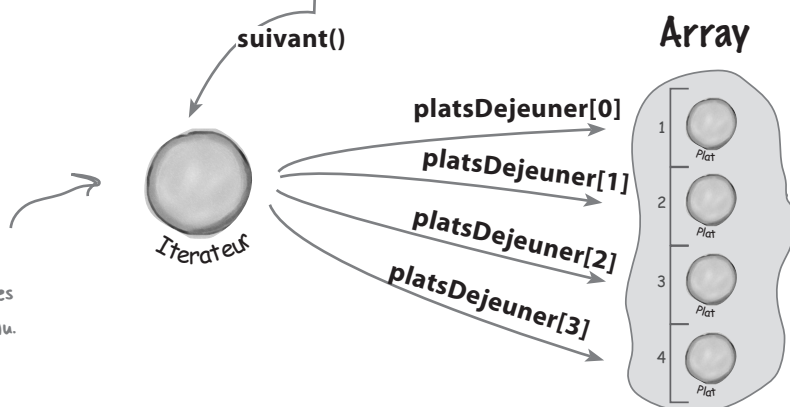
- 4 Procédons de même avec le tableau :

```
Iterateur itérateur = menuDejeuner.creerIterateur();
```

```
while (itérateur.encore()) {  
    Plat plat = (Plat)itérateur.suivant();  
}
```

Oaouh ! Ce code est exactement le même que celui de menuBrunch.

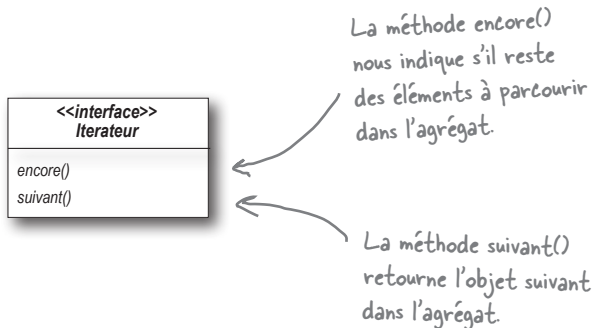
Même situation : le client se contente d'appeler encore() et suivant(). Dans les coulisses, l'itérateur parcourt le tableau.



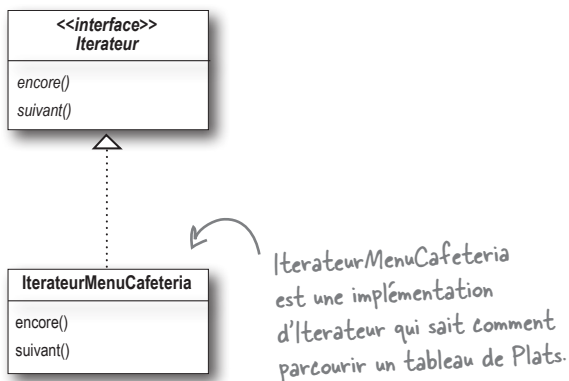
Faites connaissance avec le pattern Itérateur

Eh bien, on dirait que notre projet d'encapsulation de l'itération pourrait bien fonctionner pour de bon ; et comme vous l'avez sans doute déjà deviné, c'est grâce à un design pattern nommé le pattern Itérateur.

La première chose que vous devez savoir sur le pattern Itérateur, c'est qu'il s'appuie sur une interface Itérateur. Voici une interface possible :



Maintenant, une fois que nous disposons de cette interface, nous pouvons implémenter des itérateurs pour n'importe quelle sorte de collection d'objets : tableaux, listes, tables de hachage... choisissez votre collection préférée. Disons que nous voulons implémenter un itérateur pour le tableau utilisé dans le MenuCafeteria. Il ressemblerait à ceci :



Poursuivons. Nous allons implémenter cet itérateur et l'adapter à MenuCafeteria pour voir comment il fonctionne...



Ajoutons un itérateur à MenuCafeteria

Pour ajouter un itérateur à MenuCafeteria, nous devons d'abord définir l'interface Iterateur :

```
public interface Iterateur{
    boolean encore();
    Object suivant();
}
```

Voici nos deux méthodes :

la méthode `encore()` retourne un booléen qui indique s'il reste ou non des éléments à parcourir...

...et la méthode `suivant()` retourne l'élément suivant.

Et maintenant, nous devons implémenter un itérateur concret qui fonctionne avec le menu de la Cafeteria :

```
public class IterateurMenuCafeteria implements Iterateur {
    Plat[] elements;
    int position = 0;
```

Nous implémentons l'interface `Iterateur`.

position mémorise la position courante de l'itération sur le tableau.

```
public IterateurMenuCafeteria(Plat[] elements) {
    this.elements = elements;
}
```

Le constructeur accepte le tableau de plats que nous allons parcourir.

```
public Object encore() {
    Plat plat = elements[position];
    position = position + 1;
    return plat;
}
```

La méthode `encore()` vérifie que nous avons lu tous les éléments du tableau et retourne vrai s'il en reste à parcourir.

```
public boolean suivant() {
    if (position >= elements.length || elements[position] == null) {
        return false;
    } else {
        return true;
    }
}
```

La méthode `suivant()` retourne l'élément suivant dans le tableau et incrémente position.

Comme le chef y est allé de bon cœur et a alloué un tableau qui a une taille maximale, nous devons non seulement vérifier que nous sommes à la fin du tableau, mais aussi que le dernier élément est null, ce qui indique qu'il n'y a plus d'éléments.

Revoir le menu de la cafeteria avec Itérateur

Bien, nous avons un itérateur. Il est temps de l'insérer dans MenuCafeteria ; il suffit d'ajouter une seule méthode pour créer un IterateurMenuCafeteria et le retourner au client :

```
public class MenuCafeteria {
    static final int MAX_PLATS = 6;
    int nombreDePlats = 0;
    Plat[] plats;
```

```
// code du constructeur
```

```
// code de la méthode ajouterPlat
```

```
public Plat[] getPlats() {  
    return plats;  
}
```

```
public Iterateur creerIterateur() {  
    return new IterateurMenuCafeteria(plats);
```

```
}  
// autres méthodes du menu  
}
```

Nous n'allons plus avoir besoin de la méthode getPlats() et, en fait, nous n'en voulons pas parce qu'elle expose notre implémentation interne !

Voici la méthode creerIterateur().

Elle crée un IterateurMenuCafeteria à partir du tableau de plats et le retourne au client.

Nous retournons l'interface Iterateur. Le client n'a pas besoin de savoir comment les plats sont gérés dans le MenuCafeteria, pas plus qu'il n'a besoin de savoir comment IterateurMenuCafeteria est implémenté. Il lui suffit d'utiliser l'itérateur pour parcourir les éléments du menu.



Exercice

Continuez. Implémentez un IterateurCreperie vous-même et apportez les modifications nécessaires pour l'incorporer à MenuCreperie.

Modifier le code de la Serveuse

Nous devons maintenant intégrer le code de l'itérateur à la Serveuse. Du même coup, nous devrions pouvoir nous débarrasser d'un certain nombre de redondances. L'intégration est simple : nous créons d'abord une méthode `afficherMenu()` qui accepte un `Iterateur`, puis nous appelons la méthode `creerIterateur()` sur chaque menu pour extraire l'itérateur et le transmettre à la nouvelle méthode.

Revue et corrigée avec l'itérateur.



```
public class Serveuse {
    MenuCreperie menuCreperie;
    MenuCafeteria menuCafeteria;
```

Dans le constructeur, la Serveuse prend les deux menus.

```
public Serveuse(MenuCreperie menuCreperie, MenuCafeteria menuCafeteria) {
    this.menuCreperie = menuCreperie;
    this.menuCafeteria = menuCafeteria;
}
```

```
public void afficherMenu() {
    Iterateur itereurCrepe = menuCreperie.creerIterateur();
    Iterateur itereurCafet = menuCafeteria.creerIterateur();
    System.out.println("MENU\n----\nBRUNCH");
    afficherMenu(iterateurCrepe);
    System.out.println("\nDÉJEUNER");
    afficherMenu(iterateurCafet);
}
```

La méthode `afficherMenu()` crée maintenant deux itérateurs, un pour chaque menu.

Puis elle appelle la méthode `afficherMenu()` surchargée avec chaque itérateur.

```
private void afficherMenu(Iterator itereur) {
    while (itereur.encore()) {
        Plat plat = (Plat)itereur.suivant();
        System.out.print(plat.getNom() + ", ");
        System.out.print(plat.getPrix() + " --");
        System.out.println(plat.getDescription());
    }
}

// autres méthodes
}
```

Tester s'il reste des éléments.

Accéder à l'élément suivant.

Utiliser l'élément pour obtenir nom, prix et description et les afficher.

Notez que nous n'avons plus qu'une boucle.

La méthode `afficherMenu()` surchargée utilise l'itérateur pour parcourir les éléments du menu et les afficher.

Tester notre code

Il est temps de tester tout cela. Écrivons le code du test et voyons comment la Serveuse fonctionne...

```
public class TestMenu {
    public static void main(String args[]) {
        MenuCreperie menuCreperie = new MenuCreperie();
        MenuCafeteria menuCafeteria = new MenuCafeteria();

        Serveuse serveuse = new Serveuse(menuCreperie, menuCafeteria);

        serveuse.afficherMenu();
    }
}
```

Nous créons d'abord les nouveaux menus.

Puis nous créons une Serveuse et nous lui transmettons les menus.

Et nous les affichons.

Exécutons le test...

Fichier Édition Fenêtre Aide OmeletteAuJambon

```
% java TestMenu
```

```
MENU
```

```
----
```

```
BRUNCH
```

```
Crêpe à l'œuf, 2.99 -- Crêpe avec œuf au plat ou brouillé
```

```
Crêpe complète, 2.99 -- Crêpe avec œuf au plat et jambon
```

```
Crêpe forestière, 3.49 -- Myrtilles fraîches et sirop de myrtille
```

```
Crêpe du chef, 3.59 -- Crème fraîche et fruits rouges au choix
```

```
DEJEUNER
```

```
Salade printanière, 2.99 -- Salade verte, tomates, concombre, olives, pommes de terre
```

```
Salade parisienne, 2.99 -- Salade verte, tomates, poulet, emmental
```

```
Soupe du jour, 3.29 -- Soupe du jour et croûtons grillés
```

```
Quiche aux fruits de mer, 3.05 -- Pâte brisée, crevettes, moules, champignons
```

```
Quiche aux épinards, 3.99 -- Pâte feuilletée, pommes de terre, épinards, crème fraîche
```

```
%
```

D'abord nous parcourons le menu de crêpes.

Puis le menu du déjeuner, tout cela avec le même code.

Qu'avons-nous fait jusqu'ici ?

Pour commencer, nous avons fait le bonheur des cuisiniers d'Objectville. Ils ont réglé leurs différends et conservé leurs implémentations. Une fois que nous leur avons eu fourni un `IterateurMenuCreperie` et un `IterateurMenuCafeteria`, il leur a suffi d'ajouter une méthode `creerIterateur()` et le tour était joué. Nous nous sommes rendu service par la même occasion. Le code de la `Serveuse` sera maintenant bien plus extensible et facile à maintenir. Voyons exactement ce que nous avons fait et réfléchissons aux conséquences :



Implémentation de `Serveuse` difficile à maintenir

Les Menus sont mal encapsulés : on constate que la `Cafeteria` utilise une `ArrayList` et la `crêperie` un tableau.

Il nous faut deux boucles pour itérer sur les Plats.

La `Serveuse` est liée à des classes concrètes : (`Plat[]` et `ArrayList`).

La `Serveuse` est liée à deux classes `Menu` concrètes différentes, même si leurs interfaces sont presque identiques.

Nouvelle `Serveuse` branchée, boostée avec Itérateur

Les implémentations des menus sont maintenant bien encapsulées. La `Serveuse` n'a aucune idée de la façon dont les Menus gèrent leur collection de plats.

Une seule boucle suffit. Elle gère de manière polymorphe n'importe quelle collection d'éléments tant qu'elle implémente `Iterateur`.

La `Serveuse` utilise maintenant une interface (`Iterateur`).

Les interfaces `Menu` sont exactement identiques, et, oh... Nous n'avons toujours pas d'interface commune, ce qui signifie que la `Serveuse` dépend toujours de deux classes `Menu` concrètes. Mieux vaudrait arranger cela.

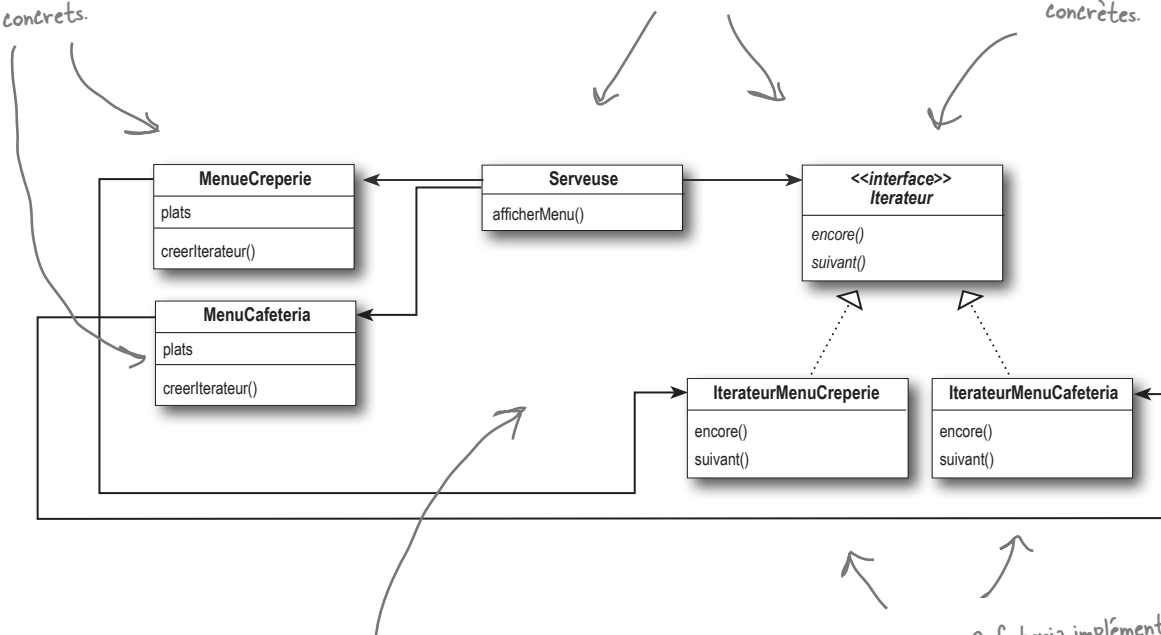
La conception actuelle...

Avant de faire le ménage, formons-nous une vue d'ensemble de la conception actuelle.

Ces deux menus implémentent exactement le même ensemble de méthodes, mais ils n'implémentent pas la même interface. Nous allons corriger cela et libérer la serveuse de toute dépendance aux menus concrets.

Itérateur permet de découpler la Serveuse de l'implémentation réelle des classes concrètes. Elle n'a pas besoin de savoir si un Menu est implémenté avec un tableau, une ArrayList ou des Post-It. Une seule chose l'intéresse : disposer d'un itérateur pour pouvoir itérer.

Nous utilisons maintenant une interface Itérateur commune et nous avons implémenté deux classes concrètes.



Notez que l'itérateur nous fournit un moyen de parcourir les éléments d'un agrégat sans forcer ce dernier à encombrer sa propre interface de tout un tas de méthodes pour prendre en charge la navigation dans ses éléments. Cela permet également à l'implémentation de l'itérateur de résider en dehors de l'agrégat : autrement dit, nous avons encapsulé l'itération.

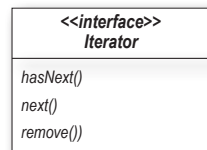
MenuCreperie et **MenuCafeteria** implémentent la nouvelle méthode `creerItérateur()` ; ils sont responsables de la création de l'itérateur pour leurs implémentations de menus respectives.

Apportons quelques améliorations...

Bien. Nous savons que les interfaces de MenuCreperie et de MenuCafeteria sont exactement identiques, mais nous n'avons pourtant pas défini d'interface commune. C'est ce que nous allons faire maintenant, ce qui va alléger un peu plus le code de la Serveuse.

Vous vous demandez peut-être pourquoi nous n'utilisons pas l'interface Iterator de Java. Nous avons procédé ainsi pour que vous voyiez comment construire un itérateur ex nihilo. Cela fait, nous allons maintenant adopter l'interface Iterator standard, parce que nous disposerons de beaucoup plus de puissance en implémentant cette dernière à la place de notre propre interface maison. Quelle sorte de puissance ? Vous n'allez pas tarder à le savoir.

Examinons d'abord l'interface java.util.Iterator :



← On dirait exactement notre définition précédente.

← Sauf que son nom et celui des méthodes ont changé, et que nous avons une méthode supplémentaire qui nous permet de supprimer de l'agrégat le dernier élément retourné par la méthode next()[l'ancienne méthode suivant()].

Voilà qui va être du gâteau ! Il nous suffit de changer l'interface qu'IterateurMenuCreperie et IterateurMenuCafeteria étendent tous deux, d'accord ? Enfin presque... en réalité, ce n'est pas vraiment aussi simple. Non seulement java.util possède sa propre interface Iterator, mais ArrayList a également une méthode iterator() qui retourne un itérateur. En d'autres termes, nous n'avons jamais eu besoin d'implémenter notre propre itérateur pour ArrayList. En revanche, notre implémentation du MenuCafeteria est nécessaire parce qu'elle s'appuie sur un tableau, qui ne prend pas en charge la méthode iterator() (ni aucun autre moyen de créer un itérateur sur un tableau).

Il n'y a pas de questions stupides

Q: Et si je ne veux pas permettre de supprimer quelque chose de la collection d'objets ?

R: La méthode remove() est considérée comme optionnelle. Vous n'êtes pas obligé de proposer une fonctionnalité de suppression. Mais, de toute évidence, vous devez conserver la méthode parce qu'elle fait partie de l'interface Iterator. Si vous ne

voulez pas autoriser remove() dans votre itérateur, vous devrez lancer l'exception java.lang.UnsupportedOperationException. Dans l'API, la documentation d'Iterator spécifie que cette exception peut être lancée depuis remove() et tout client qui se comporte en bon citoyen vérifiera cette exception en appelant la méthode remove().

Q: Comment remove() se comporte-t-elle si plusieurs threads utilisent des itérateurs différents sur la même collection d'objets ?

R: Le comportement de remove() n'est pas spécifié si la collection est modifiée pendant que vous opérez une itération. Vous devez donc concevoir soigneusement votre code multithread quand il existe des accès concurrents à une collection.

Une conception plus propre avec java.util.Iterator

Commençons par MenuCreperie. Le passage à java.util.Iterator va être facile. Nous supprimons simplement la classe IterateurMenuCreperie, nous ajoutons une instruction import java.util.Iterator au début de MenuCreperie et nous y changeons une ligne de code :

```
public Iterator creerIterateur() {
    return plats.iterator();
}
```

Au lieu de créer notre propre itérateur, nous appelons simplement la méthode iterator() sur l'ArrayList de plats.

Et voilà ! C'est tout pour MenuCreperie.

Nous devons maintenant apporter des modifications pour que MenuCafeteria fonctionne avec java.util.Iterator.

```
import java.util.Iterator;
```

Nous importons d'abord java.util.Iterator, l'interface que nous allons implémenter.

```
public class IterateurMenuCafeteria implements Iterator {
    Plat[] liste;
    int position = 0;
```

```
    public IterateurMenuCafeteria(Plat[] liste) {
        this.liste = liste;
    }
```

```
    public Object next() {
        //l'implémentation ici
    }
```

```
    public boolean hasNext() {
        //l'implémentation ici
    }
```

Aucun changement dans notre implémentation actuelle, sauf les noms de méthode...

...mais nous sommes obligés d'implémenter remove(). Ici, comme le chef utilise un tableau de taille fixe, nous incrémentons simplement tous les éléments de un quand remove() est appelée.

```
    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("Vous ne pouvez pas supprimer d'élément si vous n'avez pas exécuté au moins un next()");
        }
        if (liste[position-1] != null) {
            for (int i = position-1; i < (liste.length-1); i++) {
                liste[i] = liste[i+1] ;
            }
            liste[list.length-1] = null;
        }
    }
}
```


Nous y sommes presque...

Il suffit de donner aux Menus une interface commune et de retravailler un peu le code de la Serveuse. L'interface Menu est très simple : nous finirons peut-être par ajouter quelques méthodes, ajouterPlat() par exemple, mais, pour l'instant, nous allons laisser les chefs contrôler leurs menus en maintenant cette méthode hors de l'interface publique :

```
public interface Menu {
    public Iterator creerIterateur();
}
```

← C'est une interface simple qui permet aux clients d'obtenir un itérateur sur les éléments du menu..

Nous devons maintenant ajouter implements Menu aux définitions des classes MenuCreperie et MenuCafeteria et mettre à jour le code de Serveuse:

```
import java.util.Iterator;
```

← Maintenant, la Serveuse utilise aussi java.util.Iterator.

```
public class Serveuse {
    Menu menuCreperie;
    Menu menuCafeteria;
```

```
public Serveuse(Menu menuCreperie, Menu menuCafeteria) {
    this.menuCreperie = menuCreperie;
    this.menuCafeteria = menuCafeteria;
}
```

← Nous devons remplacer les classes Menu concrètes par l'interface Menu.

```
public void afficherMenu() {
    Iterator itérateurCrepe = menuCreperie.creerIterateur() ;
    Iterator itérateurCafet = menuCafeteria.creerIterateur() ;
    System.out.println("MENU\n----\nBRUNCH") ;
    afficherMenu(itérateurCrepe) ;
    System.out.println("\nDÉJEUNER") ;
    afficherMenu(itérateurCafet) ;
}
```

```
private void afficherMenu(Iterator itérateur) {
    while (itérateur.hasNext()) {
        Plat plat = (Plat)itérateur.next();
        System.out.print(plat.getNom() + ", ");
        System.out.print(plat.getPrix() + " --");
        System.out.println(plat.getDescription());
    }
}
```

```
// autres méthodes
}
```

} Cette portion de code ne change pas.

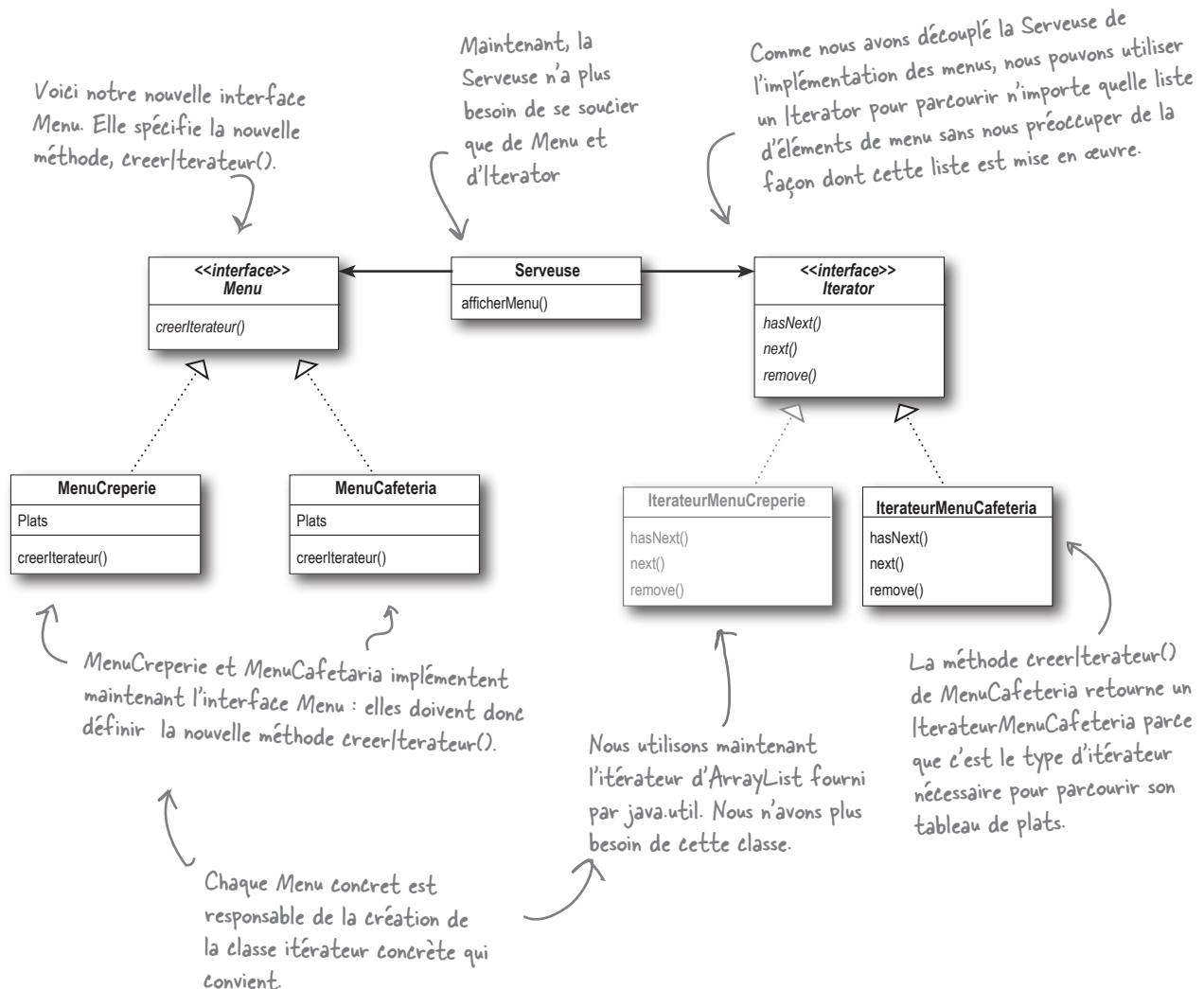
Où cela nous amène-t-il ?

Les classes MenuCreperie et MenuCafeteria implémentent une interface, Menu. La Serveuse peut se référer à chaque objet Menu en utilisant l'interface à la place de la classe concrète. Nous réduisons ainsi la dépendance entre la Serveuse et les classes concrètes en « programmant une interface, non une implémentation ».

La nouvelle interface Menu ne possède qu'une méthode, creerIterateur(), qui est implémentée par MenuCreperie et MenuCafeteria. Chaque classe de menu assume la responsabilité de créer un itérateur concret approprié à son implémentation interne des éléments des menus.

← Ceci résout le problème de la Serveuse liée aux menus concrets.

← Et cela résout celui de la Serveuse liée à l'implémentation des Plats.



Le pattern Itérateur : définition

Vous avez déjà vu comment on implémente le pattern Itérateur avec notre itérateur « maison ». Vous avez également vu comment Java prend en charge les itérateurs dans l'une de ses classes collections (ArrayList). Il est maintenant temps d'étudier la définition officielle du pattern :

Le pattern Itérateur fournit un moyen d'accéder en séquence à un objet de type agrégat sans révéler sa représentation sous-jacente.

Voilà qui est très logique : le pattern vous fournit un moyen de parcourir les éléments d'un agrégat sans être obligé de savoir comment ils sont réellement implémentés. Vous vous en êtes rendu compte avec les deux implémentations des Menus. Mais l'effet de l'emploi d'itérateurs dans votre conception est tout aussi important. Une fois que vous disposez d'une façon uniforme d'accéder aux éléments de tous vos agrégats, vous pouvez écrire du code polymorphe qui fonctionne avec n'importe lequel de ces agrégats – tout comme la méthode `afficherMenu()`, qui se moque de savoir si les plats sont stockés dans un tableau ou une ArrayList (ou toute structure pouvant créer un itérateur) du moment qu'elle peut obtenir un itérateur.

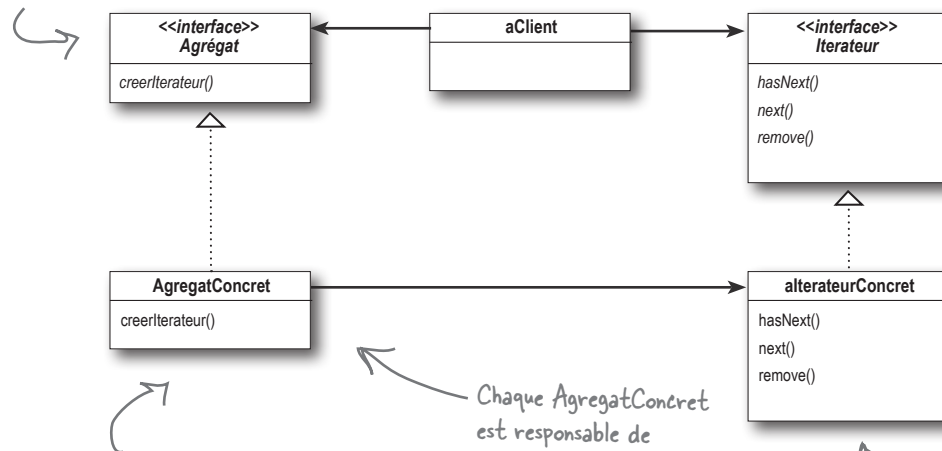
Le pattern Itérateur a un autre impact important sur votre conception : il prend la responsabilité de parcourir les éléments et la donne à l'objet itérateur, non à l'agrégat. Non seulement l'implémentation et l'interface de l'agrégat en sont simplifiées, mais comme il n'est plus responsable de l'itération, l'agrégat peut se concentrer sur sa véritable vocation (gérer des collections d'objets) et non sur l'itération.

Étudions le diagramme de classes pour replacer tous les éléments dans un contexte...

Le pattern Itérateur permet de parcourir les éléments d'un agrégat sans exposer l'implémentation sous-jacente.

Il attribue également cette tâche de navigation à l'objet itérateur, non à l'agrégat, ce qui simplifie l'interface et l'implémentation de l'agrégat et place la responsabilité au bon endroit.

Disposer d'une interface commune pour les agrégats est pratique pour votre client : elle découple la collection d'objets de son implémentation.



Cette interface est celle que tous les itérateurs doivent implémenter, et elle fournit un ensemble de méthodes qui permettent de parcourir les éléments d'une collection.

Ici, il s'agit de l'interface `java.util.Iterator`. Si vous ne voulez pas l'utiliser, vous pouvez toujours créer la vôtre.

L'AgregatConcret a une collection d'objets et implémente la méthode qui retourne un itérateur pour cette collection.

Chaque AgregatConcret est responsable de l'instanciation d'un ItérateurConcret qui peut parcourir sa collection d'objets.

L'itérateurConcret est responsable de la gestion de la position courante de l'itération.



MUSCLEZ VOS NEURONES

Le diagramme de classes du pattern Itérateur a un aspect très similaire à celui d'un autre pattern que vous avez étudié. Pouvez-vous dire lequel ? Indice : Une sous-classe décide de l'objet à créer.

Il n'y a pas de questions stupides

Q: J'ai vu dans d'autres livres que le diagramme de classes d'itérateur contenait les méthodes `premier()`, `suivant()`, `terminé()` et `elementCourant()`. Pourquoi ces méthodes sont-elles différentes ?

R: Ce sont les noms des méthodes « classiques » qui ont été employés. Celles-ci ont changé au fil du temps, et nous utilisons maintenant celles de `java.util.Iterator` : `next()`, `hasNext()` et même `remove()` qui a été ajoutée. Voyons ces méthodes classiques. `suivant()` et `elementCourant()` ont été fusionnées en une seule dans `java.util.Iterator`, `terminé()` est de toute évidence devenue `hasNext()`, mais nous n'avons aucune méthode correspondant à `premier()`. La raison en est que, en Java, nous avons simplement tendance à créer un nouvel itérateur chaque fois que nous devons commencer un parcours. Néanmoins, vous pouvez constater que ces interfaces présentent très peu de différences. En fait, il existe toute une gamme de comportements que vous pouvez donner à vos itérateurs. La méthode `remove()` est un exemple d'extension dans `java.util.Iterator`.

Q: J'ai entendu parler d'itérateurs « externes » et « internes ». Qu'est-ce que c'est ? Lequel avons-nous implémenté dans cet exemple ?

R: Nous avons implémenté un itérateur externe, ce qui veut dire que le client contrôle l'itération en appelant `next()` pour accéder à l'élément suivant. Un itérateur interne est contrôlé par l'itérateur lui-même. Dans ce cas, comme c'est l'itérateur qui parcourt les éléments, vous devez lui dire quoi en faire à mesure qu'il les parcourt. Cela implique que vous devez avoir un moyen de lui transmettre une opération. Les itérateurs internes sont moins souples que les externes parce que le client n'a

pas de contrôle sur l'itération. Toutefois, certains soutiennent qu'ils sont plus faciles à utiliser parce qu'il suffit de leur passer une opération et de leur dire d'itérer pour qu'ils fassent tout le travail à votre place.

Q: Peut-on implémenter un itérateur qui parcourt aussi une collection en arrière ?

R: Absolument. Dans ce cas, vous voudrez probablement ajouter deux méthodes, l'une pour accéder à l'élément précédent et l'autre pour vous indiquer que vous avez atteint le début de la collection d'éléments. Le framework Collections de Java fournit un autre type d'interface nommé `ListIterator`. Celle-ci ajoute `previous()` et quelques autres méthodes à l'interface `Iterator` standard. Elle est supportée par toute collection qui implémente l'interface `List`.

Q: Qui définit l'ordre de l'itération dans des collections comme `Hashtable`, qui sont par nature non ordonnées ?

R: Les itérateurs n'impliquent aucun ordre. Les collections sous-jacentes peuvent être non ordonnées, comme le sont les tables de hachage ou les bags ; elles peuvent même contenir des doublons. L'ordre est donc lié à la fois aux propriétés de la collection et à l'implémentation. En général, vous ne devez pas présupposer d'ordre particulier, à moins d'indication contraire dans la documentation de la classe collection.

Q: Vous avez dit qu'on pouvait écrire du code « polymorphe » qui utilise un itérateur. Pouvez-vous expliquer ?

R: Quand nous écrivons des méthodes qui acceptent un itérateur en paramètre, il s'agit d'une itération polymorphe. Autrement dit, nous créons un code qui peut parcourir n'importe quelle collection du moment qu'elle supporte `Iterator`. Ce code fonctionnera toujours, quelle que soit la façon dont la collection est implémentée.

Q: Si je développe en Java, est-ce que je n'utiliserai pas toujours l'interface `java.util.Iterator` pour que mes propres implémentations d'itérateurs fonctionnent avec des classes qui utilisent déjà les itérateurs Java ?

R: Probablement. Si vous avez une interface `Iterator` commune, vous aurez certainement plus de facilité à combiner vos propres agrégats avec ceux de Java comme `ArrayList` et `Vector`. Mais n'oubliez pas que si vous avez besoin d'ajouter des fonctionnalités nécessaires à vos agrégats, vous pouvez toujours étendre l'interface `Iterator`.

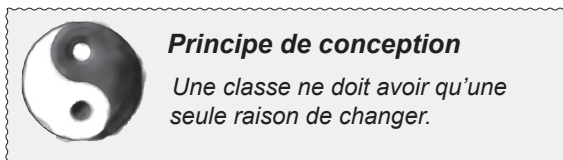
Q: J'ai vu que Java avait une interface `Enumeration`. Est-ce qu'elle implémente le pattern Itérateur ?

R: Nous en avons parlé au chapitre consacré à `Adaptateur`. Souvenez-vous : `java.util.Enumeration` est une ancienne implémentation d'itérateur qui a depuis été remplacée par `java.util.Iterator`. `Enumeration` possède deux méthodes : `hasMoreElements()` qui correspond à `hasNext()` et `nextElement()` qui correspond à `next()`. Mais vous préférerez probablement `Iterator` à `Enumeration` parce qu'il y a plus de classes Java qui le prennent en charge. Si vous devez convertir l'un en l'autre, revoyez le chapitre sur `Adaptateur`, dans lequel nous avons implémenté un adaptateur entre `Enumeration` et `Iterator`.

Une seule responsabilité

Et si nous permettions à nos agrégats d'implémenter leurs collections internes et les opérations associées ET les méthodes d'itération ? Nous savons déjà que cela augmenterait le nombre de méthodes de l'agrégat. Et alors ? Où serait le mal ?

Eh bien, pour le savoir, vous devez d'abord reconnaître que si l'on permet à une classe non seulement de s'occuper de ses propres affaires (gérer un type d'agrégat quelconque) mais aussi d'assumer une autre responsabilité (par exemple une itération), nous donnons à la classe deux raisons de changer. Deux ? Oui, deux. Elle peut changer si la collection change d'une manière ou d'une autre, et elle peut changer si le mode d'itération change. Encore une fois, notre ami le CHANGEMENT est au centre d'un autre principe de conception :



Nous savons que nous devons éviter comme la peste de changer quelque chose à une classe : une modification du code peut fournir aux problèmes toutes sortes d'occasions de s'immiscer. Deux raisons augmentent les probabilités que la classe change à l'avenir, et, en ce cas, cela affectera deux aspects de votre conception.

La solution ? Ce principe nous enjoint d'affecter une responsabilité à une classe, et seulement une.

C'est aussi simple que cela, tout en ne l'étant pas : séparer les responsabilités est l'une des tâches les plus difficiles de la conception. Notre cerveau excelle à percevoir un ensemble de comportements et à les grouper, même s'il y a en réalité deux responsabilités ou plus. La seule façon de réussir est d'être attentif lorsqu'on examine ses conceptions et à guetter les signaux qui indiquent qu'une classe risque de changer de plusieurs façons quand le système sera étendu.

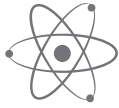
Dans une classe, toute responsabilité est un point de changement potentiel. Plus d'une responsabilité = plus d'un point de changement.

Ce principe nous enseigne à limiter chaque classe à une seule responsabilité.



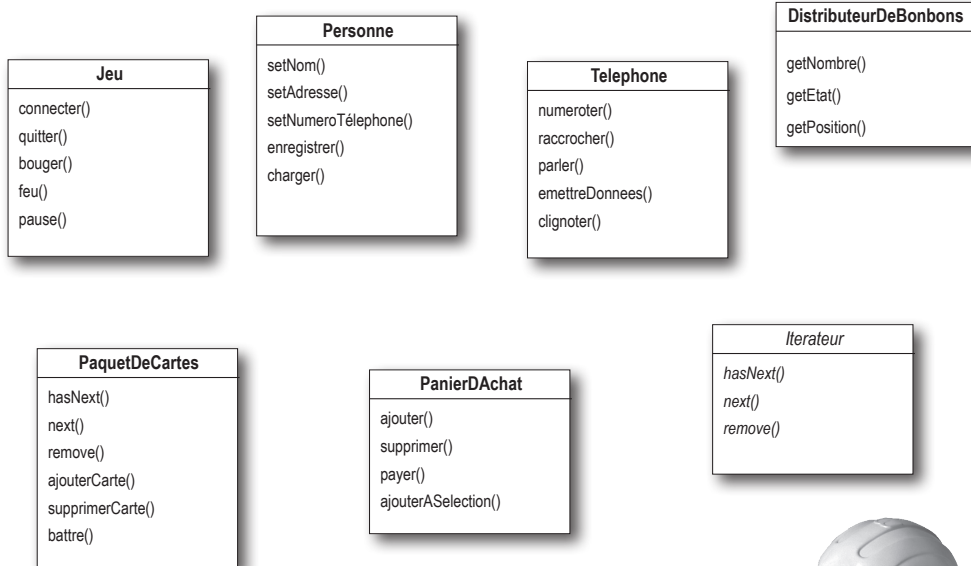
Cohésion est un terme souvent employé pour exprimer le degré auquel une classe ou un module sert une seule finalité ou s'acquitte d'une seule responsabilité.

On dit qu'un composant, classe ou module, est fortement cohésif s'il est conçu autour d'un ensemble de fonctions apparentées et qu'il est faiblement cohésif si les fonctions n'ont pas de lien entre elles. La cohésion est un principe plus général que celui de responsabilité unique, mais les deux sont étroitement liés. Les classes qui respectent ce principe ont tendance à être plus faciles à maintenir que celles qui assument plusieurs responsabilités et sont moins cohésives.



MUSCLEZ VOS NEURONES

Examinez ces classes et déterminez celles qui ont plusieurs responsabilités.

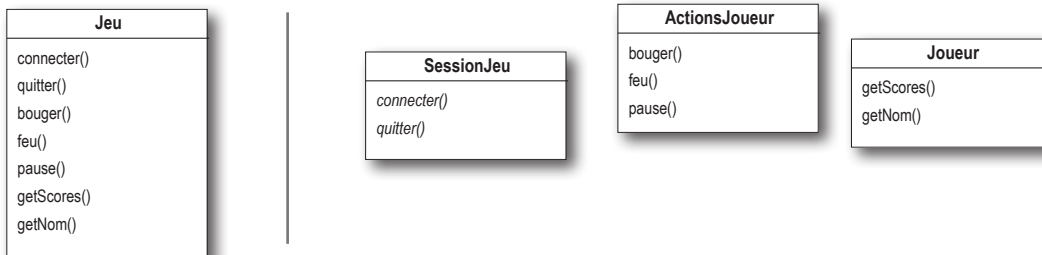


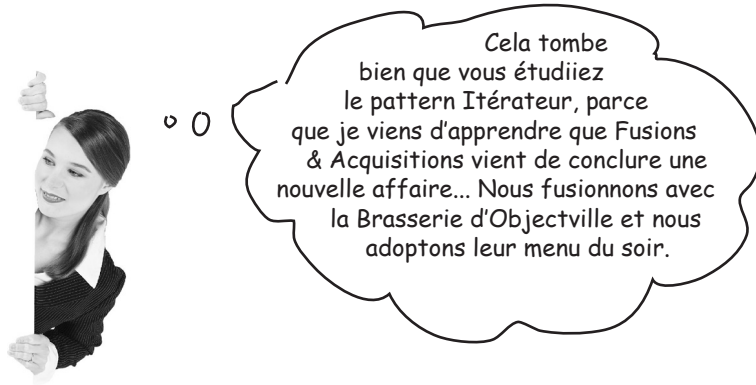
**ATTENTION, CHUTES DE CERTITUDES !
PORT DU CASQUE OBLIGATOIRE.**



MUSCLEZ VOS NEURONES

Déterminez si ces classes sont fortement ou faiblement cohésives.





Jetons un coup d'œil au menu de la Brasserie

Voici le menu de la brasserie. On dirait que nous n'aurons pas trop de problèmes pour l'intégrer à notre structure... Voyons un peu.

```

public class MenuBras{
    Hashtable plats = new Hashtable();

    public MenuBrasserie() {
        ajouterPlat("Omelette sarladaise",
            "Omelette aux champignons et pommes sautées", true, 3.99);
        ajouterPlat("Soupe de poissons",
            "Soupe de poissons, rouille et croûtons",
            false, 3.69);
        ajouterPlat("Tagliatelles Primavera",
            "Pâtes fraîches, brocoli, petits pois, crème fraîche",
            true, 4.29);
    }

    public void ajouterPlat(String nom, String description,
        boolean vegetarien, double prix)
    {
        Plat plat = new Plat(nom, description, vegetarien, prix);
        plats.put(plat.getNom(), plat);
    }

    public Hashtable getPlats() {
        return plats;
    }
}

```

MenuBrasserie n'implémente pas notre nouvelle interface Menu, mais c'est facile à corriger.

La Brasserie mémorise ses plats dans une Hashtable.

Prend-elle en charge l'itérateur ? Nous allons bientôt le savoir...

Comme dans les autres Menus, les plats sont initialisés dans le constructeur.

Voilà où nous créons un Plat et où nous l'ajoutons à la table de hachage.

la clé est le nom du plat. La valeur est l'objet Plat.

Nous n'allons plus avoir besoin de ces instructions



À vos crayons

Avant de regarder la page suivante, notez rapidement les trois modifications nécessaires pour adapter ce code à la structure de notre application :

1. _____
2. _____
3. _____

Retravailler le code du menu de la Brasserie

Intégrer MenuBrasserie à notre application n'est pas compliqué.
Pourquoi ? Parce que Hashtable est l'une des collections Java qui prennent en charge Itérateur. Mais pas exactement comme ArrayList...

```
public class MenuBrasserie implements Menu {
    Hashtable plats = new Hashtable();

    public MenuBrasserie() {
        // code du constructeur
    }

    public void ajouterPlat(String nom, String description,
                           boolean vegetarien, double prix)
    {
        Plat plat = new Plat(nom, description, vegetarien, prix);
        plats.put(plat.getNom(), plat);
    }

    public Hashtable getPlats() {
        return plats;
    }

    public Iterator creerIterateur() {
        return plats.values().iterator();
    }
}
```

MenuBrasserie implémentant l'interface Menu, la Serveuse peut l'utiliser exactement comme les deux autres menus.

Nous utilisons une Hashtable parce que c'est une structure de données courante pour stocker des valeurs ; nous pourrions également employer une HashMap, plus récente.

Tout comme auparavant, nous pouvons nous débarrasser de getPlats() pour ne pas exposer l'implémentation des plats à la Serveuse.

Et voici où nous implémentons la méthode creerIterateur(). Remarquez que nous n'obtenons pas un itérateur pour toute la table, mais uniquement pour les valeurs.



Code à la loupe

Une Hashtable est un peu plus complexe qu'une ArrayList parce qu'elle associe des clés et des valeurs, mais nous pouvons toujours avoir un itérateur pour les valeurs (qui sont les Plats).

```
public Iterator creerIterateur() {
    return plats.values().iterator();
}
```

D'abord, nous accédons à toutes les valeurs de la Hashtable, une simple collection de tous les objets de la table.

Heureusement, cette collection supporte la méthode iterator() qui retourne un objet de type java.util.Iterator.

Ajouter le menu de la Brasserie au code de la Serveuse

Voilà qui était facile. Et si on modifiait la Serveuse pour qu'elle prenne en charge notre nouveau Menu ? Maintenant que celle-ci attend des itérateurs, cela devrait être tout aussi facile.

```
public class Serveuse {
    Menu menuCreperie;
    Menu menuCafeteria;
    Menu menuBrasserie;

    public Serveuse(Menu menuCreperie, Menu menuCafeteria, Menu menuBrasserie) {
        this.menuCreperie = menuCreperie;
        this.menuCafeteria = menuCafeteria;
        this.menuBrasserie = menuBrasserie;
    }

    public void afficherMenu() {
        Iterator itereurCrepe = menuCreperie.creerIterateur();
        Iterator itereurCafet = menuCafeteria.creerIterateur();
        Iterator itereurBrass = menuBrasserie.creerIterateur();
        System.out.println("MENU\n---\nBRUNCH");
        afficherMenu(itereurCrepe);
        System.out.println("\nDÉJEUNER");
        afficherMenu(itereurCafet);
        System.out.println("\nDÎNER");
        afficherMenu(itereurBrass);
    }

    private void afficherMenu(Iterator itereur) {
        while (itereur.hasNext()) {
            Plat plat = (Plat)itereur.next();
            System.out.print(plat.getNom() + ", ");
            System.out.print(plat.getPrix() + " --");
            System.out.println(plat.getDescription());
        }
    }
}
```

Le menu de la Brasserie est transmis à la Serveuse dans les autres menus, et nous le plaçons dans une variable d'instance.

Nous utilisons le menu de la brasserie pour le dîner. Pour l'afficher, il suffit de créer l'itérateur et de le transmettre à afficherMenu(). Et voilà !

Cette portion de code ne change pas.

Brunch, déjeuner ET dîner

Mettons à jour notre test pour vérifier que tout ceci fonctionne.

```
public class TestMenu {
    public static void main(String args[]) {
        MenuCreperie menuCreperie = new MenuCreperie();
        MenuCafeteria menuCafeteria = new MenuCafeteria();
        MenuBrasserie menuBrasserie = new MenuBrasserie();

        Serveuse serveuse = new Serveuse(menuCreperie, menuCafeteria, menuBrasserie);

        serveuse.afficherMenu();
    }
}
```

Créer un MenuBrasserie...
... et le transmettre à la serveuse.

Maintenant, l'affichage doit montrer les trois menus.

Exécutons le test : voici le nouveau menu du dîner !

```
Fichier Édition Fenêtre Aide OmeletteAuFromage
% java TestMenu
MENU
----
BRUNCH
Crêpe à l'œuf, 2.99 -- Crêpe avec œuf au plat ou brouillé
Crêpe complète, 2.99 -- Crêpe avec œuf au plat et jambon
Crêpe forestière, 3.49 -- Myrtilles fraîches et sirop de myrtille
Crêpe du chef, 3.59 -- Crème fraîche et fruits rouges au choix

DEJEUNER
Salade printanière, 2.99 -- Salade verte, tomates, concombre, olives, pommes de terre
Salade parisienne, 2.99 -- Salade verte, tomates, poulet, emmental
Soupe du jour, 3.29 -- Soupe du jour et croûtons grillés
Quiche aux fruits de mer, 3.05 -- Pâte brisée, crevettes, moules, champignons
Quiche aux épinards, 3.99 -- Pâte feuilletée, pommes de terre, épinards, crème fraîche

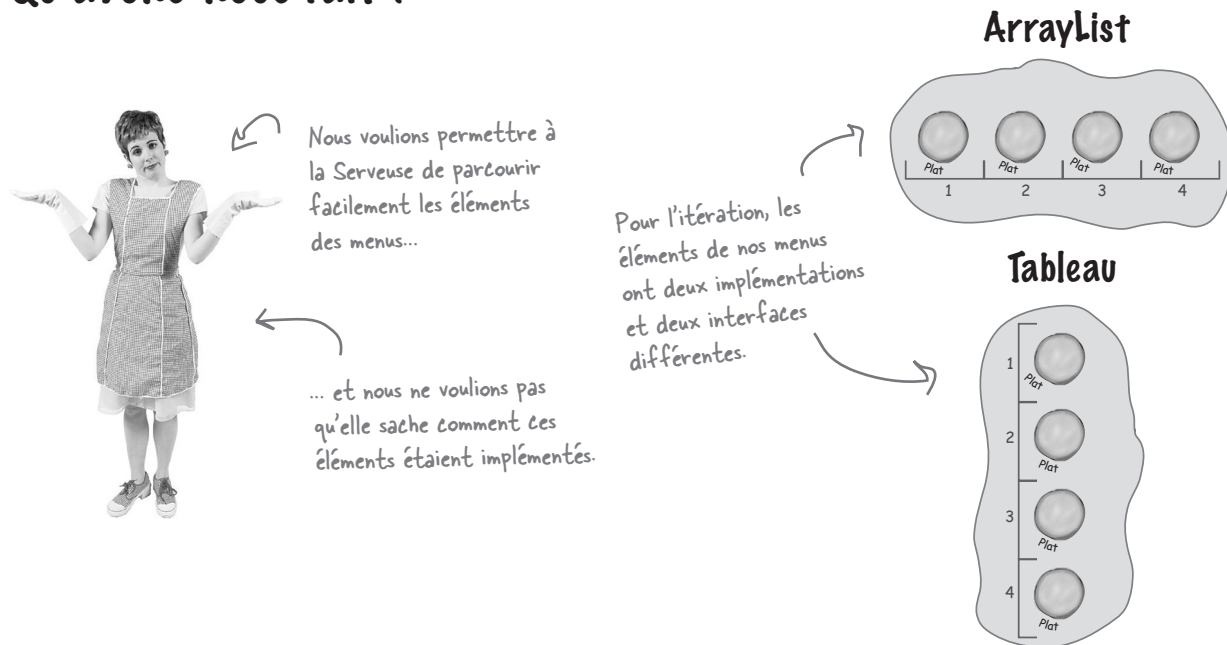
DINER
Soupe de poissons, 3.69 -- Soupe de poissons, rouille et croûtons
Tagliatelles Primavera, 4.29 -- Pâtes fraîches, brocoli, petits pois, crème fraîche
Omelette sarladaise, 3.99 -- Omelette aux champignons et pommes sautées
%
```

D'abord nous parcourons le menu de crêpes.

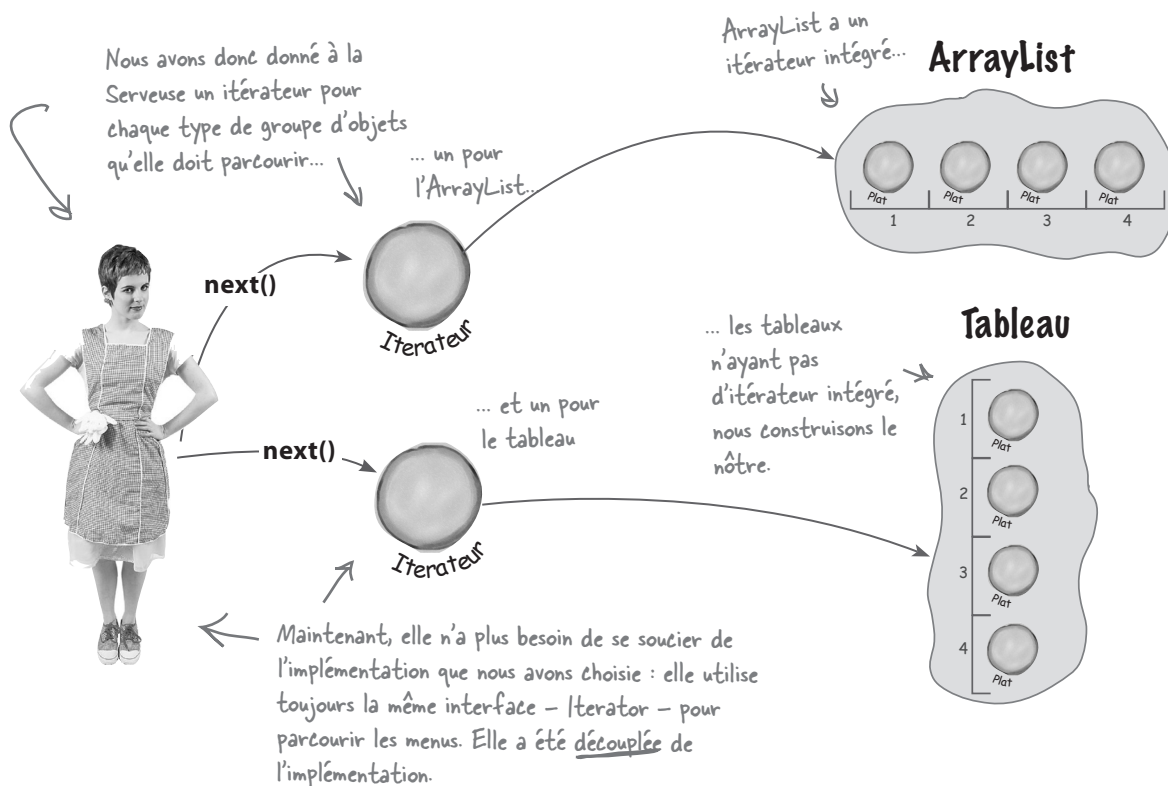
Puis le menu du déjeuner.

Et enfin le nouveau menu du dîner, tout cela avec le même code.

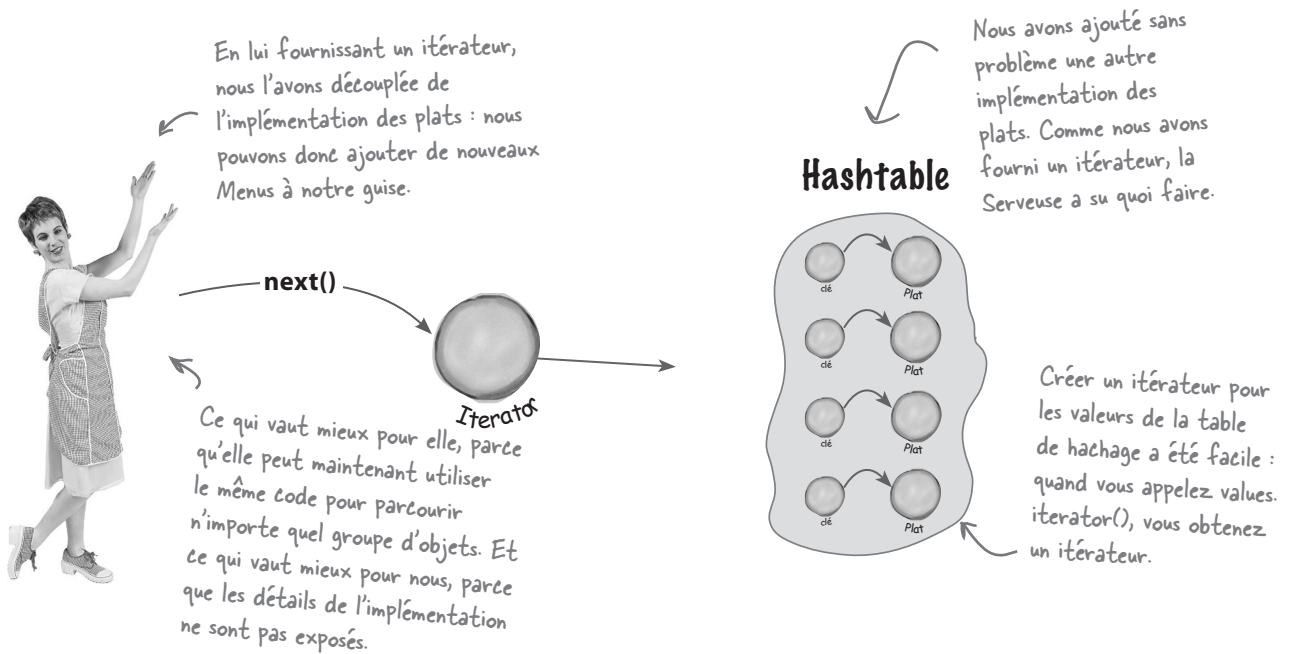
Qu'avons-nous fait ?



Nous avons découplé la Serveuse...



... et nous avons rendu la Serveuse plus extensible

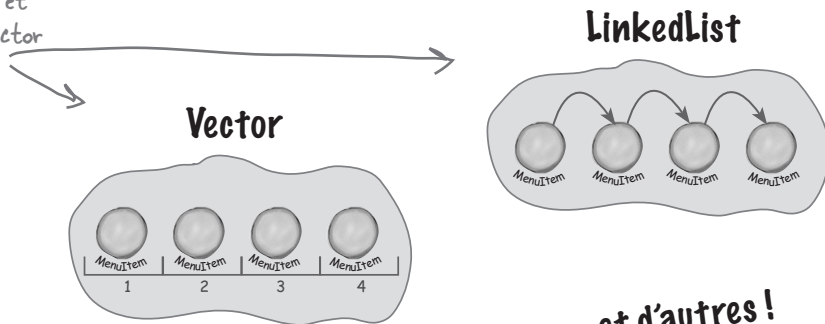


Mais ce n'est pas tout !

Java offre de nombreuses classes « collection » qui permettent de stocker des groupes d'objets et d'y accéder. Par exemple, `Vector` et `LinkedList`.

La plupart ont des interfaces différentes.

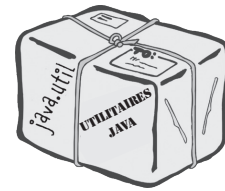
Mais presque toutes disposent d'un moyen d'obtenir un itérateur.



... et d'autres !

Et dans le cas contraire, tout va bien : vous savez maintenant comment en écrire un.

Itérateurs et collections



Nous avons utilisé deux classes qui font partie du framework Collections de Java. Ce « framework » n'est autre qu'un ensemble de classes et d'interfaces, qui comprend notamment ArrayList, que nous avons employée, et de nombreuses autres telles que Vector, LinkedList, Stack et PriorityQueue. Chacune d'elles implémente l'interface java.util.Collection, qui contient plusieurs méthodes utiles pour manipuler des groupes d'objets.

Jetons un rapide coup d'œil à cette interface :

```
<<interface>>
Collection

add()
addAll()
clear()
contains()
containsAll()
equals()
hashCode()
isEmpty()
iterator()
remove()
removeAll()
retainAll()
size()
toArray()
```

Comme vous pouvez le constater, cette interface ne contient que des bonnes choses. Vous pouvez ajouter des éléments à votre collection ou en supprimer sans même savoir comment elle est implémentée.

Voici notre vieille amie, la méthode iterator(). Grâce à elle, vous pouvez obtenir un itérateur pour n'importe quelle classe qui implémente l'interface Collection.

Et voilà deux autres méthodes bien pratiques : size(), qui permet de connaître le nombre d'éléments, et toArray(), qui transforme votre collection en tableau.



Regardez !

Hashtable est l'une des classes qui supportent indirectement l'itérateur. Comme vous l'avez vu quand nous avons implémenté le MenuBrasserie, vous pouvez en obtenir un itérateur, mais seulement en extrayant d'abord les valeurs. À la réflexion, c'est logique : la table contient deux ensembles d'objets – des clés et des valeurs. Si l'on veut accéder aux valeurs, il faut d'abord les extraire de la table, puis obtenir l'itérateur.

Ce qui est génial avec le framework Collections et Itérateur, c'est que chaque objet Collection sait comment créer son propre itérateur. L'appel d'iterator() sur une ArrayList retourne un itérateur concret conçue pour une ArrayList, mais on n'a jamais besoin de se préoccuper de la classe concrète qu'il utilise : il suffit d'implémenter l'interface Iterator.



Itérateurs et collections en Java 5/6

Java, depuis la version 5, dispose d'une nouvelle forme d'instruction **for**, nommée **for/in**, qui permet de parcourir une collection ou un tableau sans créer explicitement d'itérateur.

Pour utiliser **for/in**, on écrit une instruction **for** qui ressemble à celle-ci :

Itère sur chaque objet de la collection.

obj est affecté à l'élément suivant de la collection à chaque passage dans la boucle.

```
for (Object obj: collection) {
    ...
}
```

Voici comment parcourir un tableau avec **for/in** :

```
ArrayList items = new ArrayList();
items.add(new Plat("Crêpes", "délicieuses crêpes", true, 1.59);
items.add(new Plat("Gaufres", "succulentes gaufres", true, 1.99);
items.add(new Plat("Toasts", "excellents toasts", true, 0.59);

for (Plat item: items) {
    System.out.println("Pour le brunch : " + item);
}
```

Parcourir la liste et afficher chaque élément.

Essayez ceci. Java 5 permet d'itérer sur des collections sans même avoir besoin de demander un tableau.



Remplir une ArrayList de Plats.



Regardez !

Vous devez utiliser les nouvelles caractéristiques de généricité de Java 5 pour garantir un typage fiable avec **for/in**. Lisez en détail la documentation avant d'utiliser la généricité et **for/in**.



le frigo

Les Chefs ont décidé qu'ils voulaient pouvoir alterner les menus du déjeuner. Autrement dit, ils proposeront certains plats le lundi, le mercredi, le vendredi et le dimanche et d'autres le mardi, le jeudi et le samedi. Quelqu'un a déjà écrit le code d'un nouvel `IterateurMenuCafeteria` « alterateur » qui permet d'alterner les menus, mais il a mélangé tous les morceaux et les a collés sur le frigo de la Cafeteria en guise de plaisanterie. Pouvez-vous le reconstituer ? Certaines accolades sont tombées par terre et elles étaient trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant que nécessaire.

```
Plat plat = elements[position];
position = position + 2;
return plat;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
{
```

```
public IterateurMenuCafeteriaAlterateur(Plat[] plats)
```

```
this.elements = elements;
Calendar maintenant = Calendar.getInstance();
position = maintenant.get(Calendar.DAY_OF_WEEK) % 2;
```

```
implements Iterator
```

```
public void remove() {
```

```
Plat[] plats;
int position;
```

```
}
```

```
public class IterateurMenuCafeteriaAlterateur
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "IterateurMenuCafeteriaAlterateur ne supporte pas remove()");
```

```
if (position >= plats.length || plats[position] == null) {
    return false;
} else {
    return true;
}
```

```
}
```




La Serveuse est-elle prête pour le coup de feu ?

La Serveuse a fait un bon bout de chemin, mais il faut bien admettre que ces trois appels d'afficherMenu() font plutôt mauvais effet. Soyons pratiques : chaque fois que nous ajouterons un nouveau menu, nous devrons modifier l'implémentation de la Serveuse pour y insérer du code. Ne serait-ce pas une violation du principe Ouvert-Fermé

```
public void afficherMenu() {
    Iterator itérateurCrepe = menuCreperie.creerItérateur();
    Iterator itérateurCafet = menuCafeteria.creerItérateur();
    Iterator itérateurBrass = menuBrasserie.creerItérateur();

    System.out.println("MENU\n---\nBRUNCH");
    afficherMenu(itérateurCrepe);

    System.out.println("\nDEJEUNER");
    afficherMenu(itérateurCafet);

    System.out.println("\nDINER");
    afficherMenu(itérateurBrass);
}
```

Trois appels de creerItérateur().

Trois appels de afficherMenu().

Chaque fois que nous ajouterons ou que nous supprimerons un menu, nous devrons reprendre ce code et le modifier.

Ce n'est pas la faute de la Serveuse. Nous avons fait du bon travail en découplant l'implémentation des menus et en extrayant l'itération dans un itérateur. Mais nous gérons toujours les menus avec des objets distincts et indépendants – il nous faut un moyen de les manipuler ensemble.



MUSCLEZ VOS NEURONES

La Serveuse a toujours besoin d'appeler trois fois afficherMenu(), une fois pour chaque menu. Voyez-vous une façon de combiner les menus pour qu'il ne faille plus qu'un seul appel ? Ou peut-être pour ne transmettre à la Serveuse qu'un seul itérateur pour accéder à tous les menus ?

Pas si mal ! Il suffit de grouper les menus dans une ArrayList puis d'obtenir son itérateur pour accéder à chaque Menu. Le code de la Serveuse sera plus simple et il pourra gérer un nombre quelconque de menus.



On dirait que le chef a une idée. Essayons-la :

```
public class Serveuse {
    ArrayList menus;

    public Serveuse(ArrayList menus) {
        this.menus = menus;
    }

    public void afficherMenu() {
        Iterator itérateurMenus = menus.iterator();
        while(itérateurMenus.hasNext()) {
            Menu menu = (Menu)itérateurMenus.next();
            afficherMenu(menu.creerItérateur());
        }
    }

    void afficherMenu(Iterator itérateur) {
        while (itérateur.hasNext()) {
            Plat plat = (Plat)itérateur.next();
            System.out.print(plat.getNom() + ", ");
            System.out.print(plat.getPrix() + " -- ");
            System.out.println(plat.getDescription());
        }
    }
}
```

Maintenant nous avons simplement une ArrayList de menus.

Et nous itérons sur les menus, en transmettant l'itérateur de chacun d'eux à la méthode afficherMenu() surchargé.

Cette partie du code ne change pas.

L'idée semble bonne. Bien sûr, nous avons perdu les noms des menus, mais rien ne nous empêche d'en ajouter.

Juste au moment où nous pensions avoir terminé...

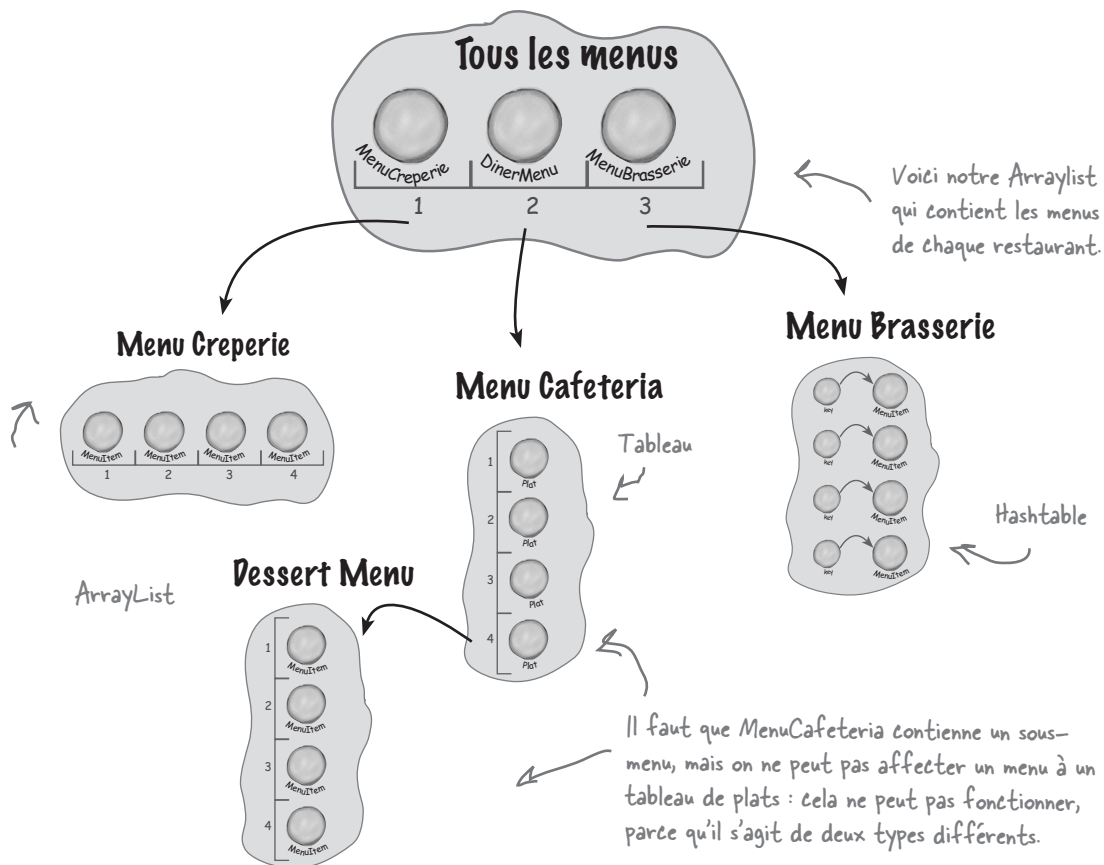
Maintenant, ils veulent ajouter un sous-menu pour les desserts.

Et puis quoi encore ? Maintenant, non seulement il va falloir prendre en charge plusieurs menus, mais aussi des menus dans des menus.

Tout irait bien si nous pouvions nous contenter de faire de la carte des desserts un élément de la collection MenuCafeteria, mais cela ne marchera pas, vu la façon dont elle est implémentée actuellement.



Ce que nous voulons (en gros) :



Mais c'est impossible !

On ne peut pas affecter un menu de desserts à un tableau de plats.

Il faut modifier quelque chose !

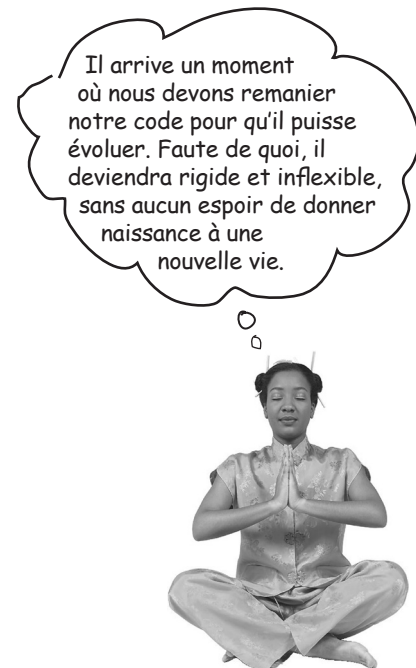
Que nous faut-il ?

Il est temps de prendre une décision déterminante et de retravailler l'implémentation des chefs afin d'obtenir une conception suffisamment générale pour qu'elle fonctionne avec tous les menus (et maintenant les sous-menus). Oui, nous allons dire aux chefs que le temps est venu pour nous de réimplémenter leurs menus.

Une fois un tel degré de complexité atteint, il nous faut retravailler la conception. Sinon, nous ne parviendrons jamais à gérer d'autres types de menus ni des sous-menus.

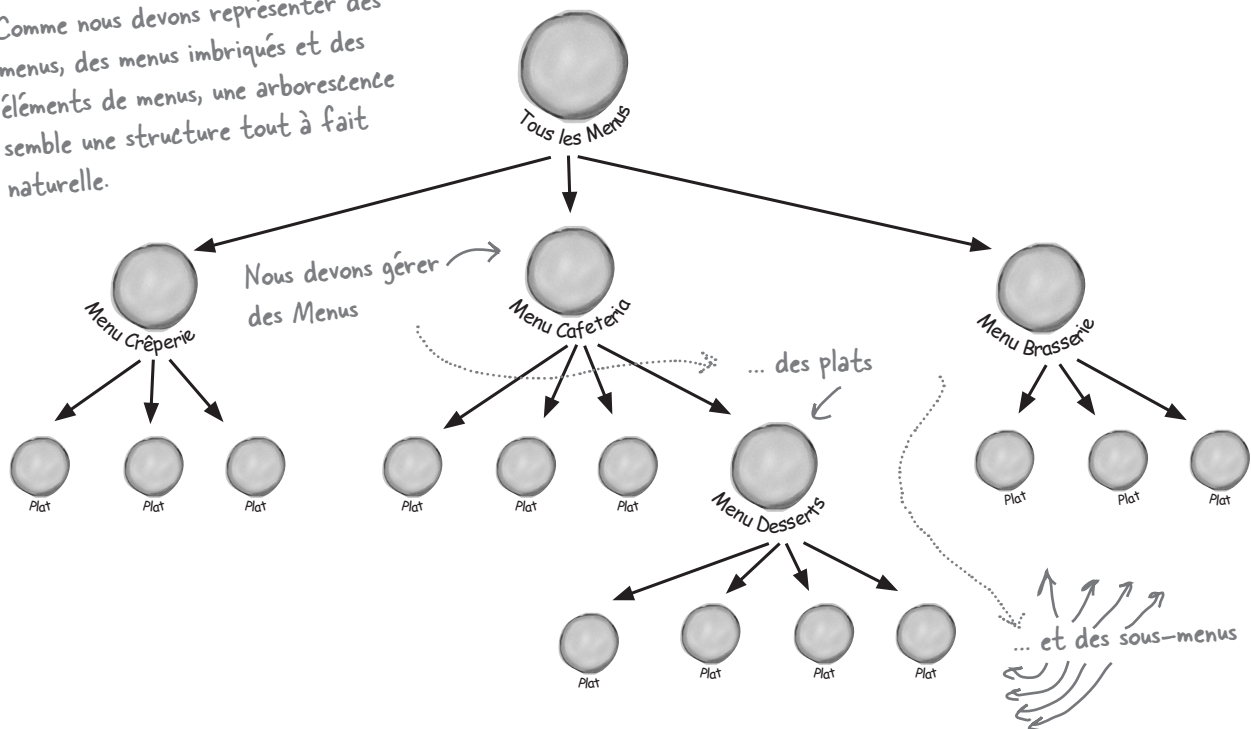
Alors ? Que nécessite donc réellement notre nouvelle conception ?

- Il nous faut une forme ou une autre de structure arborescente qui puisse prendre en compte des menus, des sous-menus et des éléments de menus (des plats).
- Nous devons être sûrs de disposer d'un moyen de parcourir les éléments de chaque menu qui soit aussi pratique que nos itérateurs actuels.
- Nous devons pouvoir naviguer dans les menus avec plus de souplesse. Par exemple, nous devrions pouvoir ne parcourir que la carte des desserts, ou bien itérer sur tout le menu de la Cafeteria, y compris la carte des desserts.

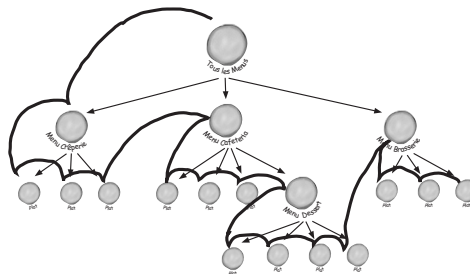


Il arrive un moment où nous devons remanier notre code pour qu'il puisse évoluer. Faute de quoi, il deviendra rigide et inflexible, sans aucun espoir de donner naissance à une nouvelle vie.

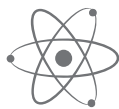
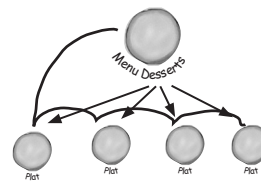
Comme nous devons représenter des menus, des menus imbriqués et des éléments de menus, une arborescence semble une structure tout à fait naturelle.



Nous avons toujours besoin de parcourir tous les éléments de l'arbre.



Et naviguer de façon plus souple, par exemple dans un seul menu.



MUSCLEZ VOS NEURONES

Comment traiteriez-vous cette nouvelle difficulté introduite par la modification des spécifications ?
Réfléchissez-y avant de tourner la page.

Le pattern Composite : définition

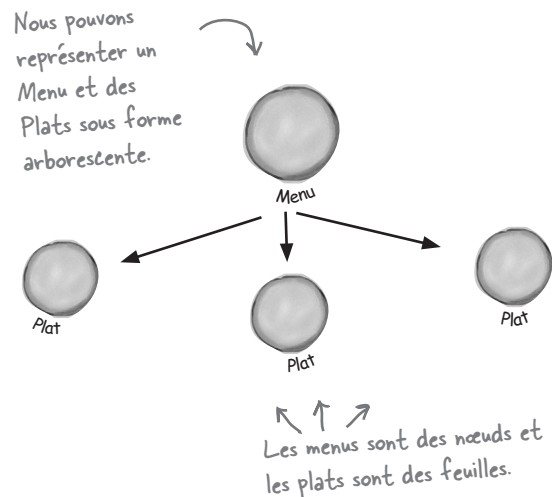
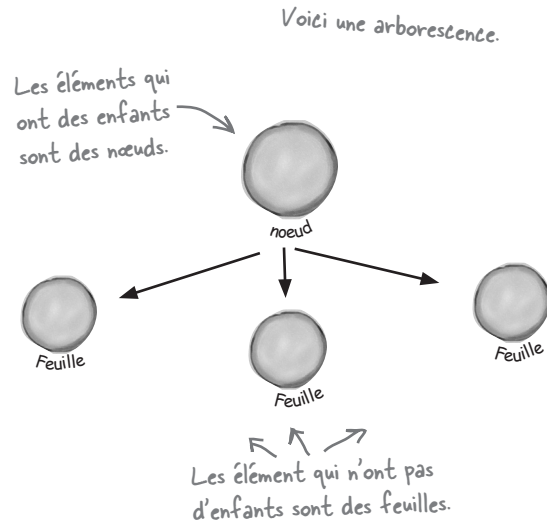
Oui, nous allons introduire un nouveau pattern pour résoudre ce problème. Nous n'avons pas abandonné Itérateur – il fait toujours partie de notre solution – mais le problème de la gestion des menus a atteint une nouvelle dimension qu'Itérateur ne peut pas prendre en compte. Nous allons donc prendre du recul et faire appel au pattern Composite.

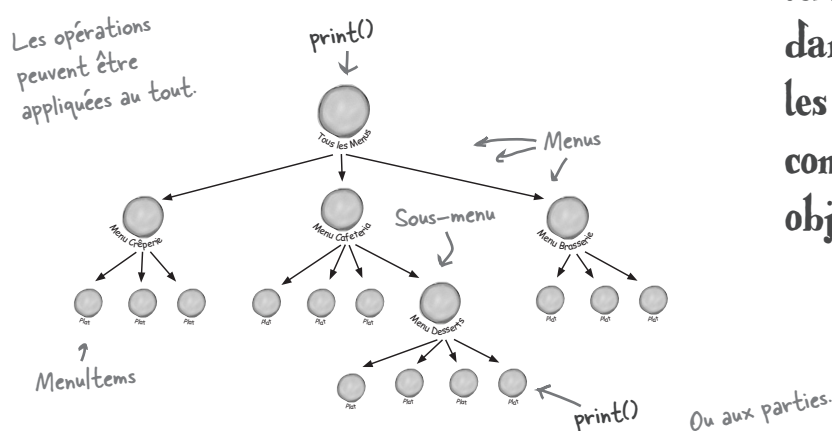
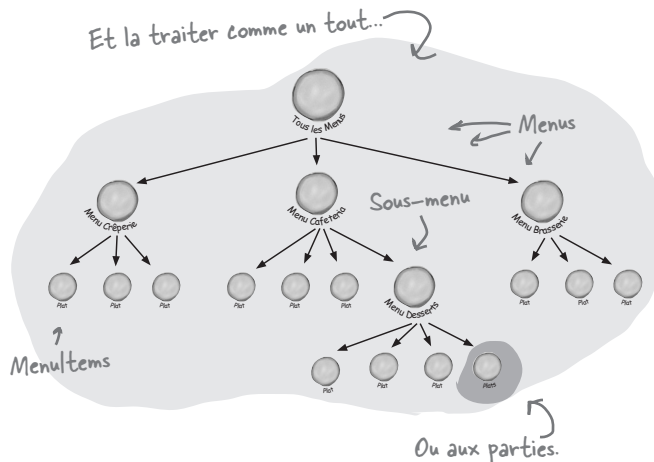
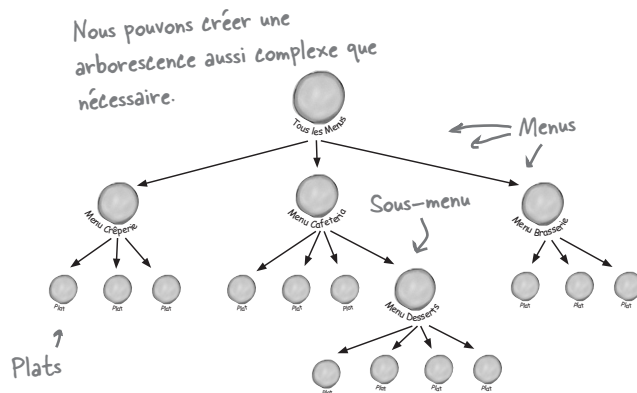
Nous irons droit au but avec ce pattern, et nous allons immédiatement vous donner la définition officielle :

Le pattern Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci.

Réfléchissons dans les termes de nos menus : ce pattern nous fournit un moyen de créer une arborescence capable de gérer un groupe de menus imbriqués et des éléments de menu dans la même structure. En plaçant les menus et les plats dans la même structure, nous créons une hiérarchie composant/composé, autrement dit une arborescence d'objets constituée de parties (menus et plats) mais qui peut être traitée comme un tout, à l'instar d'un super-menu.

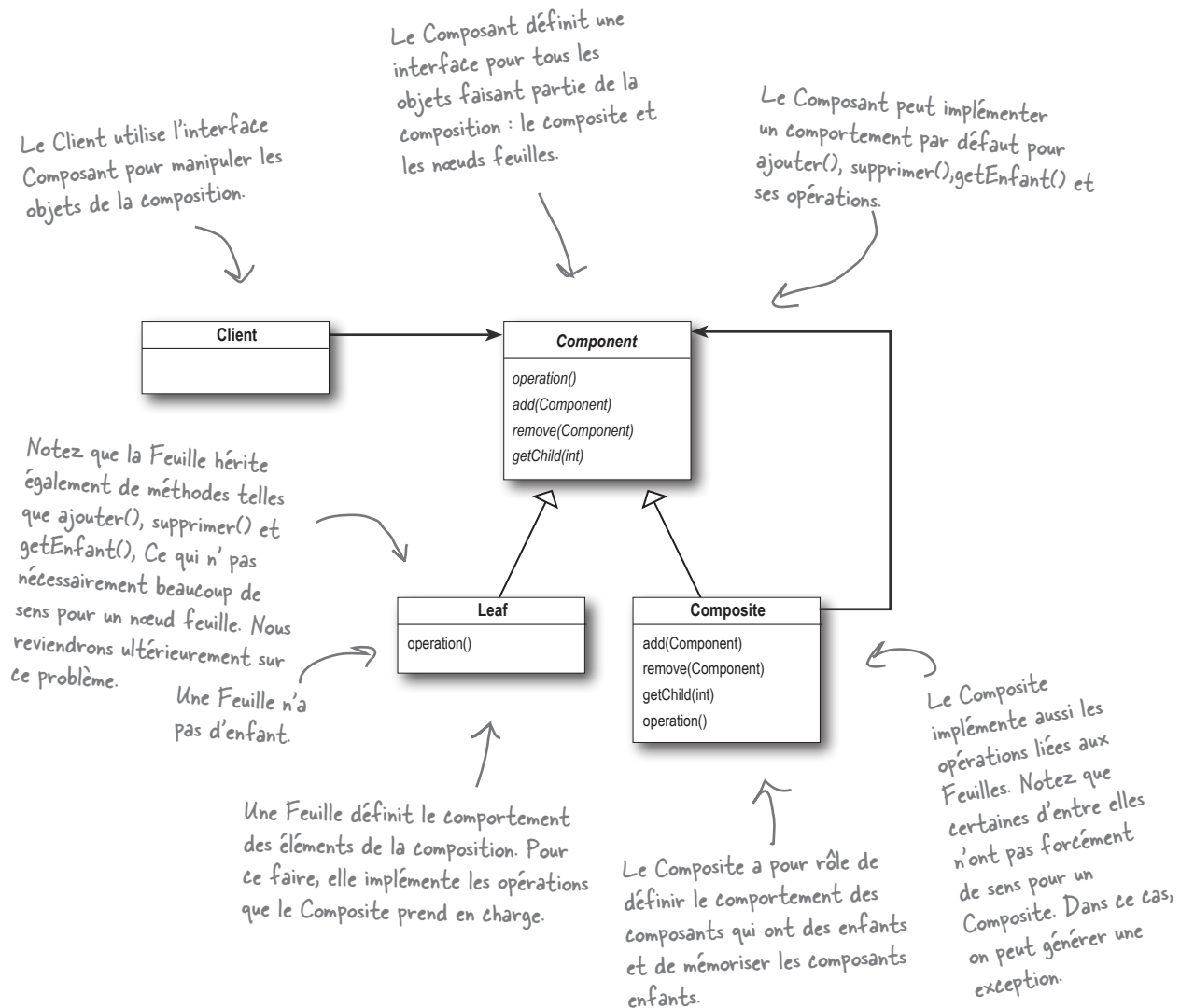
Une fois que nous avons notre super-menu, nous pouvons appliquer ce pattern pour « traiter de la même façon des objets individuels et des combinaisons ». Qu'est-ce à dire ? Cela signifie que si nous avons une arborescence de menus, de sous-menus, et éventuellement de sous-sous-menus ainsi que de plats, alors tout menu est une « composition » parce qu'il peut contenir à la fois des menus et des plats. Les objets individuels sont les plats : ils ne contiennent pas d'autres objets. Comme vous allez le voir, une conception conforme au pattern composite va nous permettre d'écrire un code simple qui peut appliquer la même opération (afficher par exemple !) à toute la structure de menus.





Le pattern Composite nous permet de créer des arborescences d'objets qui contiennent des nœuds : ce sont aussi bien des compositions d'objets que des objets individuels.

Avec une structure composite, Nous pouvons appliquer les mêmes opérations aux composites et aux composants. En d'autres termes, nous pouvons ignorer dans la plupart des cas les différences entre les compositions d'objets et les objets individuels.



Il n'y a pas de questions stupides

Q: Composant, Composite, Arborences ? Je m'y perds un peu.

R: Un composite contient des composants. Il y a deux sortes de composants : les composites et les feuilles. Vous avez dit « récursif » ? Oui. Un composite contient un ensemble d'enfants qui peuvent être à leur tour des composites ou des feuilles.

Quand vous organisez les données de cette manière, vous obtenez au final une arborescence (en réalité une arborescence inversée) dans laquelle la racine est un composite et dont les branches contiennent des composites qui se terminent par des nœuds feuilles.

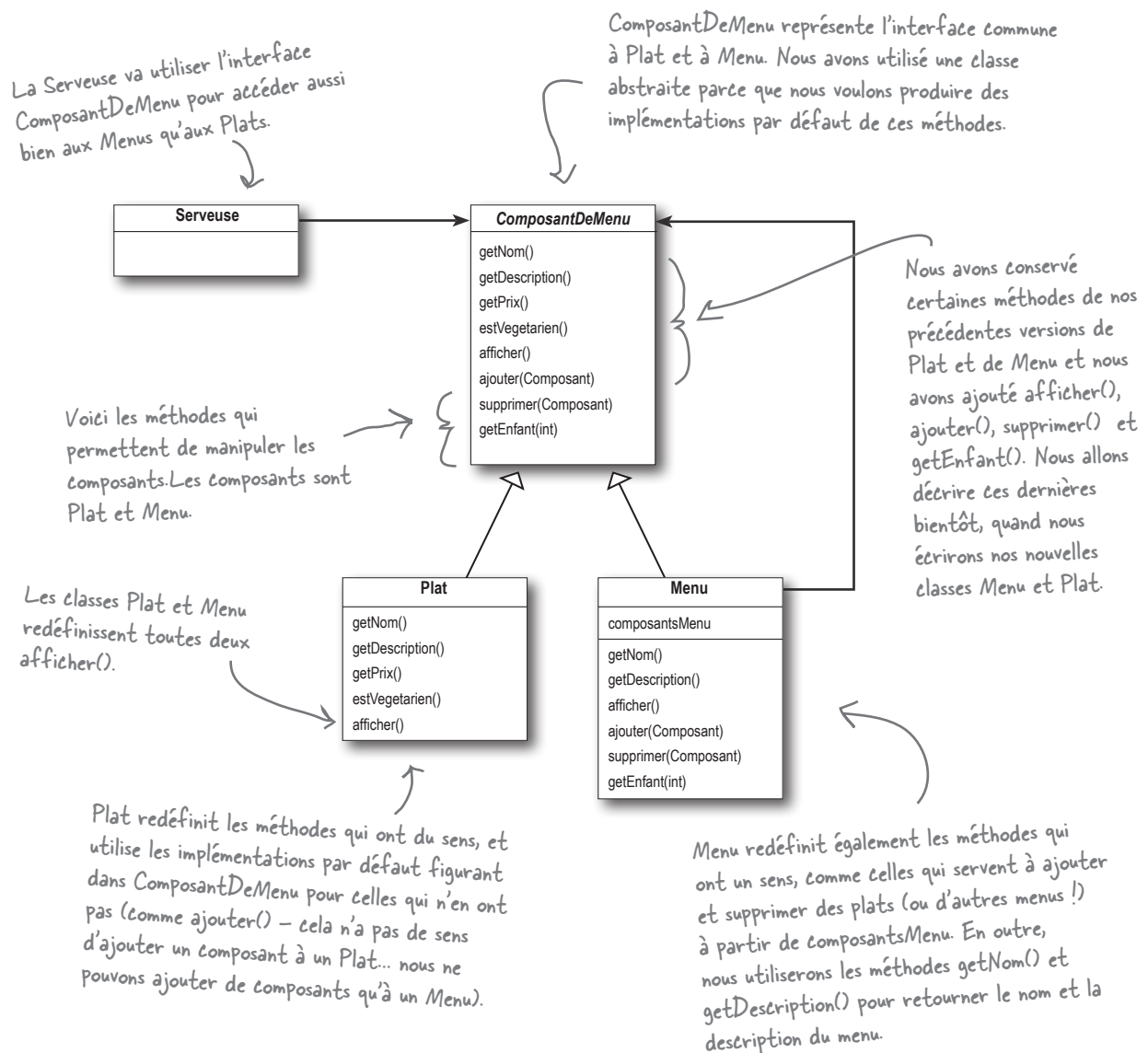
Q: Et quel est le rapport avec les itérateurs ?

R: Souvenez-vous que nous adoptons une nouvelle approche. Nous allons réimplémenter les menus avec une nouvelle solution : le pattern Composite. Ne vous attendez donc pas à ce que les itérateurs se transforment comme par magie en composites. Cela dit, les deux s'accrochent très bien l'un de l'autre. Vous verrez bientôt que nous pouvons utiliser des itérateurs dans une implémentation de composite.

Concevoir des Menus avec un Composite

Comment allons-nous donc appliquer le pattern Composite à nos menus ? Pour commencer, nous devons créer une interface composant ; celle-ci servira d'interface commune aux menus et aux plats et nous permettra de les traiter de la même manière. Autrement dit, nous pourrions appeler les mêmes méthodes sur les menus et sur les plats.

Maintenant, cela n'a peut-être pas beaucoup de sens d'appeler certaines méthodes sur un plat ou sur un menu, mais nous pouvons le gérer et nous allons le faire dans un moment. Pour l'instant, jetons un coup d'œil à une esquisse de la façon dont les menus vont s'inscrire dans la structure du pattern Composite :



Implémenter ComposantDeMenu

Bien. Nous allons commencer par la classe abstraite `ComposantDeMenu`. Souvenez-vous : son rôle consiste à fournir une interface abstraite aux nœuds feuilles et aux nœuds composites. Maintenant, vous vous demandez peut-être si `ComposantDeMenu` ne joue pas deux rôles. Cela pourrait bien être le cas et nous reviendrons sur ce point. Toutefois, nous allons fournir pour l'instant une implémentation par défaut des méthodes, de sorte que si le `Plat` (la feuille) ou le `Menu` (le composite) ne veut pas implémenter certaines méthodes (comme `getEnfant()` pour un nœud feuille), ils puissent se rabattre sur un comportement de base :

ComposantDeMenu fournit une implémentation par défaut pour chaque méthode.

```
public abstract class ComposantDeMenu {

    public void ajouter(ComposantDeMenu composantDeMenu) {
        throw new UnsupportedOperationException();
    }
    public void supprimer(ComposantDeMenu composantDeMenu) {
        throw new UnsupportedOperationException();
    }
    public ComposantDeMenu getEnfant(int i) {
        throw new UnsupportedOperationException();
    }
    public String getNom() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrix() {
        throw new UnsupportedOperationException();
    }
    public boolean estVegetarien() {
        throw new UnsupportedOperationException();
    }
    public void afficher() {
        throw new UnsupportedOperationException();
    }
}
```

Tous les composants doivent implémenter l'interface `ComposantDeMenu`. Toutefois, comme les feuilles et les nœuds jouent des rôles différents, nous ne pouvons pas définir une implémentation par défaut qui ait un sens pour chaque méthode. Parfois, le mieux que vous puissiez faire est de lancer une exception à l'exécution.

Comme certaines de ces méthodes n'ont de sens que pour les `Plats` et que d'autres n'ont de sens que pour les `Menus`, l'implémentation par défaut consiste à lancer une `UnsupportedOperationException`. De cette façon, si `Plat` ou `Menu` ne prend pas en charge une opération, ils n'ont rien d'autre à faire que d'hériter l'implémentation par défaut.

Nous avons regroupé les méthodes « composites » – autrement dit celles qui permettent d'accéder aux `ComposantDeMenu`, d'en ajouter et d'en supprimer.

Voici les méthodes « opération », celles qui sont utilisées par les `Plats`. Il s'avère que nous pouvons en employer également deux dans le code du `Menu`, comme vous le verrez dans quelques pages.

`afficher()` est une méthode « opération » que nos `Menus` et nos `Plats` implémenteront, mais nous fournissons une implémentation par défaut.

Implémenter le Plat

Bien, voyons maintenant la classe Plat. Souvenez-vous : c'est la classe feuille dans le diagramme de Composite et elle implémente le comportement des éléments du composite.

```
public class Plat extends CompositantDeMenu {
    String nom;
    String description;
    boolean vegetarien;
    double prix;

    public Plat(String nom,
                String description,
                boolean vegetarien,
                double prix)
    {
        this.nom = nom;
        this.description = description;
        this.vegetarien = vegetarien;
        this.prix = prix;
    }

    public String getNom() {
        return nom;
    }

    public String getDescription() {
        return description;
    }

    public double getPrix() {
        return prix;
    }

    public boolean estVegetarien() {
        return vegetarien;
    }

    public void afficher() {
        System.out.print(" " + getNom());
        if (estVegetarien()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrix());
        System.out.println(" -- " + getDescription());
    }
}
```

Nous devons d'abord étendre l'interface CompositantDeMenu.

Le constructeur accepte le nom, la description, etc. et mémorise une référence à chacun. Voilà qui est très semblable à notre ancienne implémentation des plats.

Voici nos méthodes d'accès – tout comme dans notre implémentation précédente.

Ce code diffère de l'implémentation précédente. Ici, nous redéfinissons la méthode afficher() de la classe CompositantDeMenu. Pour Plat, cette méthode affiche la totalité de l'entrée de menu : nom, description, prix et << (v) >> si c'est un plat végétarien.

Je suis heureux que nous prenions cette direction. Je pense que cela va me permettre d'implémenter ce menu de crêpes dont j'ai toujours rêvé.



Implémenter le Composite

Maintenons que nous avons le Plat, il nous reste la classe du composite, que nous allons nommer Menu. Souvenez-vous : le composite peut contenir des Plats ou d'autres Menus. Il y a deux méthodes de ComposantDeMenu que cette classe n'implémente pas : `getPrix()` et `estVegetarien()`, parce qu'elle n'ont pas beaucoup de sens pour un Menu.

Menu est également un ComposantDeMenu, tout comme Plat.

Menu peut avoir un nombre quelconque d'enfants de type ComposantDeMenu : nous utiliserons une ArrayList pour les contenir.

Ceci diffère de notre ancienne implémentation : nous allons attribuer à chaque Menu un nom et une description. Auparavant, nous nous reposions simplement sur le fait d'avoir des classes différentes.

Voici comment on ajoute des Plats ou d'autres Menus à un Menu. Comme les Plats et les Menus sont des ComposantDeMenu, il nous suffit d'une seule méthode pour les deux.

Vous pouvez également supprimer un ComposantDeMenu ou accéder à un ComposantDeMenu.

Voici nos méthodes get pour obtenir le nom et la description.

Remarquez que nous ne redéfinissons ni `getPrix()` ni `estVegetarien()` parce que ces méthodes n'ont pas de sens pour un menu (même si l'on peut soutenir que `estVegetarien()` en a un. Si quelqu'un essaie d'appeler ces méthodes sur un Menu, il reçoit une `UnsupportedOperationException`.

Pour afficher le menu, nous affichons son nom et sa description

```
public class Menu extends ComposantDeMenu {
    ArrayList composantsMenu = new ArrayList();
    String nom;
    String description;

    public Menu(String nom, String description) {
        this.nom = nom;
        this.description = description;
    }

    public void ajouter(ComposantDeMenu composantDeMenu) {
        composantsMenu.add(composantDeMenu);
    }

    public void supprimer(ComposantDeMenu composantDeMenu) {
        composantsMenu.remove(composantDeMenu);
    }

    public ComposantDeMenu getEnfant(int i) {
        return (ComposantDeMenu) composantsMenu.get(i);
    }

    public String getNom() {
        return nom;
    }

    public String getDescription() {
        return description;
    }

    public void afficher() {
        System.out.print("\n" + getNom());
        System.out.println(", " + getDescription());
        System.out.println("-----");
    }
}
```




Une petite seconde, Je ne comprends pas l'implémentation d'afficher(). Je pensais que j'étais censé pouvoir appliquer les mêmes opérations à un composite qu'à une feuille. Si j'applique affiche() à un composite avec cette implémentation, je n'obtiens rien d'autre qu'un simple nom de menu et une description. Je n'obtiens pas l'affichage du COMPOSITE.

Bien vu. Comme le menu est un composite et contient aussi bien des plats que d'autres menus, sa méthode affiche() devrait afficher tout ce qu'il contient. Sinon, nous devrions parcourir tout le composite et afficher chaque élément nous mêmes. On dirait que cela compromet l'intérêt d'avoir une structure composite.

Comme vous allez le voir, il est facile d'implémenter correctement affiche() parce que nous pouvons compter sur le fait que chaque composant sait s'afficher lui-même. Une petite merveille de récursivité ! Vérifions :

Corriger la méthode affiche()

```
public class Menu extends ComposantDeMenu {
    ArrayList composantsMenu = new ArrayList();
    String nom;
    String description;

    // code du constructeur

    // autres méthodes
```

```
public void affiche() {
    System.out.print("\n" + getNom());
    System.out.println(", " + getDescription());
    System.out.println("-----");
```

```
    Iterator itérateur = composantsMenu.iterator();
    while (itérateur.hasNext()) {
        ComposantDeMenu composantDeMenu =
            (ComposantDeMenu) itérateur.next();
        composantDeMenu.affiche();
    }
}
```

Il suffit de modifier la méthode affiche() pour qu'elle affiche non seulement les informations sur ce Menu, mais aussi tous ses composants : d'autres Menus et des Plats.



Regardez ! Nous utilisons un itérateur. Il nous sert à parcourir tous les éléments du Menu... Il peut s'agir d'autres Menus ou bien de Plats. Comme les Menus ET les Plats implémentent affiche(), nous nous contentons d'appeler affiche() et c'est à eux de faire le reste.

NOTE : Si, durant cette itération, Nous rencontrons un autre objet Menu, sa méthode affiche() lancera une autre itération, et ainsi de suite.

Préparer le test...

Il est presque temps de tester ce code, mais il faut d'abord mettre à jour le code de la Serveuse. Après tout, c'est elle le principal client :

```
public class Waitress {
    MenuComponent allMenus;
```

```
    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }
```

```
    public void printMenu() {
        allMenus.print();
    }
```

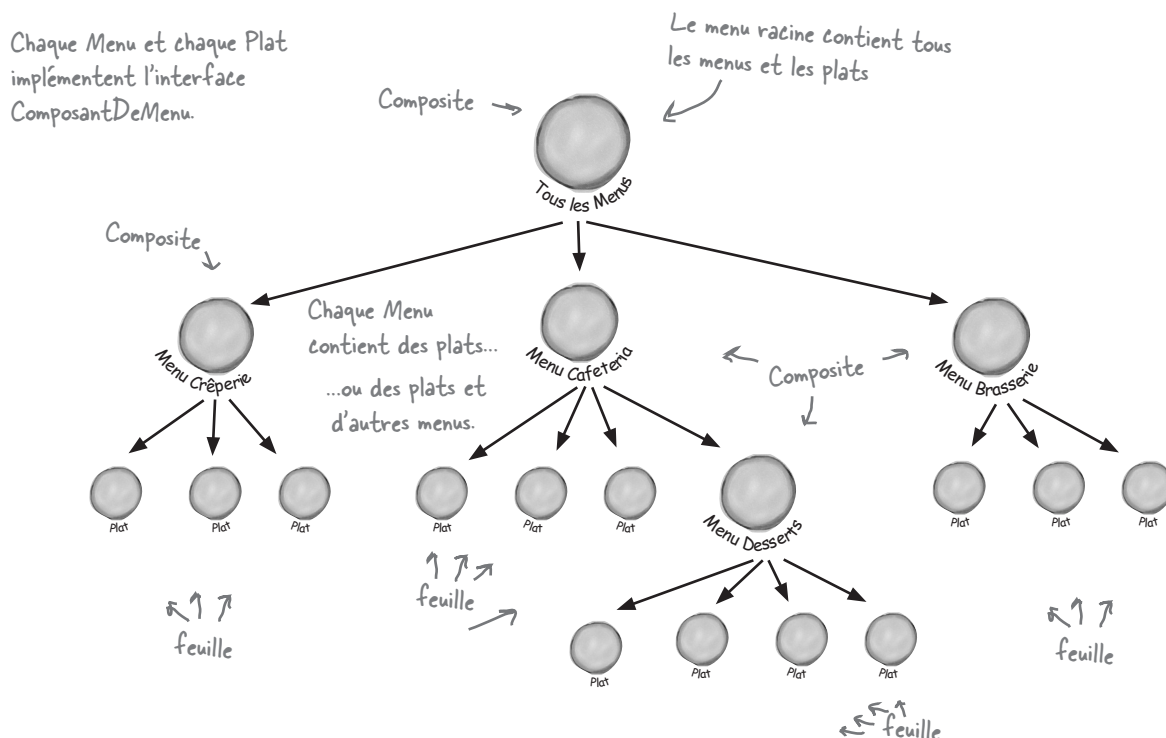
```
}
```

Et voilà ! Le code de la Serveuse n'est pas plus compliqué. Maintenant, il suffit de lui transmettre le composant racine, celui qui contient tous les autres menus. Nous l'avons nommé tousMenus.

Pour afficher toute la hiérarchie de menus – tous les menus et tous les plats – elle n'a rien d'autre à faire qu'appeler afficher() sur le menu racine.

Nous allons avoir une Serveuse très heureuse.

Parfait ! Il nous reste une dernière chose à faire avant d'exécuter notre test : nous faire une idée de l'aspect qu'aura le menu composite au moment de l'exécution :



Écrire le code du test...

Il n'y a plus qu'à écrire le test. Contrairement à la précédente version, nous allons gérer toute la création des menus dans le code du test. Nous pourrions demander à chaque chef de nous fournir son nouveau menu, mais mieux vaut tester le tout d'abord. Voici le code :

```
public class TestMenu {
    public static void main(String args[]) {
        ComposantDeMenu menuCreperie =
            new Menu("MENU CRÊPERIE", "Brunch");
        ComposantDeMenu menuCafeteria =
            new Menu("MENU CAFETERIA", "Déjeuner");
        ComposantDeMenu menuBrasserie =
            new Menu("MENU BRASSERIE", "Dîner");
        ComposantDeMenu menuDesserts =
            new Menu("MENU DESSERT", "Rien que des desserts !");
        ComposantDeMenu tousMenus = new Menu("TOUS LES MENUS", "Toutes nos offres");

        tousMenus.ajouter(menuCreperie);
        tousMenus.ajouter(menuCafeteria);
        tousMenus.ajouter(menuBrasserie);

        // ajouter des plats

        menuCafeteria.ajouter(new Plat(
            "Pasta al pesto",
            "Spaghetti, ail, basilic et parmesan",
            true,
            3.89));

        menuCafeteria.ajouter(menuDesserts);
        menuDesserts.ajouter(new Plat(
            "Tarte du chef",
            "Tarte aux pommes avec une boule de glace vanille",
            true,
            1.59));

        // ajouter des plats

        Serveuse serveuse = new Serveuse(tousMenus);

        serveuse.afficherMenu();
    }
}
```

Créons d'abord les objets menus.

Il nous faut également un menu racine que nous allons nommer tousMenus.

Nous utilisons la méthode ajouter() du Composite pour ajouter chaque menu au menu racine, tousMenus.

Il faut maintenant ajouter tous les plats. Voici un exemple. Pour le reste, consultez le code source complet.

Et nous ajoutons également un menu à un menu. Une seule chose intéresse menuCafeteria : tout ce qu'il contient, qu'il s'agisse d'un plat ou d'un menu, est un ComposantDeMenu.

On ajoute une tarte aux pommes à la carte des desserts...

Une fois que nous avons construit toute la hiérarchie de menus, nous transmettons le tout à la Serveuse. Comme vous l'avez constaté, l'affichage va être du gâteau !

Exécuter le test...

NOTE : ce résultat est basé sur le code source complet.

```
Fichier Édition Fenêtre Aide OmeletteAuxFinesHerbes
% java TestMenu

TOUS LES MENUS, Toutes nos offres
-----

MENU CRÊPERIE, Brunch
-----
Crêpe à l'oeuf(v), 2.99
  -- Crêpe avec oeuf au plat ou brouillé
Crêpe complète, 2.99
  -- Crêpe avec oeuf au plat et jambon
Crêpe forestière(v), 3.49
  -- Myrtilles fraîches et sirop de myrtille
Crêpe du chef(v), 3.59
  -- Crème fraîche et fruits rouges au choix

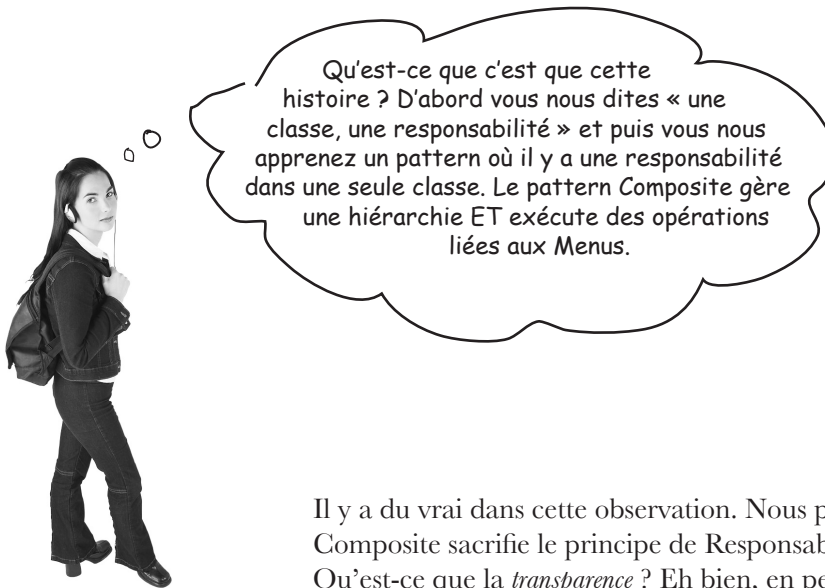
MENU CAFETERIA, Déjeuner
-----
Salade printanière(v), 2.99
  -- Salade verte, tomates, concombre, olives, pommes de terre
Salade Parisienne, 2.99
  -- Salade verte, tomates, poulet, emmental
Soupe du jour(v), 3.29
  -- Soupe du jour et croûtons grillés
Quiche aux fruits de mer, 3.05
  -- Pâte brisée, crevettes, moules, champignons
Quiche aux épinards(v), 3.99
  -- Pâte feuilletée, pommes de terre, épinards, crème fraîche
Pasta al pesto(v), 3.89
  -- Spaghetti, ail, basilic, parmesan

MENU DESSERT, Rien que des desserts !
-----
Tarte du chef(v), 1.59
  -- Tarte aux pommes et boule de glace à la vanille
Charlotte maison(v), 1.99
  -- Charlotte aux poires et sauce au chocolat
Duos de sorbets(v), 1.89
  -- Une boule fraise et une boule citron vert

MENU BRASSERIE, Dîner
-----
Omelette sarladaise(v), 3.99
  -- Omelette aux champignons et pommes sautées
Soupe de poissons, 3.69
  -- Soupe de poissons, rouille et croûtons
Tagliatelles Primavera(v), 4.29
  -- Pâtes fraîches, brocoli, petits pois, crème fraîche
%
```

Voici tous nos menus... Nous avons affiché tout ceci en appelant simplement afficher() sur le menu racine.

La nouvelle carte de desserts est affichée quand nous affichons tous les composants du menu de la Cafeteria.



Il y a du vrai dans cette observation. Nous pourrions dire que le pattern Composite sacrifie le principe de Responsabilité unique à la transparence. Qu'est-ce que la *transparence* ? Eh bien, en permettant à une interface Composant de contenir les opérations de gestion des enfants ET les opérations des feuilles, nous permettons également à un client de traiter les composites et les nœuds feuilles de manière uniforme. La nature d'un élément – composite ou nœud feuille – devient ainsi transparente pour le client.

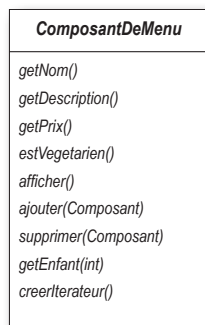
Maintenant, étant donné que nous avons les deux types d'opérations dans la classe Composant, nous y perdons quelque peu en *sécurité* parce qu'un client peut essayer d'appliquer une opération inappropriée ou sans objet à un élément (par exemple tenter d'ajouter un menu à un plat). C'est là un choix de conception : nous pourrions prendre une autre direction, séparer les responsabilités et les attribuer à des interfaces. Notre conception serait plus sûre, au sens où tout appel de méthode injustifié serait intercepté lors de la compilation ou de l'exécution, mais nous perdriions de la transparence et notre code devrait contenir des instructions conditionnelles et utiliser l'opérateur `instanceof`.

Pour revenir à votre question, il s'agit là d'un cas classique de compromis. Nous nous guidons sur des principes, mais nous devons toujours observer l'effet qu'ils ont sur nos conceptions. Parfois, nous procédons à dessein d'une façon qui semble enfreindre le principe, mais c'est dans certains cas une question de perspective. Par exemple, il peut paraître incorrect d'avoir des opérations de gestion des éléments enfants au niveau des feuilles (comme `ajouter()`, `supprimer()` et `getEnfant()`), mais, là encore, vous pouvez toujours changer de point de vue et voir une feuille comme un nœud qui n'a pas d'enfants.

Flash-back sur Itérateur

Quelques pages en arrière, nous vous avons promis de vous montrer comment combiner un Itérateur et un Composite. Vous savez que nous utilisons déjà Itérateur dans notre implémentation interne de la méthode `afficher()`, mais nous voulons également permettre à la Serveuse de parcourir la totalité d'un composite si elle en a besoin, par exemple si elle veut lire tout le menu et extraire les plats végétariens.

Pour implémenter un itérateur avec Composite, ajoutons une nouvelle méthode `creerIterateur()` dans chaque composant. Nous commencerons par la classe abstraite `ComposantDeMenu` :



Nous avons ajouté une méthode `creerIterateur()` au `ComposantDeMenu`. Cela signifie que chaque `Menu` et chaque `Plat` devront implémenter cette méthode. Cela signifie également que l'appel de `creerIterateur()` sur un composite doit s'appliquer à tous les enfants du composite.

Nous devons maintenant implémenter cette méthodes dans les classes `Menu` et `Plat` :

```

public class Menu extends ComposantDeMenu {

    // le reste du code ne change pas

    public Iterator creerIterateur() {
        return new ItérateurComposite(composantsMenu.iterator());
    }

}
  
```

Ici, nous utilisons un nouvel itérateur nommé `ItérateurComposite`. Il sait comment parcourir n'importe quel composite. Nous lui passons l'itérateur courant du composite.

```

public class Plat extends ComposantDeMenu {
    // le reste du code ne change pas

    public Iterator creerIterateur() {
        return new ItérateurNull();
    }

}
  
```

Et maintenant, le `Plat`...

Hein ? D'où sort cet `ItérateurNull` ? Vous allez bientôt le savoir.

L'itérateur Composite

L'itérateur Composite est un itérateur SÉRIEUX. Il a pour tâche de parcourir les éléments du composant et de s'assurer que tous les Menus enfants (et les petits-enfants, etc.) sont inclus.

Voici le code. Attention, il n'est pas bien long mais il peut être un peu casse-tête. Il suffit de répéter ce mantra : « la récursion est mon amie, la récursion est mon amie ».



```
import java.util.*;
```

Comme pour tous les itérateurs, nous implémentons l'interface java.util.Iterator.

```
public class ItérateurComposite implements Iterator {
    Stack pile = new Stack();
```

L'itérateur du composite racine que nous allons parcourir est transmis. Nous utilisons une pile comme structure de données.

```
public ItérateurComposite(Iterator itérateur) {
    pile.push(itérateur);
}
```

```
public Object next() {
    if (hasNext()) {
        Iterator itérateur = (Iterator) pile.peek();
        ComposantDeMenu composant = (ComposantDeMenu) itérateur.next();
        if (composant instanceof Menu) {
            pile.push(composant.creerItérateur());
        }
        return composant;
    } else {
        return null;
    }
}
```

Bien. Quand le client veut accéder à l'élément suivant, nous vérifions qu'il y en a un en appelant hasNext()...

S'il y a un élément, nous dépilons l'itérateur courant et nous lisons son élément suivant.

Si cet élément est un menu, nous avons un autre composite qui doit être inclus dans l'itération. Nous le plaçons donc sur la pile. Dans les deux cas, nous retournons le composant.

```
public boolean hasNext() {
    if (pile.empty()) {
        return false;
    } else {
        Iterator itérateur = (Iterator) pile.peek();
        if (!itérateur.hasNext()) {
            pile.pop();
            return hasNext();
        } else {
            return true;
        }
    }
}
```

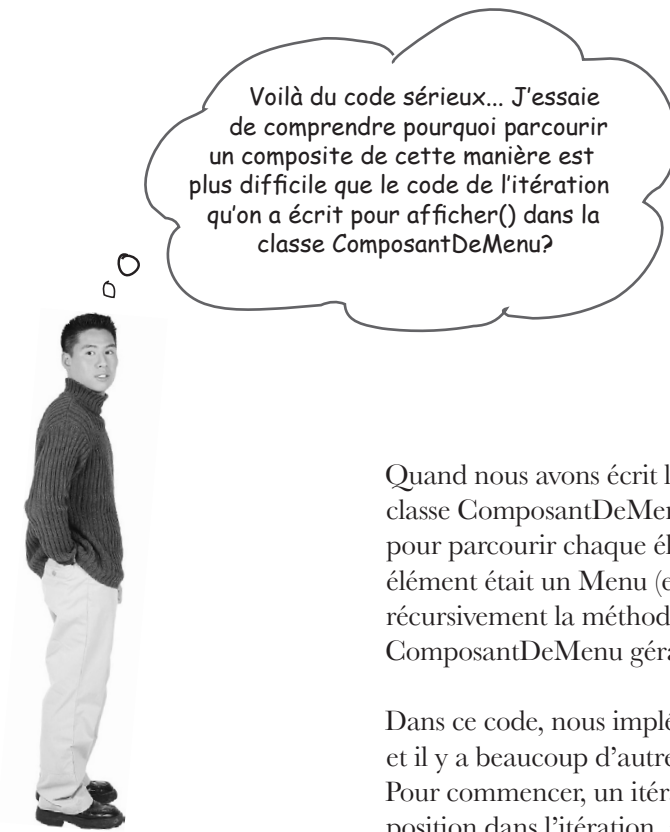
Pour savoir s'il y a un élément suivant, nous regardons si la pile est vide. Si oui, il n'y en a pas.

Sinon, nous extrayons l'itérateur du sommet de la pile et nous regardons s'il a un élément suivant. S'il n'en a pas, nous le dépilons et nous appelons hasNext() récursivement.

Sinon, il y a un élément et nous retournons true.

```
public void remove() {
    throw new UnsupportedOperationException();
}
}
```

Nous ne prenons en charge que la navigation, pas la suppression.



Voilà du code sérieux... J'essaie de comprendre pourquoi parcourir un composite de cette manière est plus difficile que le code de l'itération qu'on a écrit pour afficher() dans la classe ComposantDeMenu?

Quand nous avons écrit la méthode `afficher()` dans la classe `ComposantDeMenu`, nous avons utilisé un itérateur pour parcourir chaque élément du composant. Si cet élément était un `Menu` (et non un `Plat`), nous appelions récursivement la méthode `afficher()`. Autrement dit, le `ComposantDeMenu` gérait l'itération lui-même, en *interne*.

Dans ce code, nous implémentons un itérateur *externe* et il y a beaucoup d'autres choses à prendre en compte. Pour commencer, un itérateur externe doit maintenir sa position dans l'itération, afin qu'un client externe puisse piloter l'itération en appelant `hasNext()` et `next()`. Mais, dans ce cas, notre code a également besoin de maintenir cette position sur une structure composite, récursive. C'est pourquoi nous utilisons une pile pour maintenir notre position à mesure que nous déplaçons dans la hiérarchie du composite.



Tracez un diagramme des Menus et des Plats. Puis imaginez que vous êtes l'IterateurComposite et que votre tâche consiste à gérer les appels de hasNext() et de next(). Indiquez la façon dont l'IterateurComposite navigue dans cette structure quand ce code est exécuté :

```
public void testIterateurComposite(ComposantDeMenu composant) {  
    IterateurComposite itereur = new IterateurComposite(composant.iterateur);  
  
    while(iterateur.hasNext()) {  
        ComposantDeMenu composant = itereur.next() ;  
    }  
}
```


L'itérateur null

Maintenant, revenons à l'itérateur null. On peut le voir de la manière suivante. Un Plat n'a rien qu'on puisse parcourir, d'accord ? Alors comment gère-t-on l'implémentation de sa méthode `creerIterateur()` ? Eh bien, nous avons deux solutions :

NOTE : Un autre exemple du
« design pattern » Objet nul.

Solution 1 :

Retourner null

Nous pourrions retourner null depuis `creerIterateur()`, mais il faudrait des instructions conditionnelles dans le code client pour savoir si null a été retourné ou non.

Solution 2 :

Retourner un itérateur qui retourne toujours false quand `hasNext()` est appelée

Cela semble plus judicieux. Nous pouvons toujours retourner un itérateur, mais le client n'a jamais à se soucier de savoir si null a été retourné ou non. En effet, nous créons un itérateur mais il n'y a pas d'opération.

La seconde solution semble bien meilleure. Appelons-la `IterateurNull` et implémentons-la.

```
import java.util.Iterator;

public class NullIterator implements Iterator {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Voilà l'itérateur le plus paresseux
du monde : à chaque étape, il se
tourne les pouces.

← Quand `next()` est appelée, nous retournons null.

← Plus important encore : quand `hasNext()` est
appelée, nous retournons toujours false.

← Et `IterateurNull` ne songe même pas
à s'occuper des suppressions.

Donnez-moi le menu végétarien

Maintenant, nous disposons d'une façon de parcourir chaque élément du Menu. Utilisons-la et donnons à notre Serveuse une méthode qui nous indique exactement quels plats sont végétariens.

```
public class Serveuse {
    ComposantDeMenu tousMenus;

    public Serveuse(ComposantDeMenu tousMenus) {
        this.tousMenus = tousMenus;
    }

    public void afficherMenu() {
        tousMenus.afficher();
    }

    public void afficherMenuVegetarien() {
        Iterator itereur = tousMenus.creerIterateur();
        System.out.println("\nMENU VÉGÉTARIEN\n----");
        while (itereur.hasNext()) {
            ComposantDeMenu composantDeMenu =
                (ComposantDeMenu) itereur.next();
            try {
                if (composantDeMenu.estVegetarien()) {
                    composantDeMenu.afficher();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

La méthode `afficherMenuVegetarien()` prend le composite `tousMenus` et obtient son itérateur. Ce sera notre `IterateurComposite`.

Parcourir chaque élément du composite.

Appeler la méthode `estVegetarien()` de chaque élément. Si vrai, appeler sa méthode `afficher()`.

`afficher()` n'est appelée que sur les Plats, jamais sur les Composites. Voyez-vous pourquoi ?

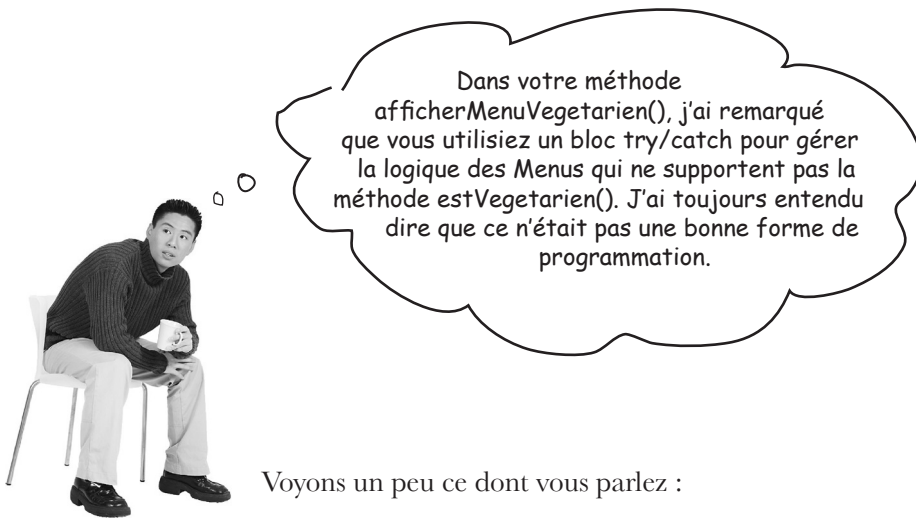
Nous avons implémenté `estVegetarien()` pour que les Menus lancent toujours une exception. Si cela se produit, nous interceptons l'exception mais nous poursuivons l'itération.

La magie d'Itérateur & Composite en duo...

Eh bien ! Il en a fallu un effort de développement pour parvenir jusqu'ici ! Maintenant, nous avons une structure de menus générale qui devrait durer un certain temps au nouvel empire de la restauration. C'est le moment de se reposer et de commander quelque chose :

```
Fichier Édition Fenêtre Aide OmeletteAuxChampignons
% java TestMenu
MENU VÉGÉTARIEN
----
Crêpe à l'oeuf(v), 2.99
  -- Crêpe avec oeuf au plat ou brouillé
Crêpe forestière(v), 3.49
  -- Myrtilles fraîches et sirop de myrtille
Crêpe du chef(v), 3.59
  -- Crème fraîche et fruits rouges au choix
Salade printanière(v), 2.99
  -- Salade verte, tomates, concombre, olives, pommes de terre
Soupe du jour(v), 3.29
  -- Soupe du jour et croûtons grillés
Quiche aux épinards(v), 3.99
  -- Pâte feuilletée, pommes de terre, épinards, crème fraîche
Pasta al pesto(v), 3.89
  -- Spaghetti, ail, basilic, parmesan
Tarte du chef(v), 1.59
  -- Tarte aux pommes et boule de glace à la vanille
Charlotte maison(v), 1.99
  -- Charlotte aux poires et sauce au chocolat
Duo de sorbets(v), 1.89
  -- Une boule fraise et une boule citron vert
Tarte du chef(v), 1.59
  -- Tarte aux pommes et boule de glace à la vanille
Charlotte maison(v), 1.99
  -- Charlotte aux poires et sauce au chocolat
Duo de sorbets(v), 1.89
  -- Une boule fraise et une boule citron vert
Omelette sarladaise(v), 3.99
  -- Omelette aux champignons et pommes sautées
Tagliatelles Primavera(v), 4.29
  -- Pâtes fraîches, brocoli, petits pois, crème fraîche
%
```

← Le menu végétarien est composé de tous les plats végétariens de chaque menu.



Voyons un peu ce dont vous parlez :

```
try {
    if (composantDeMenu.estVegetarien()) {
        composantDeMenu.afficher();
    }
} catch (UnsupportedOperationException) {}
```

Nous appelons `estVegetarien()` sur tous les `ComposantDeMenu`, mais les Menus lancent une exception parce qu'ils ne prennent pas en charge l'opération.

Si le composant de menu ne prend pas en charge l'opération, nous nous contentons de lancer l'exception et de l'ignorer.

En général, nous sommes d'accord : `try/catch` sert à gérer les erreurs, non la logique applicative. Quelles sont nos autres options ?

Nous aurions pu vérifier le type du composant à l'exécution avec `instanceof` pour vérifier que c'est un `Plat` avant d'appeler `estVegetarien()`. Mais nous perdriions du même coup en *transparence* parce que nous ne traiterions pas les Menus et les Plats de la même manière.

Nous pourrions également modifier la méthode `estVegetarien()` dans les Menus pour qu'elle retourne faux. C'est une solution simple qui préserve la transparence.

Dans notre solution, nous recherchons la clarté : nous voulons réellement communiquer l'idée que cette opération n'est pas prise en charge sur le Menu (ce qui n'équivaut pas à dire que `estVegetarien()` est faux). Cela permet également à un développeur d'implémenter une méthode `estVegetarien()` raisonnable pour le Menu et qu'elle fonctionne avec le code existant.

C'est notre choix, et nous nous y tenons.



Interview

Cette semaine :

Le pattern Composite nous parle de problèmes d'implémentation

DPTLP : Nous nous entretenons ce soir avec le pattern Composite. Si vous nous parliez un peu de vous, Composite ?

Composite : Bien sûr... Je suis le pattern à utiliser quand vous avez des collections d'objets avec des relations tout-partie et que vous voulez pouvoir traiter ces objets de la même manière.

DPTLP : Bien, entrons dans le vif du sujet... Qu'entendez-vous par « relations tout-partie » ?

Composite : Imaginez une interface utilisateur graphique, une IHM. On y trouve souvent un composant principal, comme un cadre ou un panneau, qui contient d'autres composants : menus, zones de texte, barres de défilement et boutons. Votre IHM est donc composée de plusieurs parties, mais, quand vous l'affichez, vous vous la représentez généralement comme un tout. Vous dites au composant principal de s'afficher et vous comptez sur lui pour afficher les parties qui le composent. Nous appelons objets composites les composants qui contiennent d'autres composants, et les objets qui ne contiennent pas d'autres composants sont les objets feuilles.

DPTLP : C'est ce que vous entendez par « traiter les objets de la même manière » ? Avoir des méthodes communes que vous pouvez appeler sur les composites et les feuilles ?

Composite : Exactement. Je peux dire à un objet composite de s'afficher ou à un objet feuille de s'afficher et ils se comporteront comme il faut. Le composite s'affichera en disant à tous ses composants de s'afficher.

DPTLP : Cela implique que tous les objets aient la même interface. Que se passe-t-il si les objets de votre composite font des choses différentes ?

Composite : Eh bien, pour que le composite fonctionne de manière transparente pour le client, vous devez implémenter la même interface pour tous les objets du composite, sinon le client doit savoir quelle interface chaque objet implémente, ce qui est contraire au but recherché. De toute évidence, cela signifie que vous aurez à un moment donné des objets pour lesquels les appels de méthodes n'auront pas de sens.

DPTLP : Et comment résolvez-vous le problème ?

Composite : Eh bien, il y a plusieurs solutions. Vous pouvez vous contenter de ne rien faire ou de retourner null ou false, selon ce qui convient à votre application. Vous pouvez aussi être plus proactif et lancer une exception. Bien sûr, il faut que le client accepte de coopérer un peu et vérifie que l'appel de méthode n'a rien provoqué d'inattendu.

DPTLP : Mais si le client ne sait pas à quel objet il a affaire, comment pourra-t-il jamais savoir quel appel effectuer sans vérifier le type ?

Composite : Si vous êtes quelque peu créatif, vous pouvez structurer vos méthodes pour que l'implémentation par défaut fasse quelque chose qui ait un sens. Par exemple, si le client appelle `getEnfant()` sur le composite, cela a un sens. Et cela a également un sens sur une feuille, si vous voyez la feuille comme un objet qui n'a pas d'enfants.

DPTLP : Ah... futé ! Mais j'ai entendu dire que certains clients étaient si contrariés par ce problème qu'ils exigeaient des interfaces séparées pour les différents objets, afin qu'il soit impossible d'effectuer des appels de méthode qui n'ont pas de sens. Est-ce que c'est toujours le pattern Composite ?

Composite : Oui. C'est une version beaucoup plus sûre du pattern Composite, mais elle nécessite que le client vérifie le type de chaque objet avant de réaliser un appel, pour que l'objet soit typé correctement.

DPTLP : Dites-nous en un peu plus sur la façon dont le composite et les objets feuilles sont structurés.

Composite : C'est généralement une arborescence, une sorte de hiérarchie. La racine est le composite de haut niveau, et tous ses enfants sont d'autres composites ou des feuilles.

DPTLP : Est-ce qu'il arrive que les enfants pointent dans l'autre sens, vers leurs parents ?

Composite : Oui, un composant peut avoir un pointeur vers un parent pour faciliter la navigation dans la structure. Et si vous avez une référence vers un enfant et que vous devez le supprimer, il faudra que le parent supprime l'enfant. La référence au parent facilite également cette opération.

DPTLP : Il y a vraiment beaucoup de choses à prendre en compte dans votre implémentation. Y a-t-il d'autres facteurs à considérer quand on implémente le pattern Composite ?

Composite : Oui, il y en a... L'un d'eux est l'ordre des enfants. Que se passe-t-il si un composite doit avoir des enfants ordonnés de façon particulière ? Dans ce cas, vous aurez besoin d'un schéma plus élaboré pour ajouter et supprimer les enfants, et vous devrez prêter une attention particulière à la façon de naviguer dans la hiérarchie.

TLP : Un point intéressant. Je n'y avais pas pensé.

Composite : Et aviez-vous pensé à la mise en cache ?

DPTLP : La mise en cache ?

Composite : Oui, la mise en cache. Parfois, si la structure composite est complexe ou coûteuse à traverser, il est utile d'implémenter une mise en cache des nœuds. Par exemple, si vous parcourez en permanence un composite pour calculer un résultat donné, vous pouvez implémenter un cache qui mémorise le résultat temporairement pour économiser des traversées.

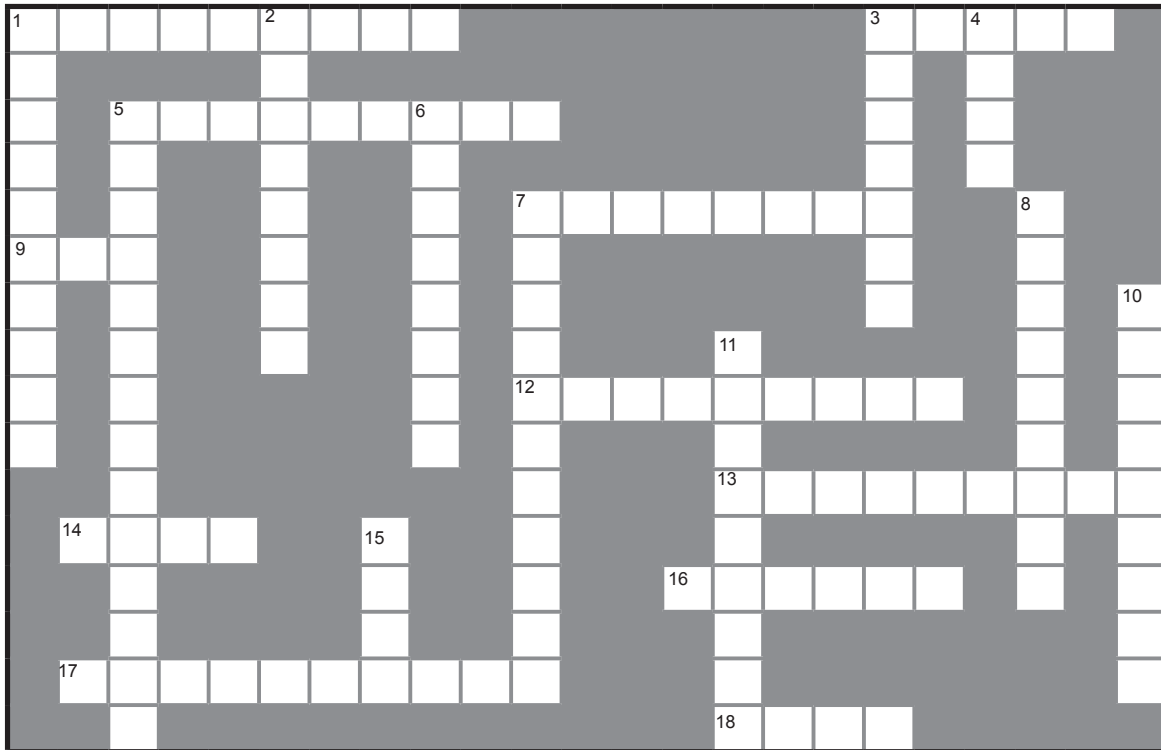
DPTLP : Eh bien, je n'aurais pas pensé que le pattern Composite recelait tant de surprises. Avant de nous quitter, encore une question. Selon vous, quel est votre point fort le plus significatif ?

Composite : Assurément, le fait de simplifier la vie de mes clients. Comme ils n'ont pas besoin de savoir s'ils ont affaire à un objet composite ou à un objet feuille, ils n'ont pas besoin d'écrire des instructions if partout pour être sûrs d'appeler les bonnes méthodes sur les bons objets. Ils peuvent souvent effectuer un seul appel de méthode et exécuter une opération sur toute une structure.

DPTLP : Cela semble un bénéfice important. Vous êtes sans nul doute un pattern très utile pour collectionner et manipuler des objets. Mais nous sommes en retard sur l'horaire... Merci d'être venu et à bientôt pour une prochaine émission.



Il est de nouveau temps...



Horizontalement

1. Les packages d'interfaces utilisateurs utilisent souvent ce pattern pour leurs composants.
3. Les menus de la Brasserie serviront pour le _____.
5. Parcours.
7. Peut être abstraite.
9. Les plats d'Objectville n'en manquent pas.
12. Parcourt une collection.
14. On en trouve normalement dans une cafétéria.
16. La Cafétéria et la Crêperie ont réalisé une _____.
17. Hashtable et ArrayList implémentent cette interface.
18. Il utilise une ArrayList pour implémenter ses menus.

Verticalement

1. Contenus dans des composites.
2. Maintenant, elle connaît Java !
3. Ce menu nous a obligé à modifier toute notre implémentation.
4. Il utilise un tableau pour implémenter ses menus.
5. Nous l'avons encapsulée.
6. A remplacé Enumeration.
7. On utilise généralement ce pattern pour créer des itérateurs.
8. L'itérateur composite en utilise beaucoup.
10. Cette classe permet de créer des tales de hachage.
11. Collection et Iterator sont dans ce package.
13. Nous avons supprimé ItérateurMenueCreperie parce que cette classe fournit déjà un itérateur.
15. Collection ordonnée.

Qui fait quoi ?

Faites correspondre chaque pattern avec sa description :

Pattern	Description
Stratégie	Les clients traitent les collections d'objets et les objets individuels de la même manière
Adaptateur	Fournit un moyen de naviguer dans une collection d'objets sans en exposer l'implémentation
Itérateur	Simplifie l'interface d'un groupe de classes
Façade	Modifie l'interface d'une ou plusieurs classes
Composite	Permet à un groupe d'objets d'être informé du changement d'un état
Observateur	Encapsule des comportements interchangeables et emploie la délégation pour décider lequel utiliser



Votre boîte à outils de concepteur

Deux nouveaux patterns pour votre boîte à outils : deux façons géniales de gérer des collections d'objets.

Principes OO

Encapsulez ce qui varie.
Préférez la composition à l'héritage
Programmez des interfaces, non des implémentations.
Efforcez-vous de coupler faiblement les objets qui interagissent.
Les classes doivent être ouvertes à l'extension mais fermées à la modification.
Dépendez des abstractions. Ne dépendez pas des classes concrètes.
Ne parlez qu'à vos amis
Ne nous appelez pas, nous vous appellerons.
Une classe ne doit avoir qu'une seule raison de changer

Bases

abstraction
encapsulation
polymorphisme
héritage

encore un autre principe pour gérer le changement dans une conception.

Patterns OO

Stratégie - fournit un moyen d'accéder en séquence à un objet de type agrégat sans révéler sa représentation sous-jacente

Itérateur - fournit un moyen d'accéder en séquence à un objet de type agrégat sans révéler sa représentation sous-jacente

Façade - fournit une interface unifiée à l'ensemble des interfaces d'un système

Composite - compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci

Un autre chapitre avec deux patterns pour le prix d'un.



POINTS D'IMPACT

- Un itérateur permet d'accéder aux éléments d'un agrégat sans exposer sa structure interne.
- Un itérateur a pour tâche de parcourir un agrégat et encapsule l'itération dans un autre objet.
- Lorsque nous utilisons un itérateur, nous soulageons l'agrégat de la responsabilité de prendre en charge les opérations nécessaires pour naviguer dans ses données.
- Un itérateur fournit une interface commune pour parcourir les éléments d'un agrégat. Il permet ainsi d'exploiter le polymorphisme pour écrire le code qui utilise ces éléments.
- Il convient de s'efforcer de n'affecter qu'une seule responsabilité à chaque classe.
- Le pattern Composite fournit une structure qui peut contenir à la fois des objets individuels et des composites.
- Le pattern Composite permet aux clients de traiter les composites et les objets individuels de la même manière.
- Un composant est un objet quelconque dans une structure de composite. Les composants peuvent être d'autres composites ou des nœuds feuilles.
- Une conception implémentant un composite entraîne de nombreux compromis. Vous devez équilibrer transparence et sûreté en fonction de vos besoins.



Solutions des exercices



À vos crayons

Par rapport à notre implémentation d'afficherMenu(), quelles sont les affirmations qui sont vraies ?

- ☒ A. Nous codons pour des implémentations concrètes de MenuCreperie et de MenuCafeteria, non pour une interface.
- ☐ B. La Serveuse n'implémentant pas l'API Java Waitress, elle n'adhère pas à un standard.
- ☐ C. Si nous décidions de passer de l'emploi de MenuCafeteria à un autre type de menu qui implémenterait sa liste de plats sous forme de Hashtable, le code de Serveuse nécessiterait beaucoup de modifications.
- ☒ D. La Serveuse doit savoir comment chaque menu représente sa collection de plats interne, ce qui viole l'encapsulation.
- ☒ E. Nous avons du code dupliqué : la méthode afficherMenu() a besoin de deux boucles séparées pour parcourir les deux types de menus. Et si nous ajoutons un troisième menu, il nous faudrait une nouvelle boucle.
- ☐ F. L'implémentation n'étant pas basée sur MXML (Menu XML), elle ne présente pas l'interopérabilité désirable.



À vos crayons

Avant de regarder la page suivante, notez rapidement les trois modifications nécessaires pour adapter ce code à la structure de notre application :

1. implémenter l'interface Menu
2. nous débarrasser de getPlats()
3. ajouter creerIterateur() et retourner un itérateur capable de parcourir les valeurs de la table.



Solution du jeu du frigo

La classe `IterateurMenuCafeteriaAlternateur` reconstituée :

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public class IterateurMenuCafeteriaAlternateur
```

```
implements Iterator
```

```
{
```

```
    Plat[] plats;
    int position;
```

```
    public IterateurMenuCafeteriaAlternateur(Plat[] plats) {
```

```
{
```

```
        this.plats = plats;
        Calendar maintenant = Calendar.getInstance();
        position = maintenant.get(Calendar.DAY_OF_WEEK) % 2;
```

```
}
```

```
    public boolean hasNext() {
```

```
        if (position >= plats.length || plats[position] == null) {
            return false;
        } else {
            return true;
        }
    }
```

```
}
```

```
    public Object next() {
```

```
        Plat plat = plats[position];
        position = position + 2;
        return menuItem;
```

```
}
```

```
    public void remove() {
```

```
        throw new UnsupportedOperationException(
            "IterateurMenuCafeteriaAlternateur ne supporte pas remove()");
```

```
}
```

```
}
```

Remarquez que cette
implémentation ne supporte
pas `remove()`

✱ Qui fait quoi ? ✱

Faites correspondre chaque pattern avec sa description :

Pattern	Description
Stratégie	Les clients traitent les collections d'objets et les objets individuels de la même manière
Adaptateur	Fournit un moyen de naviguer dans une collection d'objets sans en exposer l'implémentation
Itérateur	Simplifie l'interface d'un groupe de classes
Façade	Modifie l'interface d'une ou plusieurs classes
Composite	Permet à un groupe d'objets d'être informé du changement d'un état
Observateur	Encapsule des comportements interchangeables et emploie la délégation pour décider lequel utiliser

[illegible]

✧ L'état des choses ✧



J'avais tellement cru que la vie serait facile à Objectville... Mais maintenant, chaque fois que j'arrive au bureau, je reçois une autre demande de changement. Je suis sur le point de craquer ! Ah, si j'avais été plus assidue au groupe de patterns de Lucie du mercredi soir... Je suis dans un de ces états !

Un fait méconnu : les patterns Stratégie et État sont des jumeaux qui ont été séparés à la naissance. Comme vous le savez, le pattern Stratégie a créé une affaire des plus florissantes sur le marché des algorithmes interchangeables. Quant à lui, État a peut-être suivi une voie plus noble : il aide les objets à contrôler leur comportement en modifiant leur état interne. On le surprend souvent à dire à ses clients « Répétez après moi : Je suis bon, je suis intelligent, ma simple présence suffit... »

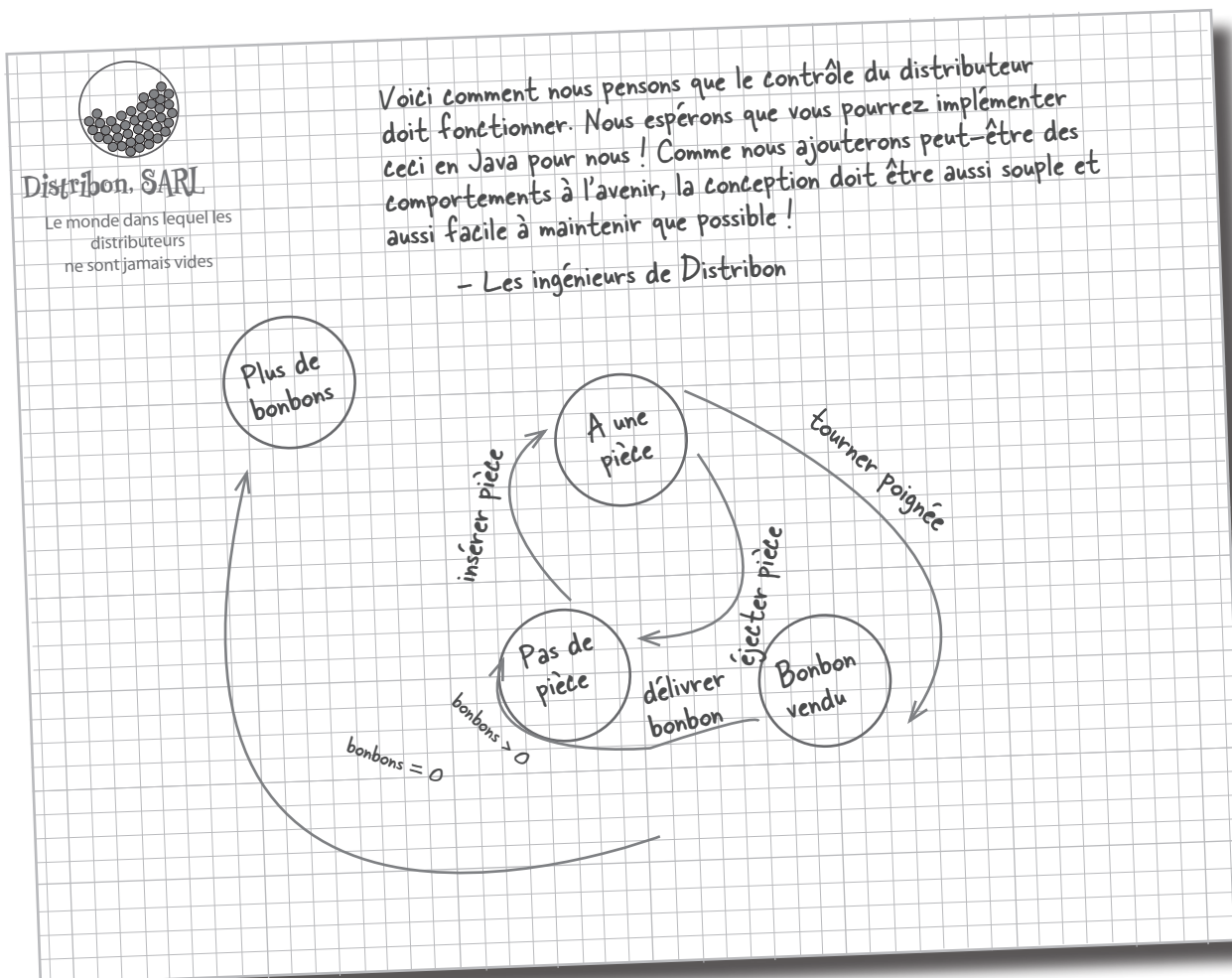
Nouveau casse-tête

Les grilles-pain Java font terriblement « années 90 ». Aujourd'hui, on intègre Java à de *vraies* machines, comme les distributeurs de bonbons. Oui ! Les distributeurs de bonbons sont à la pointe de la technologie. Les plus grands fabricants se sont rendu compte que l'insertion d'une unité centrale dans leurs appareils permettait d'augmenter les ventes, de gérer les stocks en ligne et de mesurer la satisfaction des consommateurs avec une précision accrue.

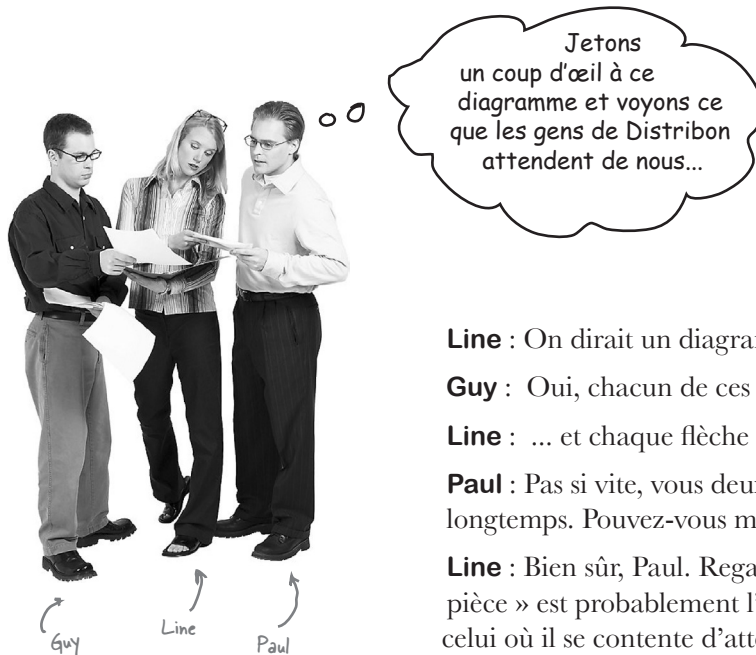
Mais ces fabricants sont des experts en distributeurs de bonbons, pas des développeurs. C'est pourquoi ils ont sollicité votre aide :



Du moins, c'est ce qu'ils disent. Nous pensons qu'ils étaient fatigués d'une technologie qui remonte presque au dix-neuvième siècle et qu'il leur fallait trouver un moyen de rendre leur travail plus motivant.



Conversation dans un box



Line : On dirait un diagramme d'états.

Guy : Oui, chacun de ces cercles est un état...

Line : ... et chaque flèche représente une transition.

Paul : Pas si vite, vous deux. J'ai étudié les diagrammes d'état il y a longtemps. Pouvez-vous me rappeler de quoi vous parlez ?

Line : Bien sûr, Paul. Regarde ces cercles : ce sont les états. « Pas de pièce » est probablement l'état initial du distributeur de bonbons, celui où il se contente d'attendre que tu insères une pièce. Tous les états sont simplement les différentes configurations de l'appareil qui se

comportent d'une certaine manière, et qui ont besoin qu'une action donnée les fasse passer dans un autre état.

Guy : Oui, tu vois, pour qu'il y ait un autre état, tu dois faire quelque chose, par exemple insérer une pièce dans la machine. Tu vois la flèche entre « Pas de pièce » et « A une pièce » ?

Paul : Oui...

Guy : Cela signifie simplement que l'appareil est dans l'état « Pas de pièce », et que si tu insères une pièce il passera à l'état « A une pièce ». C'est ce qu'on appelle une transition.

Paul : Oh, Je vois ! Et si je suis dans l'état « A une pièce », je peux tourner la poignée et passer à l'état « Bonbon vendu » ou éjecter la pièce et revenir à l'état « Pas de pièce ».

Line : C'est cela, Paul ! Pour en revenir au diagramme, il n'a pas l'air trop mal. De toute évidence, nous avons quatre états, et je pense que nous avons aussi quatre actions : « insérer pièce », « éjecter pièce », « tourner poignée » et « délivrer bonbon ». Mais... quand nous délivrons le bonbon, nous testons pour savoir s'il reste zéro bonbon ou plus dans l'état « Bonbon vendu », puis nous passons soit à l'état « Plus de bonbons » soit à l'état « Pas de pièce ». Nous avons donc cinq transitions d'un état à un autre.

Line : Ce test pour savoir s'il reste des bonbons implique également qu'on doit mémoriser le nombre de bonbons. Chaque fois que la machine nous donne un bonbon, ce peut être le dernier. Et si c'est le cas, nous devons passer à l'état « plus de bonbons ».

Guy : N'oubliez pas non plus que l'utilisateur pourrait faire quelque chose d'absurde, par exemple essayer d'éjecter la pièce quand l'appareil est dans l'état « Pas de pièce » ou encore insérer deux pièces.

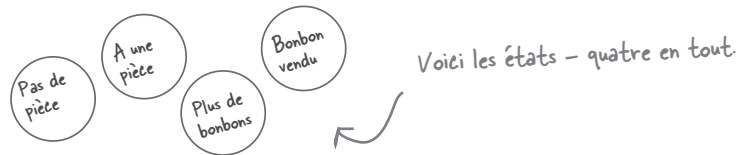
Paul : Oh, je n'y avais pas pensé. Nous devons également nous en occuper.

Guy : Pour chaque action possible, il suffit de regarder dans quel état nous sommes et d'agir en conséquence. Nous pouvons le faire ! Commençons par faire correspondre le diagramme d'états au code...

Machines à états : première mouture

Comment allons-nous passer de ce diagramme d'états au code ? Voici une rapide introduction à l'implémentation des machines à états :

- 1 Commençons par rassembler nos quatre états :



- 2 Puis créons une variable d'instance pour contenir l'état courant et définissons des valeurs pour chaque état :

Appelons « Plus de bonbons » « Epuisé » pour abrégé.

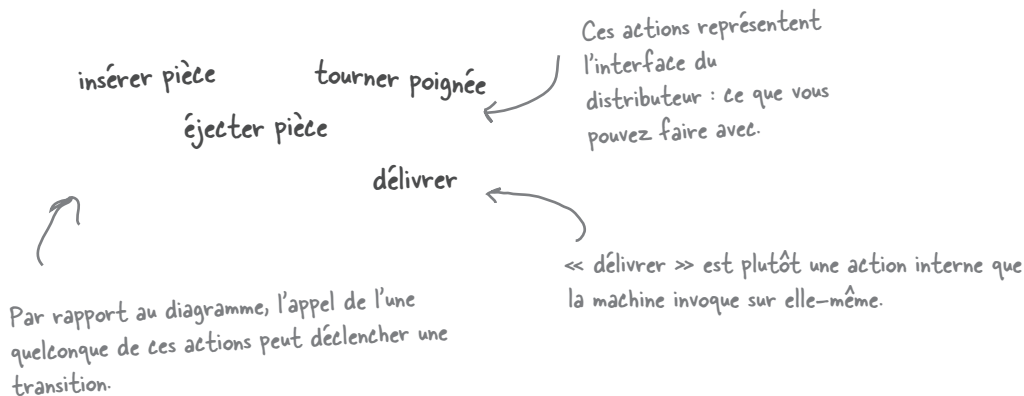
```
final static int EPUISE = 0;
final static int SANS_PIECE = 1;
final static int A_PIECE = 2;
final static int VENDU = 3;
```

```
int etat = EPUISE;
```

Voici chaque état représenté par un entier unique...

...et voilà la variable d'instance qui contient l'état courant. Nous la positionnons à « Epuisé » puisque l'appareil est vide quand on le sort de sa boîte et qu'on le met en service.

- 3 Nous recherchons maintenant toutes les actions qui peuvent se produire dans le système.



- 4 Maintenant, nous créons une classe qui va jouer le rôle de machine à états. Pour chaque action, nous créons une méthode qui utilise des instructions conditionnelles pour déterminer le comportement approprié à chaque état. Par exemple, pour l'action d'insérer une pièce, nous pourrions écrire une méthode comme celle-ci :

```
public void insererPiece() {
    if (etat == A_PIECE) {
        System.out.println("Vous ne pouvez plus insérer de pièces");
    } else if (etat == EPUISE) {
        System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock");
    } else if (etat == VENDU) {
        System.out.println("Veuillez patienter, le bonbon va tomber");
    } else if (etat == SANS_PIECE) {
        etat = A_PIECE;
        System.out.println("Vous avez inséré une pièce");
    }
}
```

Chaque état possible est testé à l'aide d'une instruction conditionnelle...

...et présente le comportement approprié pour chaque état possible...

...mais peut également déclencher une transition vers d'autres états, comme le montre le diagramme..

Il s'agit ici d'une technique courante : modéliser un état dans un objet en créant une variable d'instance qui contient les valeurs de l'état et écrire des instructions conditionnelles dans nos méthodes pour gérer les différents états.

Après ce bref aperçu, implémentons notre distributeur de bonbons !



Écrire le code

Il est temps d'implémenter notre Distributeur. Nous savons que nous allons avoir une variable d'instance qui contiendra l'état courant. À partir de là, il suffit de gérer toutes les actions, tous les comportements et toutes les transitions qui peuvent se produire. Pour les actions, nous devons implémenter l'insertion d'une pièce, l'éjection d'une pièce, l'action sur la poignée et la délivrance du bonbon. Il nous faut également implémenter le code qui teste si la machine est vide ou non.

```

public class Distributeur {

    final static int EPUISE = 0;
    final static int SANS_PIECE = 1;
    final static int A_PIECE = 2;
    final static int VENDU = 3;

    int etat = EPUISE;
    int nombre = 0;

    public Distributeur(int nombre) {
        this.nombre = nombre;
        if (nombre > 0) {
            etat = SANS_PIECE;
        }
    }

    public void insererPiece() {
        if (etat == A_PIECE) {
            System.out.println("Vous ne pouvez plus insérer de pièces");
        } else if (etat == SANS_PIECE) {
            etat = A_PIECE;
            System.out.println("Vous avez inséré une pièce");
        } else if (etat == EPUISE) {
            System.out.println("Vous ne pouvez pas insérer de pièces, nous sommes en rupture de stock");
        } else if (etat == VENDU) {
            System.out.println("Veuillez patienter, le bonbon va tomber");
        }
    }
}

```

Voici les quatre états : ils correspondent à ceux du diagramme d'états de Distribon.

Voici la variable d'instance qui va mémoriser l'état courant. Nous commençons dans l'état EPUISE.

Nous avons une deuxième variable d'instance qui mémorise le nombre de bonbons restant dans la machine.

Le constructeur accepte le stock de bonbons initial. Si le stock est différent de zéro, la machine entre dans l'état SANS_PIECE, signifiant qu'elle attend que quelqu'un insère une pièce. Sinon, elle reste dans l'état EPUISE.

Maintenant, nous commençons à implémenter les actions sous forme de méthodes...

Quand une pièce est insérée, si...

...une autre pièce est déjà insérée, nous l'indiquons au client ; sinon, nous acceptons la pièce et déclenchons une transition vers l'état A_PIECE.

et si la machine est en rupture de stock, nous éjectons la pièce.

Si le client vient d'acheter un bonbon, il doit attendre que la transaction soit terminée avant d'insérer une autre pièce.


```

public void ejecterPiece() {
    if (etat == A_PIECE) {
        System.out.println("Pièce retournée");
        etat = SANS_PIECE;
    } else if (etat == SANS_PIECE) {
        System.out.println("Vous n'avez pas inséré de pièce");
    } else if (etat == VENDU) {
        System.out.println("Vous avez déjà tourné la poignée");
    } else if (etat == EPUISE) {
        System.out.println("Éjection impossible, vous n'avez pas inséré de pièce");
    }
}

public void tournerPoignee() {
    if (etat == VENDU) {
        System.out.println("Inutile de tourner deux fois !");
    } else if (etat == SANS_PIECE) {
        System.out.println("Vous avez tourné mais il n'y a pas de pièce");
    } else if (etat == EPUISE) {
        System.out.println("Vous avez tourné, mais il n'y a pas de bonbons");
    } else if (etat == A_PIECE) {
        System.out.println("Vous avez tourné...");
        etat = VENDU;
        delivrer();
    }
}

public void delivrer() {
    if (etat == VENDU) {
        System.out.println("Un bonbon va sortir");
        nombre = nombre - 1;
        if (nombre == 0) {
            System.out.println("Aïe, plus de bonbons !");
            etat = EPUISE;
        } else {
            etat = SANS_PIECE;
        }
    } else if (etat == SANS_PIECE) {
        System.out.println("Il faut payer d'abord");
    } else if (etat == EPUISE) {
        System.out.println("Pas de bonbon délivré");
    } else if (etat == A_PIECE) {
        System.out.println("Pas de bonbon délivré");
    }
}
// autres méthodes comme toString() et remplir()
}

```

Maintenant, si le client essaie de récupérer sa pièce...

S'il y a une pièce, nous la lui rendons et nous retournons à l'état SANS_PIECE.

Sinon, s'il n'y en a pas, nous ne pouvons pas la lui rendre.

Vous ne pouvez pas éjecter : puisqu'il n'y a pas de bonbons, la machine n'accepte pas de pièces !

Le client essaie de tourner la poignée...

Quelqu'un essaie de tricher

Si le client vient de tourner la poignée, nous ne pouvons pas le rembourser : il a déjà le bonbon !

Il faut d'abord une pièce

Nous ne pouvons pas délivrer de bonbons, il n'y en a pas.

Victoire ! Le client va avoir un bonbon. On passe à l'état VENDU et on appelle la méthode delivrer().

Appelée pour délivrer un bonbon

Nous sommes dans l'état VENDU : on lui donne le bonbon !

Voici où nous traitons la condition « plus de bonbons ». Si c'était le dernier, nous positionnons l'état de la machine à EPUISE ; sinon nous revenons à l'état SANS_PIECE.

Rien de ceci ne devrait arriver. Mais si c'est le cas, nous retournons une erreur, non un bonbon.

Test maison

Ne dirait-on pas une bonne petite conception bien solide s'appuyant sur une méthodologie bien pensée ? Faisons-lui subir un petit test maison avant de la livrer à Distribon pour qu'ils chargent le code dans leurs distributeurs. Voici le code :

```

public class TestDistributeur {
    public static void main(String[] args) {
        Distributeur distributeur = new Distributeur(5);

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.tournerPoignee();

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.ejecterPiece();
        distributeur.tournerPoignee();

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.tournerPoignee();
        distributeur.insererPiece();
        distributeur.tournerPoignee();
        distributeur.ejecterPiece();

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.insererPiece();
        distributeur.tournerPoignee();
        distributeur.insererPiece();
        distributeur.tournerPoignee();
        distributeur.insererPiece();
        distributeur.tournerPoignee();

        System.out.println(distributeur);
    }
}

```

Remplir avec cinq bonbons

Afficher l'état de la machine.

Insérer une pièce...

Tourner la poignée ; nous devrions recevoir notre bonbon.

Afficher de nouveau l'état de la machine

Insérer une pièce...

Demander à la récupérer..

Tourner la poignée. Nous ne devons pas avoir de bonbon..

Afficher de nouveau l'état de la machine.

Insérer une pièce....

Tourner la poignée; nous devrions recevoir notre bonbon

Insérer une pièce...

Tourner la poignée; nous devrions recevoir notre bonbon

Demander la pièce que nous n'avons pas insérée.

Afficher de nouveau l'état de la machine.

Insérer deux pièces...

Tourner la poignée; nous devrions recevoir notre bonbon.

Maintenant, le test de charge... ☹️

Afficher encore une fois l'état de la machine.

Fichier Fenêtre Édition Aide distribon.com

%java TestDistributeur

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock: 5 bonbons
L'appareil attend une pièce

Vous avez inséré une pièce
Vous avez tourné...
Un bonbon va sortir

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock: 4 bonbons
L'appareil attend une pièce

Vous avez inséré une pièce
Pièce retournée
Vous avez tourné mais il n'y a pas de pièce

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock: 4 bonbons
L'appareil attend une pièce

Vous avez inséré une pièce
Vous avez tourné...
Un bonbon va sortir
Vous avez inséré une pièce
Vous avez tourné...
Un bonbon va sortir
Vous n'avez pas inséré de pièce

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock: 2 bonbons
L'appareil attend une pièce

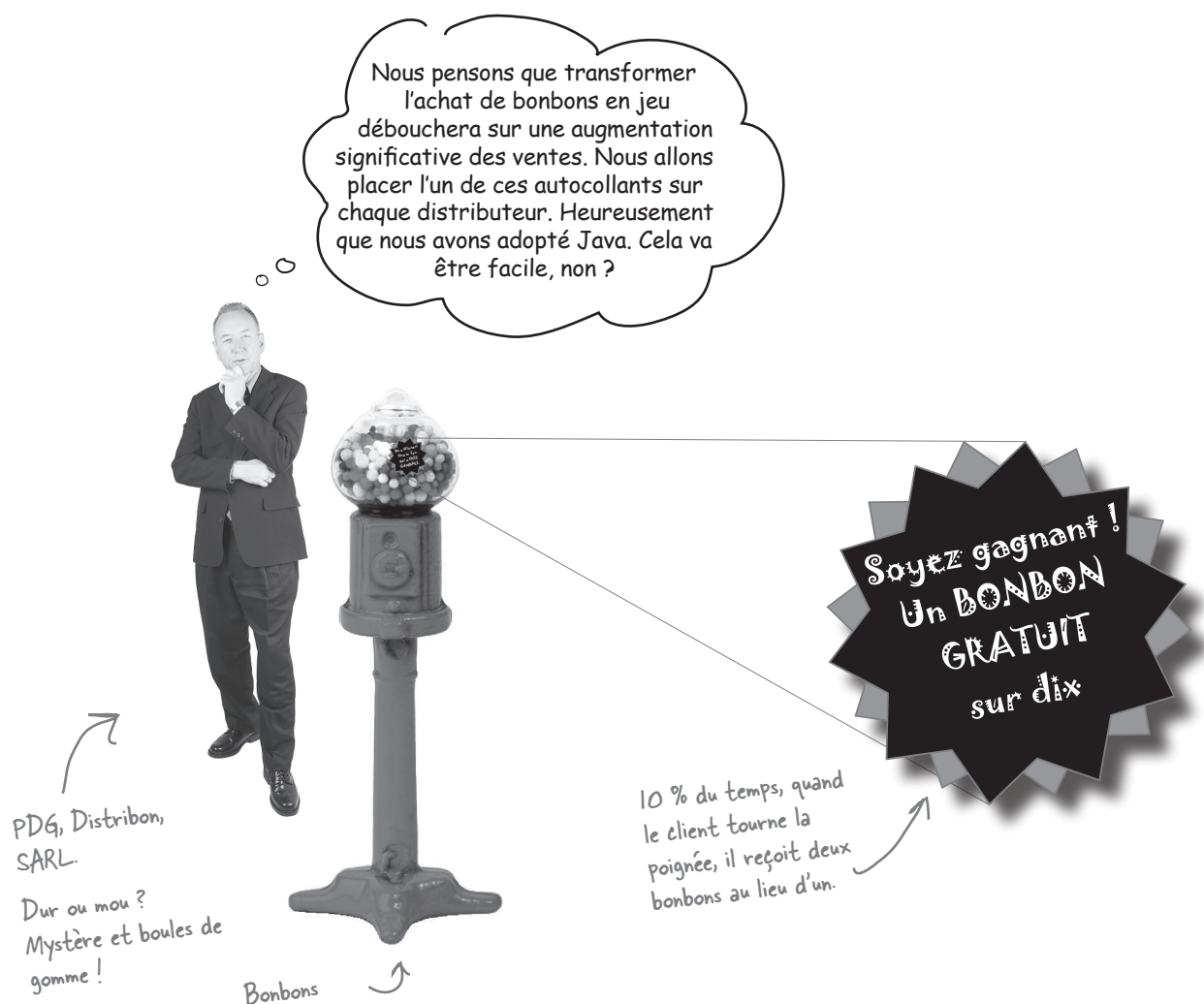
Vous avez inséré une pièce
Vous ne pouvez plus insérer de pièces
Vous avez tourné...
Un bonbon va sortir
Vous avez inséré une pièce
Vous avez tourné...
Un bonbon va sortir
Aïe, plus de bonbons !
Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock
Vous avez tourné, mais il n'y a pas de bonbons

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock: 0 bonbons
L'appareil est en rupture de stock

Vous vous y attendiez... une demande de changement !

Distribon, SARL a chargé votre code dans leur distributeur le plus récent, et leurs experts en assurance qualité sont en train de le mettre à l'épreuve. Jusque là, tout est parfait de leur point de vue.

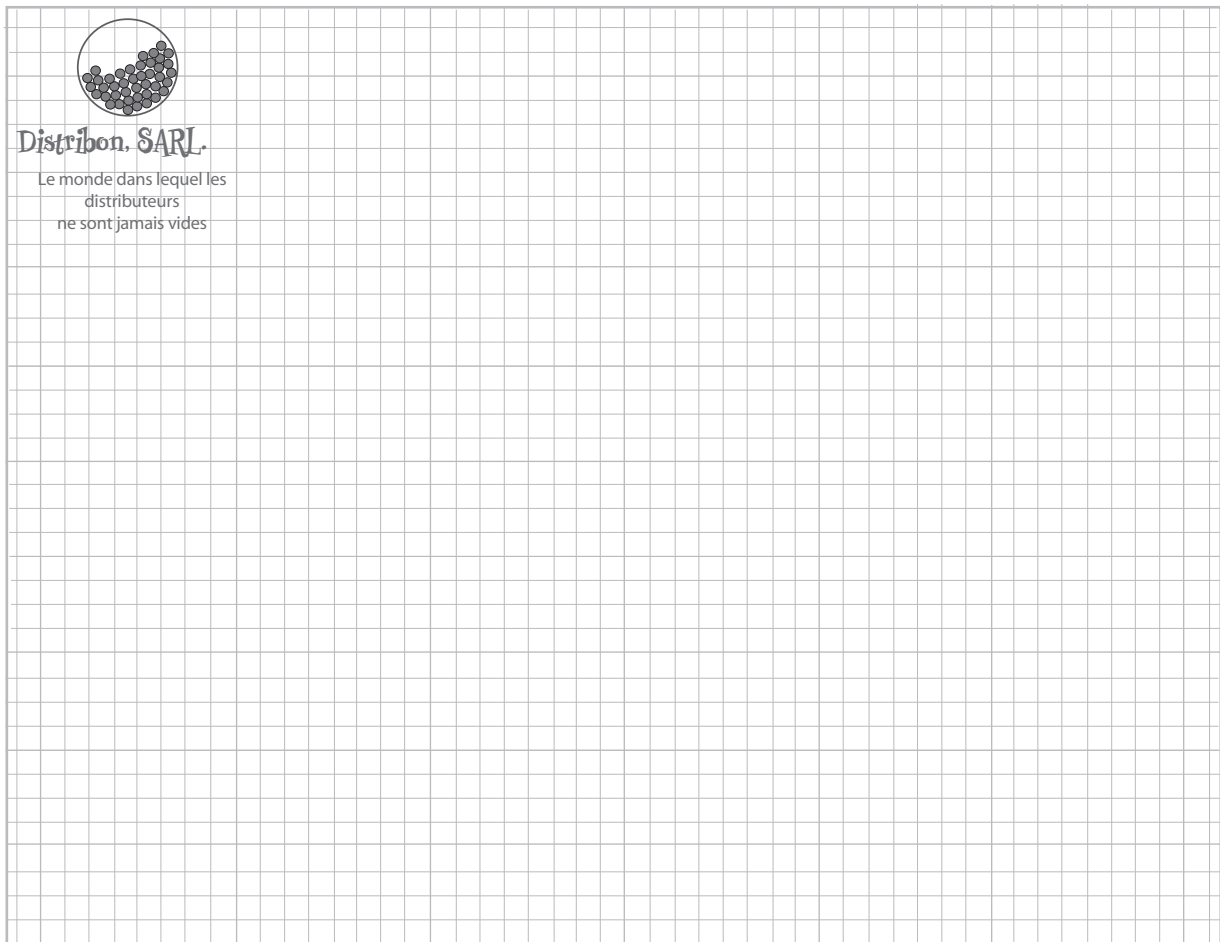
En fait, tout est tellement parfait qu'ils veulent passer à la vitesse supérieure...





Problème de conception

Tracez un diagramme d'états du Distributeur qui gère le concours.
Dans ce concours, 10 % du temps, l'état Vendu conduit à libérer deux bonbons au lieu d'un. Comparez votre réponse à la nôtre (en fin de chapitre) pour être sûr que nous sommes d'accord avant de poursuivre...



Utilisez le papier à en-tête de Distribon pour tracer votre diagramme d'états

Nous sommes dans un drôle d'état

Ce n'est pas parce que vous avez écrit votre code en vous appuyant sur une méthodologie bien pensée qu'il est extensible pour autant. En fait, si l'on revient en arrière et qu'on regarde le nombre de modifications nécessaires, eh bien...

```
final static int EPUISE = 0;
final static int SANS_PIECE = 1;
final static int A_PIECE = 2;
final static int VENDU = 3;

public void insererPiece() {
    // code pour insérer une pièce
}

public void ejecterPiece() {
    // code pour éjecter la pièce
}

public void tournerPoignee() {
    // code pour tourner la poignée
}

public void delivrer() {
    // code pour libérer le bonbon
}
```

D'abord, il faudrait ajouter ici un état **GAGNANT**.
Pas impossible...

.. mais alors, il faudrait ajouter de nouvelles instructions conditionnelles dans chaque méthode pour gérer l'état **GAGNANT**. Cela fait beaucoup de code à modifier.

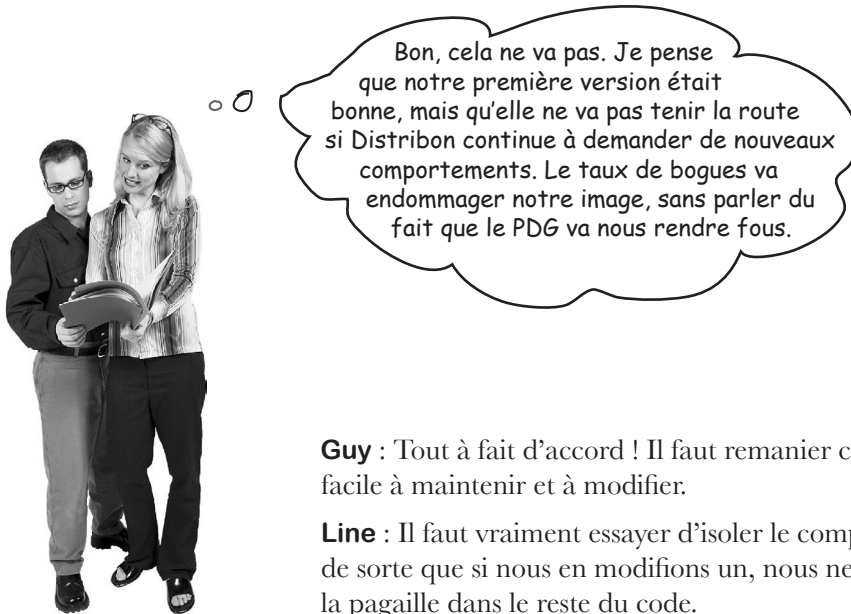
tournerPoignee() va être particulièrement délicate, car vous allez devoir ajouter du code pour tester si vous avez un **GAGNANT**, puis passer soit à l'état **GAGNANT** soit à l'état **VENDU**.



À vos crayons

Dites si les affirmations suivantes décrivent l'état de notre implémentation. (Plusieurs réponses possibles.)

- | | |
|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <input type="checkbox"/> A. Ce code n'adhère pas au principe Ouvert-Fermé. | <input type="checkbox"/> D. Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles. |
| <input type="checkbox"/> B. Ce code ferait la fierté d'un programmeur FORTRAN. | <input type="checkbox"/> E. Nous n'avons pas encapsulé ce qui varie. |
| <input type="checkbox"/> C. Cette conception n'est pas orientée objet. | <input type="checkbox"/> F. Les ajouts ultérieurs sont susceptibles de provoquer des bogues dans le code. |



Guy : Tout à fait d'accord ! Il faut remanier ce code pour qu'il soit facile à maintenir et à modifier.

Line : Il faut vraiment essayer d'isoler le comportement de chaque état, de sorte que si nous en modifions un, nous ne risquons pas de mettre la pagaille dans le reste du code.

Guy : Bien, autrement dit appliquer le bon vieux principe « encapsuler ce qui varie ».

Line : Exactement.

Guy : Si nous plaçons le comportement de chaque état dans sa propre classe, chaque état n'implémentera que ses propres actions.

Line : Tout à fait. Et peut-être que le Distributeur peut simplement déléguer à l'objet état qui représente l'état courant.

Guy : Ah, bravo : préférer la composition... encore un bon principe à l'œuvre.

Line : Oui. Eh bien, je ne suis pas sûre à 100 % de la façon dont cela va fonctionner, mais je crois qu'on est sur la bonne piste.

Guy : Je me demande si cela facilitera l'ajout de nouveaux états.

Line : Je pense que oui... Nous devons toujours modifier le code, mais la portée des changements sera beaucoup plus restreinte, parce que l'ajout d'un nouvel état se limitera à l'insertion d'une nouvelle classe et peut-être à la modification de quelques transitions ici et là.

Guy : Voilà qui me plaît bien. Attaquons-nous à cette nouvelle conception !

La nouvelle conception

On dirait que nous avons un nouveau plan : au lieu de maintenir notre code existant, nous allons le retravailler pour encapsuler les objets état dans leur propre classe, puis déléguer à l'état courant quand une action a lieu.

Comme nous appliquons nos principes de conception, nous devrions aboutir à une solution plus facile à maintenir en aval. Voici comment nous allons procéder :

- 1 Tout d'abord, nous allons définir une interface Etat qui contiendra une méthode pour chaque action liée au Distributeur.**
- 2 Ensuite, nous allons implémenter une classe pour chaque état de la machine. Ces classes seront responsables du comportement du distributeur quand il se trouvera dans l'état correspondant.**
- 3 Enfin, nous allons nous débarrasser de toutes nos instructions conditionnelles et les remplacer par une délégation à la classe état qui travaillera à notre place.**

Non seulement nous appliquons les principes de conception, mais, comme vous allez le voir, nous implémentons de fait le pattern État. Mais nous ne nous occuperons pas de la définition officielle du pattern avant d'avoir retravaillé notre code...



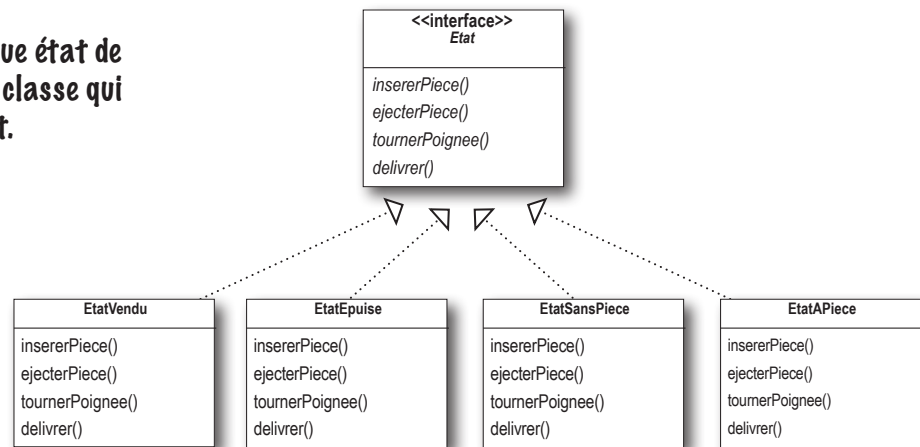
Définir l'interface et les classes pour les états

Créons d'abord une interface Etat que tous nos états implémenteront :

Voici l'interface pour tous les états. Les méthodes correspondent directement aux actions qui pourraient être celles du distributeur (ce sont les mêmes méthodes que dans le code précédent).

Puis nous encapsulons chaque état de notre conception dans une classe qui implémente l'interface Etat.

Pour savoir quels états il nous faut, nous nous reportons au code précédent...



```

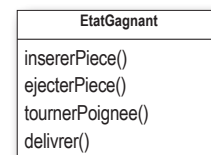
public class Distributeur {

    final static int VENDU = 3;
    final static int EPUISE = 0;
    final static int SANS_PIECE = 1;
    final static int A_PIECE = 2;

    int etat = EPUISE;
    int nombre = 0;
}
  
```

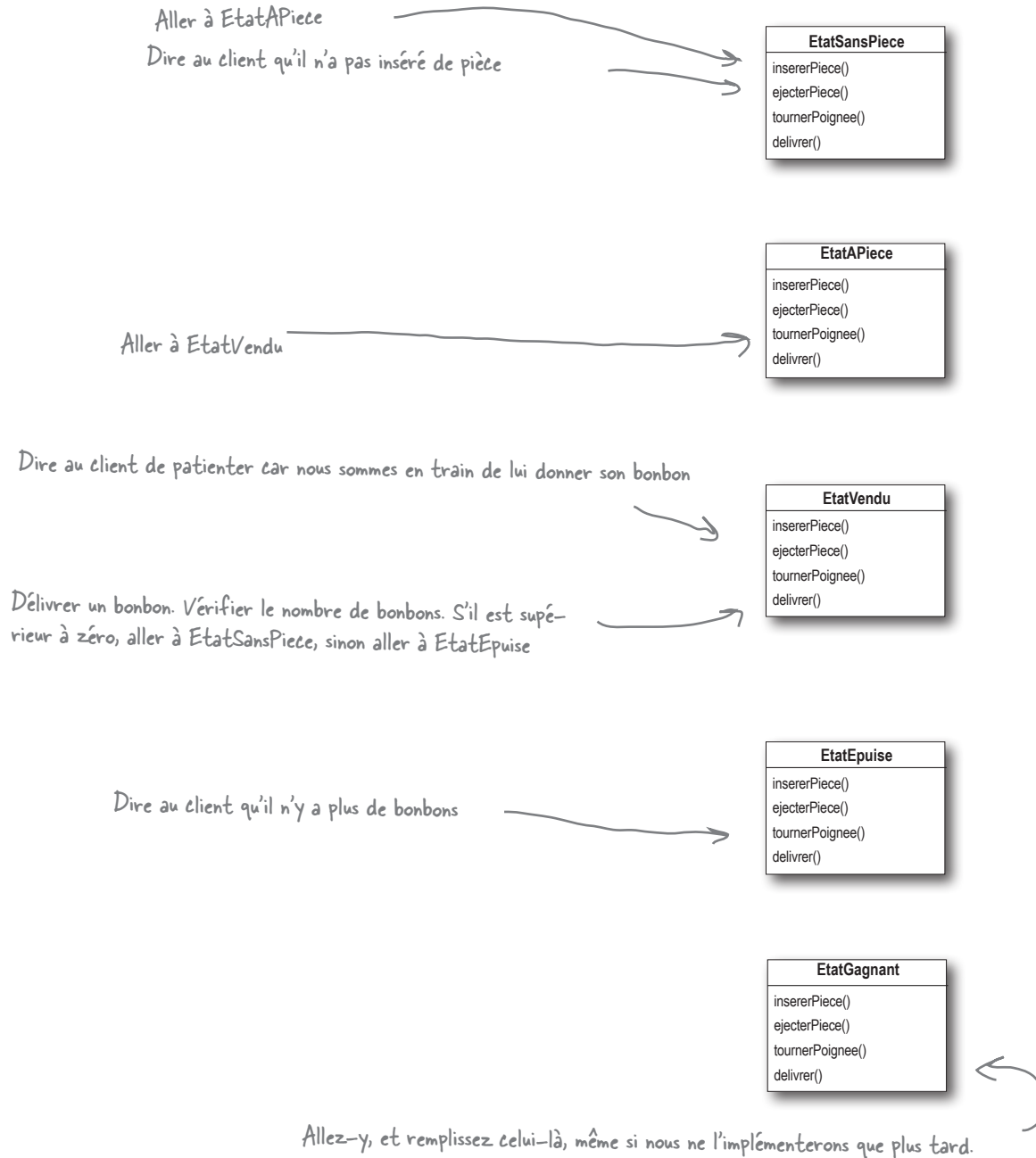
... et nous faisons correspondre chaque état directement à une classe.

N'oubliez pas qu'il va nous falloir aussi un état << gagnant >> qui implémente également l'interface Etat. Nous y reviendrons après avoir réimplémenté la première version du Distributeur.



À vos crayons

Pour implémenter nos états, nous devons d'abord spécifier le comportement des classes quand chaque action est appelée. Annotez le diagramme ci-dessous en inscrivant le comportement de chaque action dans chaque classe. Nous en avons déjà indiqué quelques-uns pour vous.



Implémenter les classes pour les états

Il est temps d'implémenter un état. Nous connaissons les comportements nécessaires et il faut maintenant les traduire en code. Nous allons nous calquer étroitement sur le code de la machine à états que nous avons déjà écrit, mais, cette fois, tout est réparti dans des classes différentes.

Commençons par EtatSansPiece :

Nous devons d'abord implémenter l'interface Etat.

```
public class EtatSansPiece implements Etat {
    Distributeur distributeur;

    public EtatSansPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        distributeur.setEtat(distributeur.getEtatAPiece());
    }

    public void ejecterPiece() {
        System.out.println("Vous n'avez pas inséré de pièce");
    }

    public void tournerPoignee() {
        System.out.println("Vous avez tourné, mais il n'y a pas de pièce");
    }

    public void delivrer() {
        System.out.println("Il faut payer d'abord");
    }
}
```

On transmet une référence au Distributeur dans le constructeur et on la place dans une variable d'instance.

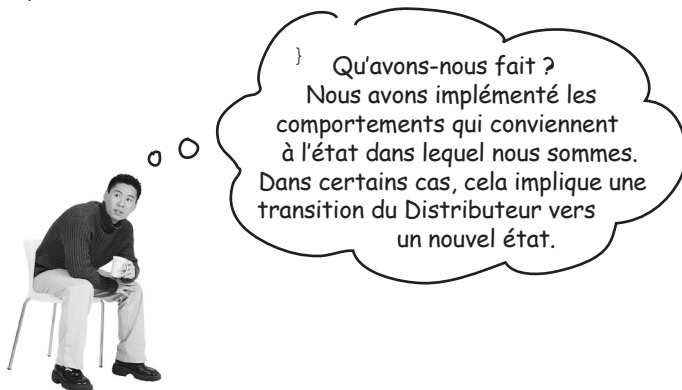
Si quelqu'un insère une pièce, nous affichons un message indiquant qu'elle a été acceptée et nous passons dans l'état EtatAPiece.

Vous allez voir dans une seconde comment ceci fonctionne...

Vous ne pouvez pas reprendre votre pièce : vous ne nous l'avez jamais donnée !

Et si vous ne payez pas, vous ne pouvez pas avoir de bonbon.

Impossible de donner gratuitement des bonbons..



Retravaillons le distributeur

Avant de terminer les classes État, nous allons retravailler le Distributeur pour que vous voyiez comment tout cela s'articule. Nous commencerons par les variables d'instance et nous abandonnerons les entiers pour adopter des objets état :

```
public class Distributeur {  
  
    final static int EPUISE = 0;  
    final static int SANS_PIECE = 1;  
    final static int A_PIECE = 2;  
    final static int VENDU = 3;  
  
    int etat = EPUISE;  
    int nombre = 0;  
}
```

Ancien code

Nous actualisons la classe Distributeur, afin d'utiliser les nouvelles classes à la place des entiers statiques. Le code est très similaire à une exception près : dans un cas nous avons des entiers, dans l'autre des objets...

```
public class Distributeur {  
  
    Etat etatEpuise;  
    Etat etatSansPiece;  
    Etat etatAPiece;  
    Etat etatVendu;  
  
    Etat etat = etatEpuise;  
    int nombre = 0; }  
}
```

Nouveau code

Tous les objets Etat sont créés et affectés dans le constructeur.

Cette variable ne contient plus un entier, mais un objet Etat.

Voyons maintenant la classe Distributeur terminée...

```
public class Distributeur {
```

```
    Etat etatEpuise;
    Etat etatSansPiece;
    Etat etatAPiece;
    Etat etatVendu;
    Etat etat = etatEpuise;
    int nombre = 0;
    public Distributeur(int nombreBonbons) {
        etatEpuise = new EtatEpuise(this);
        etatSansPiece = new EtatSansPiece(this);
        etatAPiece = new EtatAPiece(this);
        etatVendu = new EtatVendu(this);
        this.nombre = nombreBonbons;
        if (nombreBonbons > 0) {
            etat = etatSansPiece;
        }
    }
```

```
    public void insererPiece() {
        etat.insererPiece();
    }
```

```
    public void ejecterPiece() {
        etat.ejecterPiece();
    }
```

```
    public void tournerPoignee() {
        etat.tournerPoignee();
        etat.delivrer();
    }
```

```
    void setEtat(Etat etat) {
        this.etat = etat;
    }
```

```
    void liberer() {
        System.out.println("Un bonbon va sortir...");
        if (nombre != 0) {
            nombre = nombre - 1;
        }
    }
```

```
    // Autres méthodes, dont une méthode get pour chaque état...
}
```

Voici de nouveau les États...

...et la variable d'instance etat.

La variable d'instance nombre contient le nombre de bonbons – initialement, le distributeur est vide.

Notre constructeur accepte le nombre de bonbons initial et le stocke dans une variable d'instance.

Il crée également les instances des états, une pour chacun.

S'il y a plus de 0 bonbons, nous positionnons l'état à EtatSansPiece.

Au tour des actions. Maintenant, elles sont TRÈS FACILES à implémenter. Il suffit de déléguer à l'état courant.

Notez que nous n'avons pas besoin de méthode d'action pour delivrer() dans la classe Distributeur parce que ce n'est qu'une action interne : l'utilisateur ne peut pas demander directement à la machine de délivrer. Mais nous appelons bien delivrer() sur l'objet état depuis la méthode tournerPoignee().

Cette méthode permet à d'autres objets (comme nos objets état) de déclencher une transition de la machine vers un autre état.

La machine prend en charge une méthode auxiliaire, liberer(), qui libère le bonbon et décrémente variable d'instance nombre.

Par exemple des méthodes comme getEtatSansPiece() pour obtenir chaque objet état et getNombre() pour obtenir le nombre de bonbons.

Implémenter d'autres états

Maintenant que vous commencez à voir comment le Distributeur et les états s'articulent, implémentons les classes EtatAPiece et EtatVendu...

```
public class EtatAPiece implements Etat {
    Distributeur distributeur;

    public EtatAPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous ne pouvez pas insérer d'autre pièce");
    }

    public void ejecterPiece() {
        System.out.println("Pièce retournée");
        distributeur.setEtat(distributeur.getEtatSansPiece());
    }

    public void tournerPoignee() {
        System.out.println("Vous avez tourné...");
        distributeur.setEtat(distributeur.getEtatVendu());
    }

    public void delivrer() {
        System.out.println("Pas de bonbon délivré");
    }
}
```

Quand l'état est instancié, nous lui transmettons une référence au Distributeur. Cela sert à déclencher la transition de la machine vers un autre état.

Une action inappropriée pour cet état.

On rend sa pièce au client et on retourne à EtatSansPiece.

Quand la poignée a été tournée, nous déclenchons la transition vers EtatVendu en appelant la méthode setEtat() et en lui transmettant l'objet EtatVendu. L'objet EtatVendu est extrait par la méthode getEtatVendu() (il y a une de ces méthodes get pour chaque état).

Autre action inappropriée pour cet état.

Voyons maintenant la classe EtatVendu...

```
public class EtatVendu implements Etat {
    //constructeur et variables d'instance

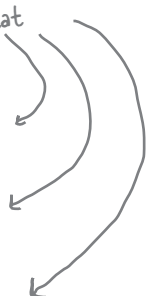
    public void insererPiece() {
        System.out.println("Veuillez patienter, le bonbon va tomber");
    }

    public void ejecterPiece() {
        System.out.println("Vous avez déjà tourné la poignée");
    }

    public void tournerPoignee() {
        System.out.println("Inutile de tourner deux fois !");
    }

    public void delivrer() {
        distributeur.liberer();
        if (distributeur.getNombre() > 0) {
            distributeur.setEtat(distributeur.getEtatSansPiece());
        } else {
            System.out.println("Aïe, plus de bonbons !");
            distributeur.setEtat(distributeur.getEtatEpuise());
        }
    }
}
```

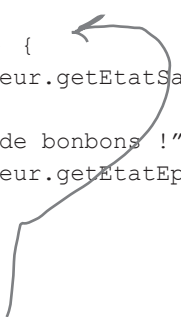
Voici toutes
les actions
inappropriées pour
cet état




Et voilà où le vrai
travail commence...



Nous sommes dans EtatVendu, ce
qui veut dire que le client a payé.
Nous devons donc dire à la machine
de libérer un bonbon.



Puis nous lui demandons quel est
le nombre de bonbons, et nous
passons soit à EtatSansPiece soit à
EtatEpuise.




MUSCLEZ VOS NEURONES

Regardez de nouveau l'implémentation du Distributeur. Si la poignée a été tournée et que l'opération échoue (par exemple parce que le client n'a pas inséré de pièce), nous appelons quand même delivrer() alors que c'est inutile. Comment corrigeriez-vous cela ?

À vos crayons

Il reste une classe que nous n'avons pas implémentée : EtatEpuise. Pourquoi ne le feriez-vous pas ? Pour ce faire, réfléchissez soigneusement à la façon dont le Distributeur doit se comporter dans chaque situation. Vérifiez votre réponse au lieu de continuer...

```
public class EtatEpuise implements  {
    Distributeur distributeur;

    public EtatEpuise(Distributeur distributeur) {

    }

    public void insererPiece() {

    }

    public void ejecterPiece() {

    }

    public void tournerPoignee() {

    }

    public void delivrer() {

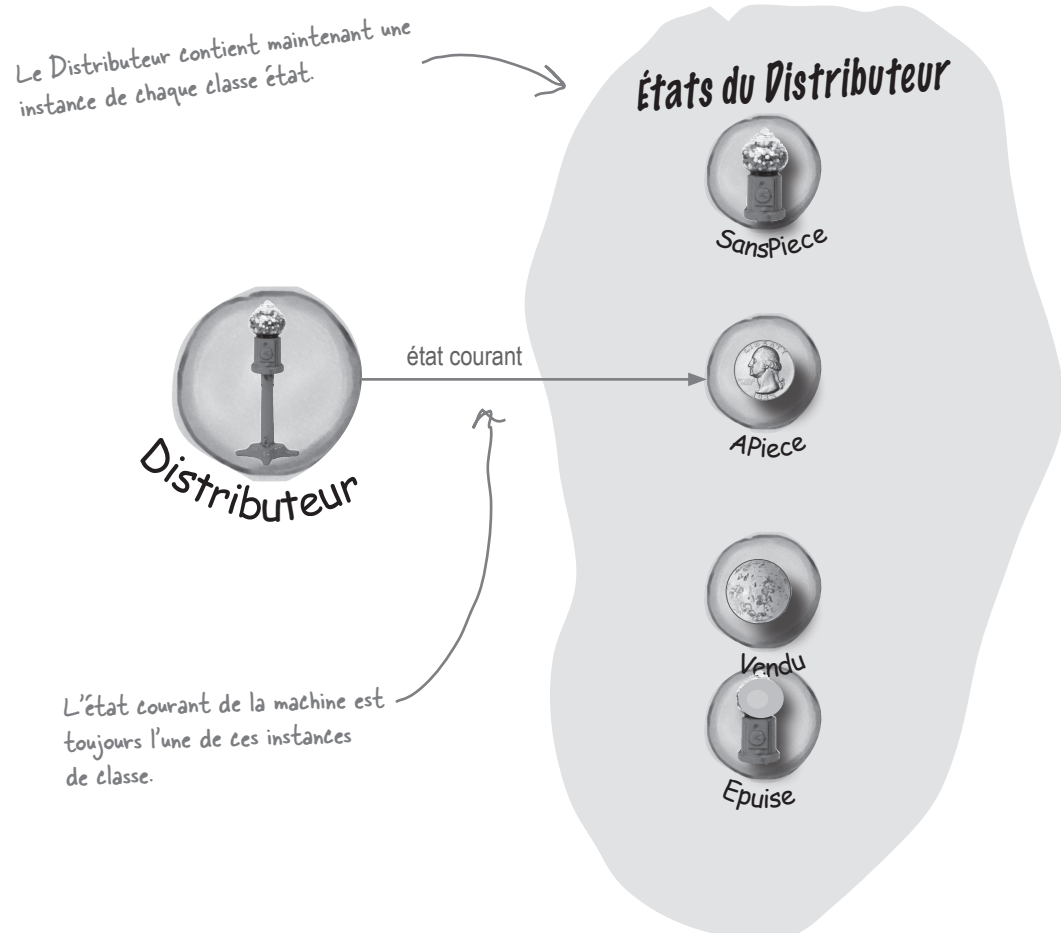
    }
}
```

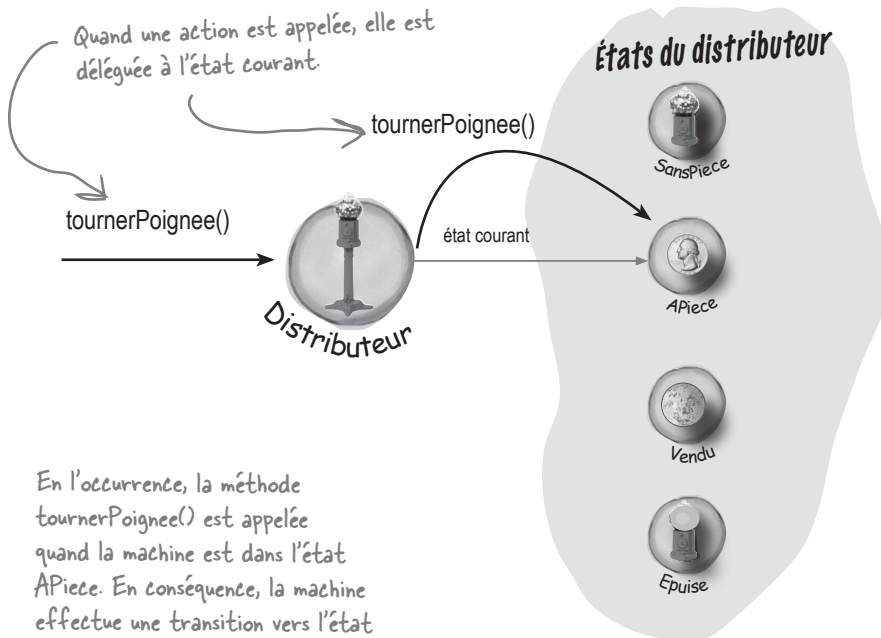

Qu'avons-nous fait jusqu'ici ?

Pour commencer, nous avons maintenant une implémentation du Distributeur qui diffère beaucoup de la première version du point de vue *structurel*, tout en étant *fonctionnellement* identique. En modifiant la structure de cette implémentation, vous avez :

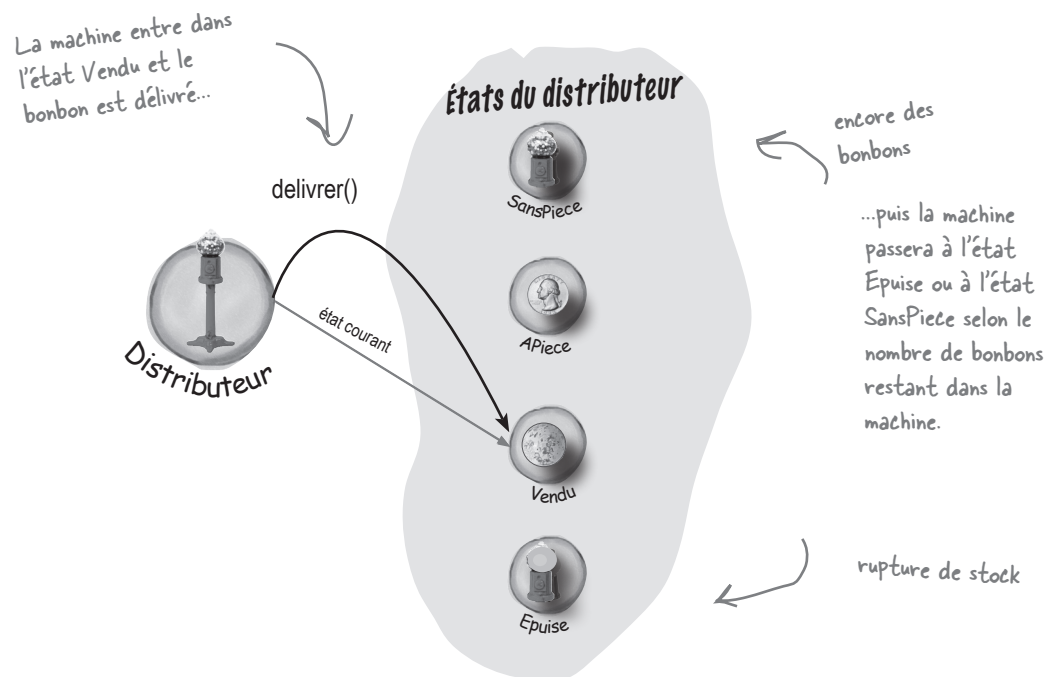
- isolé le comportement de chaque état dans sa propre classe ;
- supprimé toutes ces fâcheuses instructions conditionnelles qui auraient été difficiles à maintenir ;
- fermé chaque état à la modification et laissé le Distributeur ouvert à l'extension en ajoutant de nouvelles classes état (nous allons le faire dans une seconde) ;
- créé une base de code et une structure de classes qui correspondent plus étroitement au diagramme de Distribon, et qui sont plus faciles à lire et à comprendre.

Regardons maintenant d'un peu plus près l'aspect fonctionnel :





TRANSITION VERS L'ÉTAT VENDU





Dans les coulisses : Visite libre



Tracez les étapes du Distributeur en commençant par l'état SansPiece. Annotez également le diagramme en indiquant les actions et les résultats. Pour cet exercice, vous pouvez supposer qu'il y a plein de bonbons dans le distributeur.

①



états du distributeur



②



états du distributeur



③



états du distributeur



④



états du distributeur



Le pattern État : définition

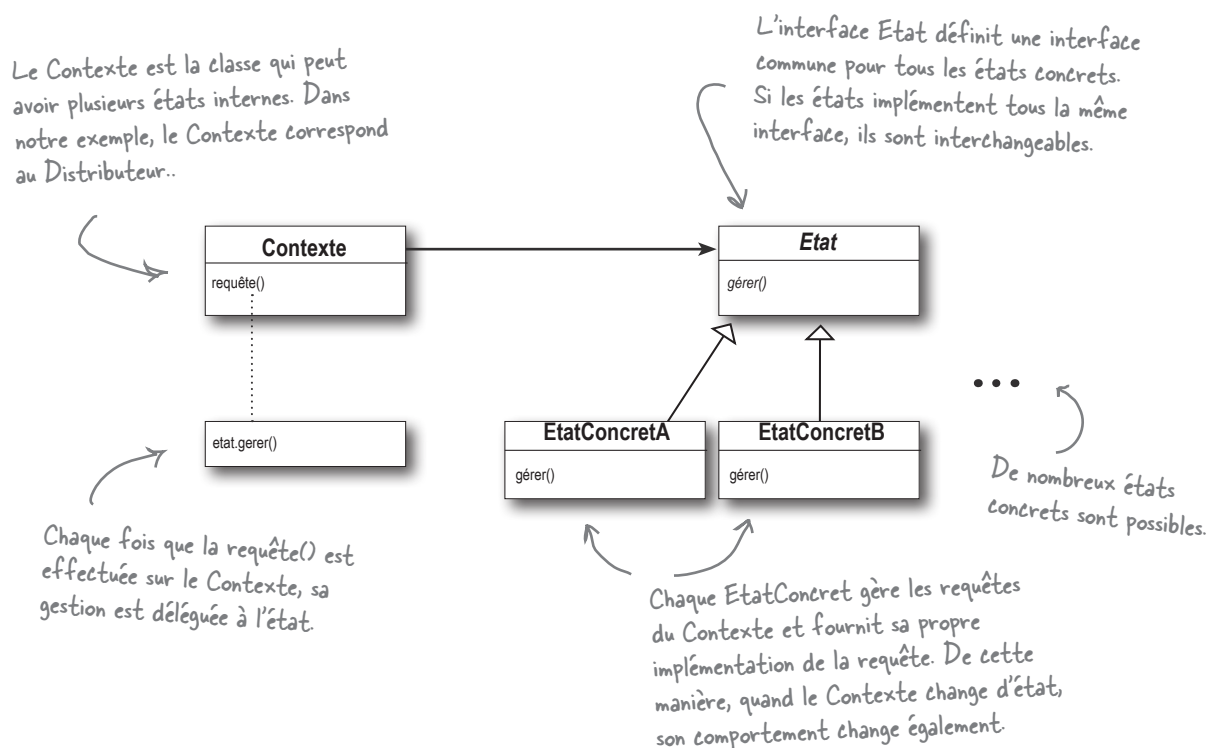
Oui, c'est la vérité : nous venons d'implémenter le pattern État ! Voyons maintenant de quoi il s'agit :

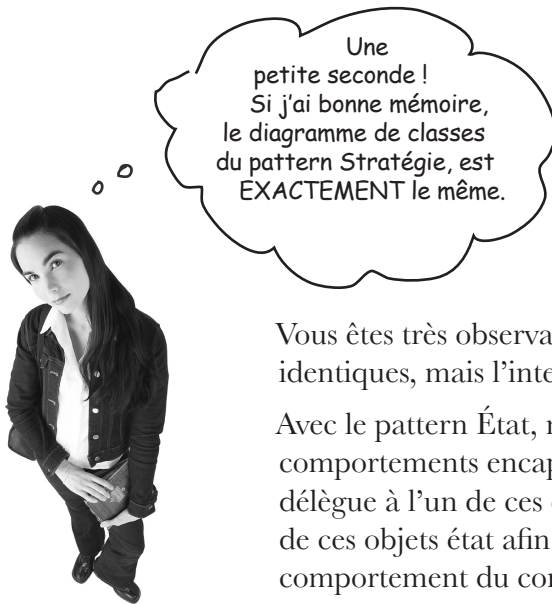
Le pattern État permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.

La première partie de la description est limpide, non ? Puisque le pattern encapsule l'état dans des classes séparées et délègue à l'objet représentant l'état courant, nous savons que le comportement change avec l'état interne. Le Distributeur en est un bon exemple : quand la machine est dans l'EtatSansPiece et que vous insérez une pièce, vous obtenez un comportement différent (elle accepte la pièce) que si elle avait été dans l'EtatAPiece (la machine rejette la pièce).

Mais qu'en est-il de la seconde partie de la définition ? Que veut dire « tout se passera comme si l'objet changeait de classe » ? Représentez-vous le point de vue d'un client : si un objet que vous utilisez peut changer complètement de comportement, il vous semble alors instancié à partir d'une autre classe. Mais, en réalité, vous savez que nous utilisons une composition pour donner l'apparence d'un changement de classe en référençant simplement des objets état différents.

Bien. Il est maintenant temps d'étudier le diagramme de classes du pattern État :





Vous êtes très observateur ! Oui, les diagrammes de classes sont globalement identiques, mais l'intention des deux patterns est différente.

Avec le pattern État, nous sommes en présence d'un ensemble de comportements encapsulés dans des objets état. À tout moment, le contexte délègue à l'un de ces états. Au fil du temps, l'état courant change pour l'un de ces objets état afin de refléter l'état interne du contexte, si bien que le comportement du contexte change également. Le client ne sait généralement pas grand-chose, voire rien, des objets état.

Avec Stratégie, le client spécifie généralement l'objet stratégie avec lequel le contexte est composé. Maintenant, si le pattern permet de changer l'objet stratégie au moment de l'exécution, il y a souvent un objet stratégie qui est particulièrement adapté à un objet contexte. Par exemple, au chapitre 1, certains de nos canards étaient configurés pour voler avec un comportement de vol classique (comme les colverts), tandis que d'autres étaient configurés avec un comportement de vol qui les clouait au sol (comme les canards en plastique et les leurres).

En général, vous pouvez voir le pattern Stratégie comme une solution de rechange souple au sous-classement. Si vous utilisez l'héritage pour définir le comportement d'une classe, vous êtes coincé si vous avez besoin de le modifier. Avec Stratégie, vous pouvez changer le comportement en composant avec un objet différent.

Représentez-vous le pattern État comme une solution qui permet d'éviter de placer une foule d'instructions conditionnelles dans votre contexte. En encapsulant les comportements dans des objets état, il suffit de modifier ces derniers dans le contexte pour changer son comportement.

il n'y a pas de questions stupides

Q: Dans le Distributeur, ce sont les états qui décident de l'état dans lequel ils doivent être ensuite. Est-ce que c'est toujours un EtatConcret qui décide des transitions ?

R: Non, pas toujours. L'autre solution consiste à laisser le Contexte décider du flux des transitions. En règle générale, lorsque les transitions sont fixes, il est judicieux de les placer dans le Contexte. En revanche, si les transitions sont plus dynamiques, on les place classiquement dans les classes état elles-mêmes. Par exemple, dans le Distributeur, le choix de la transition vers SansPiece ou vers Epuise dépend du nombre de bonbons au moment de l'exécution.

Le fait de placer des transitions dans les classes état présente l'inconvénient de créer des dépendances entre ces classes. Dans notre implémentation du Distributeur, nous avons tenté de le réduire en utilisant des méthodes get sur le Contexte au lieu de coder en dur des classes état concrètes explicites.

Remarquez que, en prenant cette décision, vous décidez également quelles sont les classes fermées à la modification – le Contexte ou les classes état – à mesure que le système évolue.

Q: Est-ce qu'il arrive que les clients interagissent directement avec les états ?

R: Non. Comme le Contexte utilise les états pour représenter son état interne et son comportement, toutes les requêtes aux états proviennent du Contexte. Les clients ne modifient pas directement l'état du Contexte. Il appartient au Contexte de surveiller son état, et l'on ne veut généralement pas qu'un client modifie l'état d'un Contexte sans que celui-ci ne le sache.

Q: Si j'ai beaucoup d'instances du Contexte dans mon application, est-il possible de partager les objets état entre eux ?

R: Oui, absolument. En fait, c'est un scénario très courant. La seule exigence est que vos objets état ne mémorisent pas leur propre état interne. Sinon, il vous faut une instance unique par contexte.

Pour partager vos états, vous affecterez généralement chaque état à une variable d'instance statique. Si un état a besoin d'utiliser des méthodes ou des variables de votre Contexte, vous devez également lui fournir une référence au contexte dans chaque méthode gerer().

Q: On dirait que l'emploi du pattern État augmente toujours le nombre de classes dans nos conceptions. Regardez combien notre Distributeur a de classes de plus que la version d'origine !

R: Vous avez raison. En encapsulant les comportements dans des classes état distinctes, on finit toujours par avoir plus de classes. C'est souvent le prix à payer pour conserver de la souplesse. À moins que votre code ne soit une implémentation « provisoire » que vous jetterez ensuite, envisagez de le construire avec des classes supplémentaires et vous vous en félicitez probablement en aval. Notez que le point important est souvent le nombre de classes que vous exposez à vos clients, et qu'il existe des moyens pour leur cacher ces classes supplémentaires (par exemple en déclarant une visibilité package).

Mais réfléchissez. Si votre application a beaucoup d'états et que vous décidez de ne pas utiliser des objets séparés, vous allez vous retrouver avec d'énormes instructions conditionnelles monolithiques, et votre code sera plus difficile à maintenir et à comprendre. En utilisant des objets, vous rendez les états explicites et votre code est plus lisible et plus facile à maintenir.

Q: Le diagramme de classes du pattern État représente l'état comme une classe abstraite. Mais vous avez utilisé une interface dans l'implémentation de l'état du distributeur...

R: Oui. Étant donné que nous n'avions aucune fonctionnalité commune à placer dans une classe abstraite, nous avons choisi une interface. Dans votre propre implémentation, vous préférez peut-être une classe abstraite. Cette technique a l'avantage de vous permettre d'ajouter des méthodes à la classe abstraite plus tard, sans toucher aux implémentations des états concrets.

Terminons le jeu

Souvenez-vous, nous n'avons pas encore fini. Il nous reste le jeu à programmer. Mais, maintenant que nous avons implémenté le pattern État, nous allons le faire en un clin d'œil. Tout d'abord, nous devons ajouter un état à la classe Distributeur :

```
public class Distributeur {
```

```
    Etat etatEpuise;
    Etat etatSansPiece;
    Etat etatAPiece;
    Etat etatVendu;
    Etat etatGagnant;
```

Il suffit d'ajouter ici le nouvel EtatGagnant et de l'initialiser dans le constructeur

```
    Etat etat = etatEpuise;
    int nombre = 0;
```

N'oubliez pas qu'il faudra également ajouter ici une méthode getEtatGagnant.

```
    // méthodes
```

```
}
```

Implémentons maintenant la classe EtatGagnant elle-même. Elle est remarquablement similaire à la classe EtatVendu :

```
public class EtatGagnant implements Etat {
```

```
    // variables d'instance et constructeur
```

```
    // message d'erreur insererPiece
```

```
    // message d'erreur ejecterPiece
```

```
    // message d'erreur tournerPoignee
```

Exactement comme dans EtatVendu.

Ici, nous en libérons deux et nous passons à EtatSansPiece ou à EtatEpuise.

```
    public void delivrer() {
```

```
        System.out.println("VOUS AVEZ GAGNÉ ! Deux bonbons pour le prix d'un !");
```

```
        distributeur.liberer();
```

```
        if (distributeur.getNombre() == 0) {
```

```
            distributeur.setEtat(distributeur.getEtatEpuise());
```

```
        } else {
```

```
            distributeur.liberer();
```

```
        if (distributeur.getNombre() > 0) {
```

```
            distributeur.setEtat(distributeur.getEtatSansPiece());
```

```
        } else {
```

```
            System.out.println("Aïe, plus de bonbons !");
```

```
            distributeur.setEtat(distributeur.getEtatEpuise());
```

```
        }
```

```
    }
```

```
}
```

```
}
```

Tant qu'il reste un second bonbon, nous le libérons.

Terminons le jeu

Il reste une dernière modification à apporter : nous devons implémenter le jeu de hasard et ajouter une transition vers `EtatGagnant`. Nous allons ajouter les deux à `EtatAPiece`, puisque c'est là où le client tourne la poignée :

```
public class EtatAPiece implements Etat {
    Random hasard = new Random(System.currentTimeMillis());
    Distributeur distributeur;

    public EtatAPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous ne pouvez pas insérer d'autre pièce");
    }

    public void ejecterPiece() {
        System.out.println("Pièce retournée");
        distributeur.setEtat(distributeur.getEtatSansPiece());
    }

    public void tournerPoignee() {
        System.out.println("Vous avez tourné...");
        int gagnant = hasard.nextInt(10);
        if ((gagnant == 0) && (distributeur.getNombre() > 1)) {
            distributeur.setEtat(distributeur.getEtatGagnant());
        } else {
            distributeur.setEtat(distributeur.getEtatVendu());
        }
    }

    public void delivrer() {
        System.out.println("Pas de bonbon délivré");
    }
}
```

Nous ajoutons d'abord un générateur de nombres aléatoires pour générer les dix pour cent de chance de gagner...

...puis nous déterminons si le client a gagné.

S'il a gagné et qu'il reste assez de bonbons pour qu'il en ait deux, nous passons à `EtatGagnant`. Sinon, nous passons à `EtatVendu` (comme nous l'avons toujours fait).

Eh bien, il n'y avait pas plus simple à écrire ! Nous avons simplement ajouté un nouvel état au `Distributeur` puis nous l'avons implémenté. À partir de là, il nous a suffi d'implémenter le jeu de hasard et la transition vers l'état correct. On dirait que notre nouvelle stratégie de codage porte ses fruits...

Démonstration au PDG de Distribon, SARL

Le PDG de Distribon est passé voir une démo de notre nouveau code. Espérons que ces états sont tous en ordre ! La démonstration sera courte (la faible capacité d'attention du PDG est bien connue), mais suffisamment longue pour qu'on puisse espérer gagner au moins une fois.

Non, le code n'a pas vraiment changé. Nous l'avons simplement raccourci un peu.

Cette fois encore, nous commençons avec cinq bonbons.

```
public class TestDistributeur {
    public static void main(String[] args) {
        Distributeur distributeur = new Distributeur(5);

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.tournerPoignee();

        System.out.println(distributeur);

        distributeur.insererPiece();
        distributeur.tournerPoignee();
        distributeur.insererPiece();
        distributeur.tournerPoignee();

        System.out.println(distributeur);
    }
}
```

Comme nous voulons un état gagnant, nous n'arrêtons pas d'investir des pièces et de tourner la poignée. De temps à autre, nous affichons l'état du distributeur....

L'équipe de développement au grand complet derrière la porte de la salle de conférences, attendant de voir si la nouvelle conception basée sur le pattern État va fonctionner !!





Ça alors !, On a eu de la chance ou quoi ? Dans notre démo au PDG, nous n'avons pas gagné une fois mais deux !

Fichier Fenêtre Édition Aide AnnieAime LesSucettes

```
%java TestDistributeur
Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock : 5 bonbons
L'appareil attend une pièce

Vous avez inséré une pièce
Vous avez tourné...
VOUS AVEZ GAGNÉ ! Deux bonbons pour le prix d'un !
Un bonbon va sortir...
Un bonbon va sortir...

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock : 3 bonbons
L'appareil attend une pièce

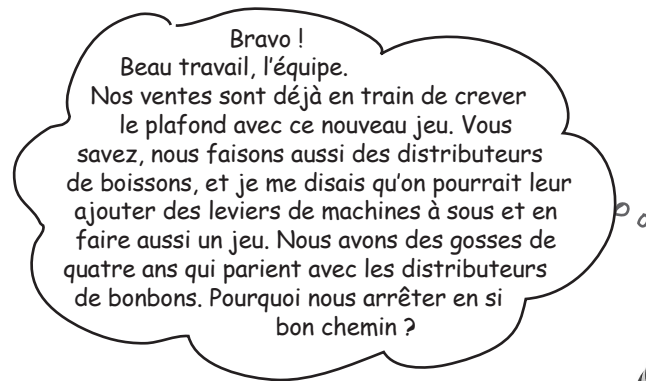
Vous avez inséré une pièce
Vous avez tourné...
Un bonbon va sortir...
Vous avez inséré une pièce
Vous avez tourné...
VOUS AVEZ GAGNÉ ! Deux bonbons pour le prix d'un !
Un bonbon va sortir...
Un bonbon va sortir...
Aïe, plus de bonbons !

Distribon, SARL.
Distributeur compatible Java, modèle 2004
Stock : 0 bonbons
L'appareil est en rupture de stock
%
```

Il n'y a pas de questions stupides

Q: Pourquoi faut-il un EtatGagnant ? Est-ce qu'EtatVendu ne pourrait pas juste délivrer deux bonbons ?

R: Excellente question. EtatVendu et EtatGagnant sont presque identiques, sauf qu'EtatGagnant délivre deux bonbons au lieu d'un. Vous pourriez sans doute placer le code qui délivre les deux bonbons dans EtatVendu. Bien sûr, il y a un inconvénient : vous avez maintenant DEUX états représentés dans une seule classe : l'état où vous êtes gagnant et celui où vous ne l'êtes pas. Vous sacrifiez donc la clarté pour réduire la duplication de code. Autre facteur à considérer, le principe que vous avez appris au chapitre précédent : une classe, une responsabilité. En plaçant la responsabilité d'EtatGagnant dans EtatVendu, vous avez attribué à EtatVendu DEUX responsabilités. Que se passe-t-il quand la promotion se termine ? Ou quand les enjeux du concours changent ? C'est donc une situation de compromis qui entraîne une décision de conception.



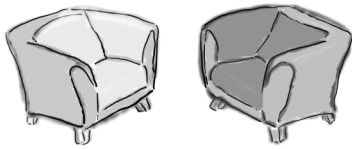
Bilan de santé...

Oui, le PDG de Distribon a probablement besoin d'un bilan de santé, mentale du moins, mais là n'est pas notre propos. Réfléchissons aux aspects du Distributeur que nous aimerions consolider avant de livrer la version « de luxe » :

- Nous avons beaucoup de code dupliqué dans les états Vendu et Gagnant et nous gagnerions à y faire un peu de ménage. Comment procéder ? Nous pourrions transformer Etat en classe abstraite et lui intégrer un comportement par défaut pour les méthodes. Après tout, des messages d'erreur comme « Vous avez déjà inséré une pièce » ne seront pas vus par le client. En conséquence, tout le comportement lié à la réponse aux erreurs pourrait être générique et hérité de la classe abstraite Etat.
- La méthode `delivrer()` est toujours appelée, même si la poignée est tournée quand il n'y a pas de pièce. Tant que la machine fonctionne correctement et ne délivre rien à moins d'être dans le bon état, ce serait facile à corriger en codant `tournerPoignee()` pour qu'elle retourne un booléen ou en introduisant des exceptions. Selon vous, quelle est la meilleure solution ?
- Toute la logique des transitions est dans les classes État. Quels sont les problèmes qui pourraient en résulter ? Allons-nous la transférer dans le Distributeur ? Quels seraient les avantages et les inconvénients de cette approche ?
- Allez-vous instancier beaucoup d'objets Distributeur ? Si oui, vous voudrez peut-être déplacer les instances des états dans des variables d'instance statiques et les partager. En ce cas, quelles seraient les modifications nécessaires dans le Distributeur et dans les états ?

← M'enfin, Paulo, je suis un distributeur de bonbons, pas un ordinateur !

Face à face :



Ce soir : **les patterns Stratégie et État enfin réunis.**

Stratégie

Hé, frangin. Tu as su que j'étais dans le chapitre 1 ?

J'étais parti donner un coup de main aux types du Patron de méthode : ils avaient besoin de moi pour finir leur chapitre. Alors, à quoi s'occupe mon noble frère ces temps derniers ?

Je ne sais pas, mais on dirait toujours que tu viens de pomper ce que je fais et que tu emploies d'autres mots pour le décrire. Réfléchis : je permets aux objets d'incorporer différents comportement ou algorithmes grâce à la composition ou à la délégation. Tu ne fais que me copier.

Ah oui ? Comment cela ? Je ne vois pas.

Oui, c'était du *bon* travail... et je suis sûr que tu vois que c'est beaucoup plus puissant que d'hériter de ton comportement, non ?

Désolé, il va falloir que tu t'expliques.

État

Oui, c'est ce que dit la rumeur publique.

Comme d'habitude : j'aide les classes à présenter des comportements différents dans différents états.

J'admets que nous avons des pratiques apparentées, mais mon intention est totalement différente de la tienne. Et la façon dont j'enseigne à mes clients à utiliser la composition et la délégation n'est absolument pas la même.

Eh bien, si tu passais un peu plus de temps à penser à autre chose qu'à *toi*, tu le saurais. Réfléchis un peu à ta façon de fonctionner : tu instancias une classe et tu lui donnes généralement un objet stratégie qui implémente un comportement quelconque. Comme au chapitre 1 où tu distribuais aux canards des comportements pour cancaner, d'accord ? Les vrais canards faisaient coin-coin pour de bon et les canards en plastique couinaient.

Oui, naturellement. Maintenant, réfléchis à *ma* façon de fonctionner : elle est complètement différente.

Stratégie

Oh, arrête ! Moi aussi, je peux changer de comportement lors de l'exécution ! C'est tout l'intérêt de la composition !

Bon, je te l'accorde, je n'encourage pas mes objets à avoir des ensembles de transitions bien définis entre les états. En fait, je préfère généralement contrôler la stratégie qu'ils appliquent.

Oui, oui, je te laisse à tes fantasmes. Tu te comportes comme si tu étais un grand pattern comme moi, mais, regarde bien : je suis dans le chapitre 1 alors qu'ils t'ont relégué au chapitre 10. Tu peux me dire combien de lecteurs vont vraiment aller jusque là ?

C'est mon frère ! Le rêveur de la famille....

État

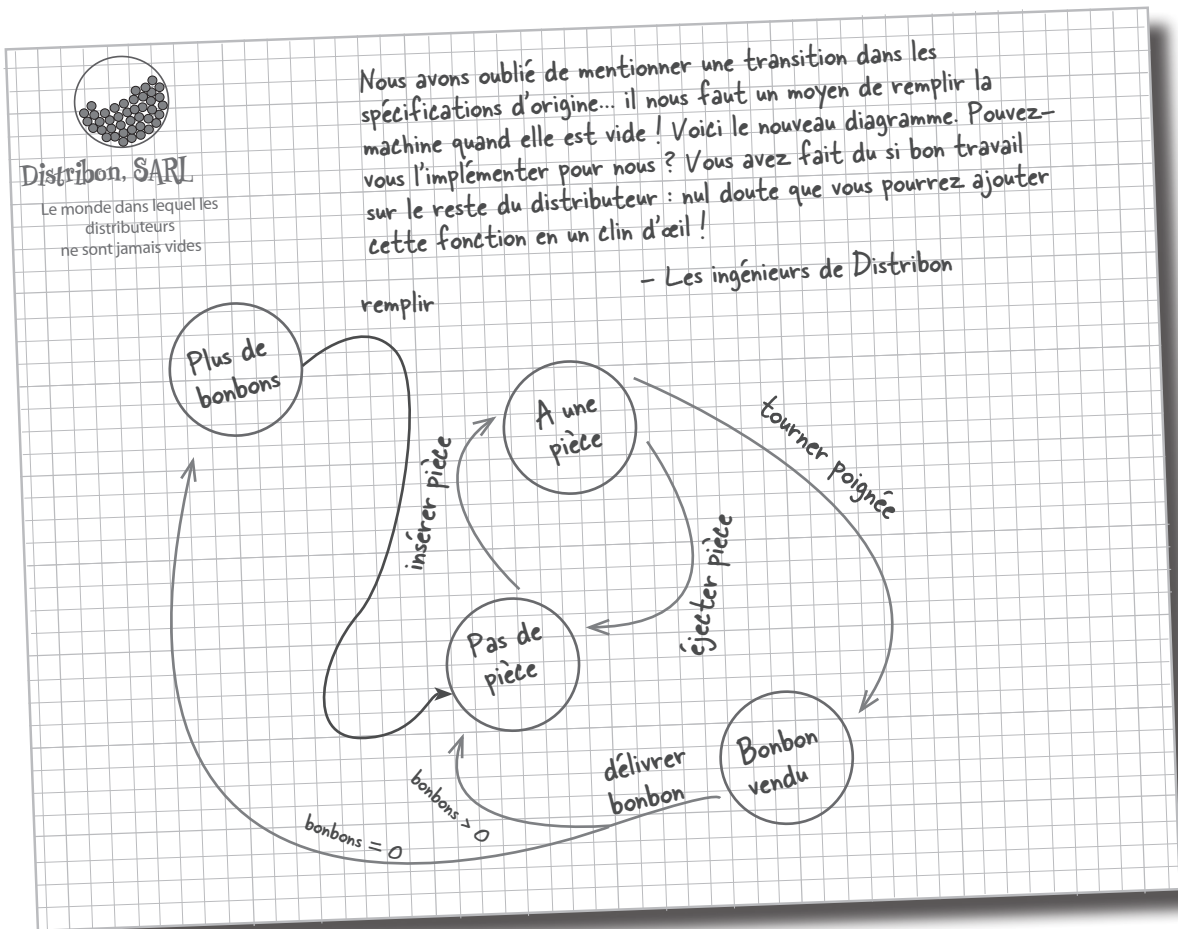
Bien. Quand mes objets Contexte sont créés, je peux leur indiquer leur état initial, mais ils peuvent modifier leur propre état ultérieurement.

Bien sûr, mais mon fonctionnement est construit autour d'états discrets. Mes objets Contexte changent d'état au fil du temps selon des transitions bien définies. Autrement dit, le changement d'état est intégré à mon schéma. Voilà comment je fonctionne !

Écoute, nous avons déjà dit que notre structure était semblable mais que notre intention était différente. Admets-le : il y a assez de place pour nous deux.

Tu plaisantes ? C'est un ouvrage de la collection Tête la première, et les lecteurs de Tête la première assurent. Bien sûr qu'ils vont arriver au chapitre 10 !

Nous avons failli oublier !



À vos crayons



Nous avons besoin de vous pour écrire la méthode `remplir()`. Elle n'a qu'un seul argument - le nombre de bonbons que nous allons ajouter dans le distributeur - et elle doit mettre à jour le compteur de bonbons et réinitialiser l'état de la machine.

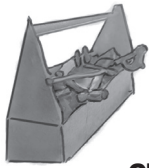
Vous avez fait un travail stupéfiant ! J'ai encore quelques idées qui vont révolutionner l'industrie des distributeurs et je veux que vous les implémentiez. Chut ! Je vous en parlerai dans le prochain chapitre.



Qui fait quoi ?

Faites correspondre chaque à sa description :

Pattern	Description
État	Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
Stratégie	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
Patron de méthode	Encapsule des comportements basés sur des états et délègue le comportement à l'état courant



Votre boîte à outils de concepteur

C'est la fin d'un autre chapitre : vous avez suffisamment de patterns pour passer n'importe quel entretien d'embauche les doigts dans le nez !

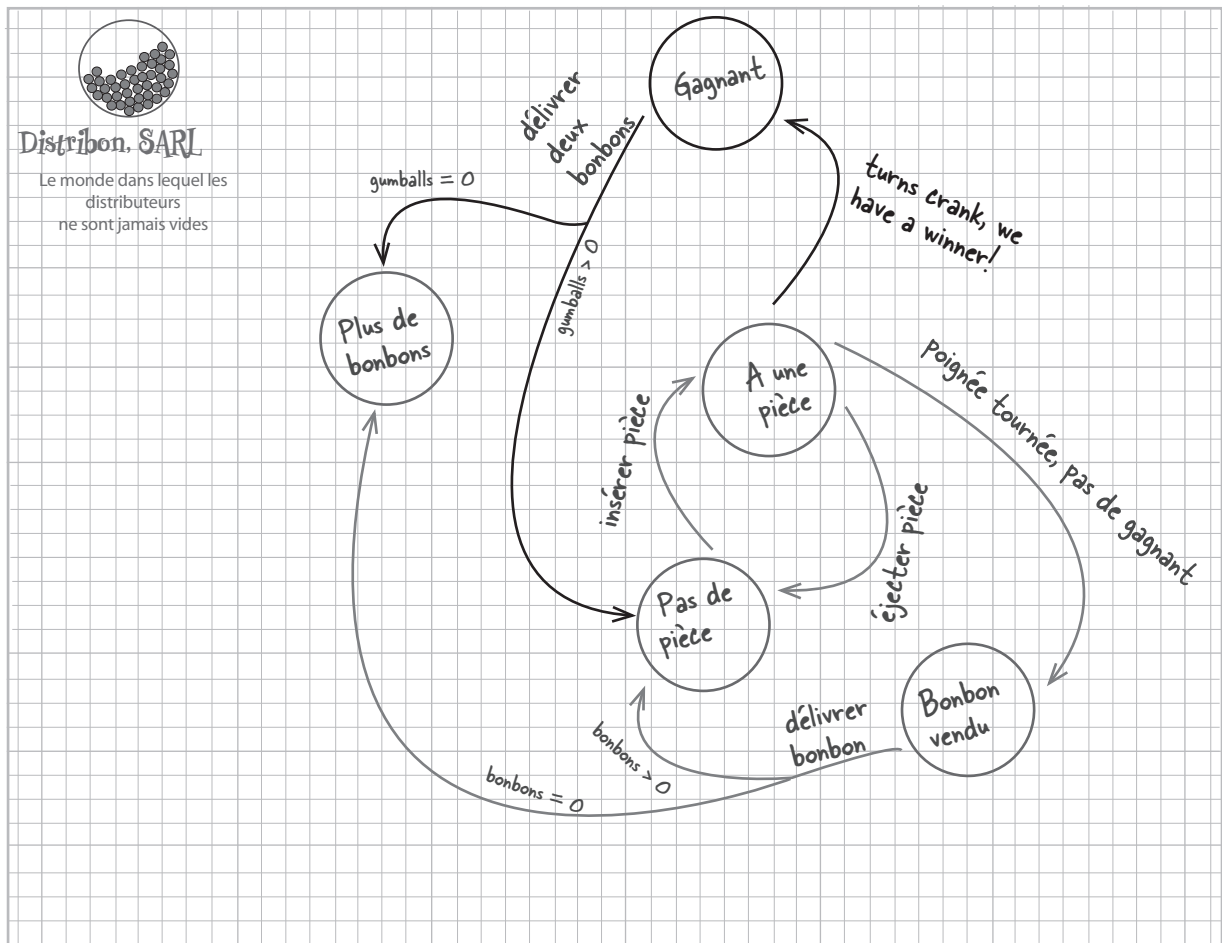


POINTS D'IMPACT

- Le pattern État permet à un objet d'avoir plusieurs comportements différents selon son état interne.
- À la différence d'une machine à états procédurale, le pattern État représente les états comme des classes à part entière.
- Le Contexte obtient son comportement en déléguant à l'objet représentant l'état courant avec lequel il est composé.
- En encapsulant chaque état dans une classe, nous isolons tous les changements qui seront nécessaires.
- Les diagrammes d'état des patterns État et Stratégie sont identiques mais leur intention diffère.
- Le pattern Stratégie configure généralement les classes Contexte avec un comportement ou un algorithme.
- Le pattern État permet à un Contexte de modifier son comportement à mesure que son état change.
- Les transitions entre états peuvent être contrôlées par les classes État ou par les classes Contexte.
- L'application du pattern État entraînera généralement une augmentation du nombre des classes dans votre conception.
- Les classes État peuvent être partagées entre les instances du Contexte.



Solutions des exercices





Solutions des exercices

À vos crayons

Dites si les affirmations suivantes décrivent l'état de notre implémentation.
(Plusieurs réponses possibles)

- | | |
|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <input checked="" type="checkbox"/> A. Ce code n'adhère pas au principe Ouvert-Fermé. | <input checked="" type="checkbox"/> D. Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles. |
| <input checked="" type="checkbox"/> B. Ce code ferait la fierté d'un programmeur FORTRAN. | <input checked="" type="checkbox"/> E. Nous n'avons pas encapsulé ce qui varie. |
| <input checked="" type="checkbox"/> C. Cette conception n'est pas orientée objet. | <input checked="" type="checkbox"/> F. Les ajouts ultérieurs sont susceptibles de provoquer des bogues dans le code. |

À vos crayons

Il reste une classe que nous n'avons pas implémentée : EtatEpuise. Pourquoi ne le feriez-vous pas ? Pour ce faire, réfléchissez soigneusement à la façon dont le Distributeur doit se comporter dans chaque situation. Vérifiez votre réponse avant de continuer...

Dans l'état Epuisé, nous ne pouvons absolument rien faire tant qu'on n'a pas rempli le Distributeur.

```
public class EtatEpuise implements Etat {
    Distributeur distributeur;

    public EtatEpuise(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous ne pouvez pas insérer de pièces, nous sommes en rupture de stock");
    }

    public void ejecterPiece() {
        System.out.println("Éjection impossible, vous n'avez pas inséré de pièce");
    }

    public void tournerPoignee() {
        System.out.println("Vous avez tourné, mais il n'y a pas de bonbons");
    }

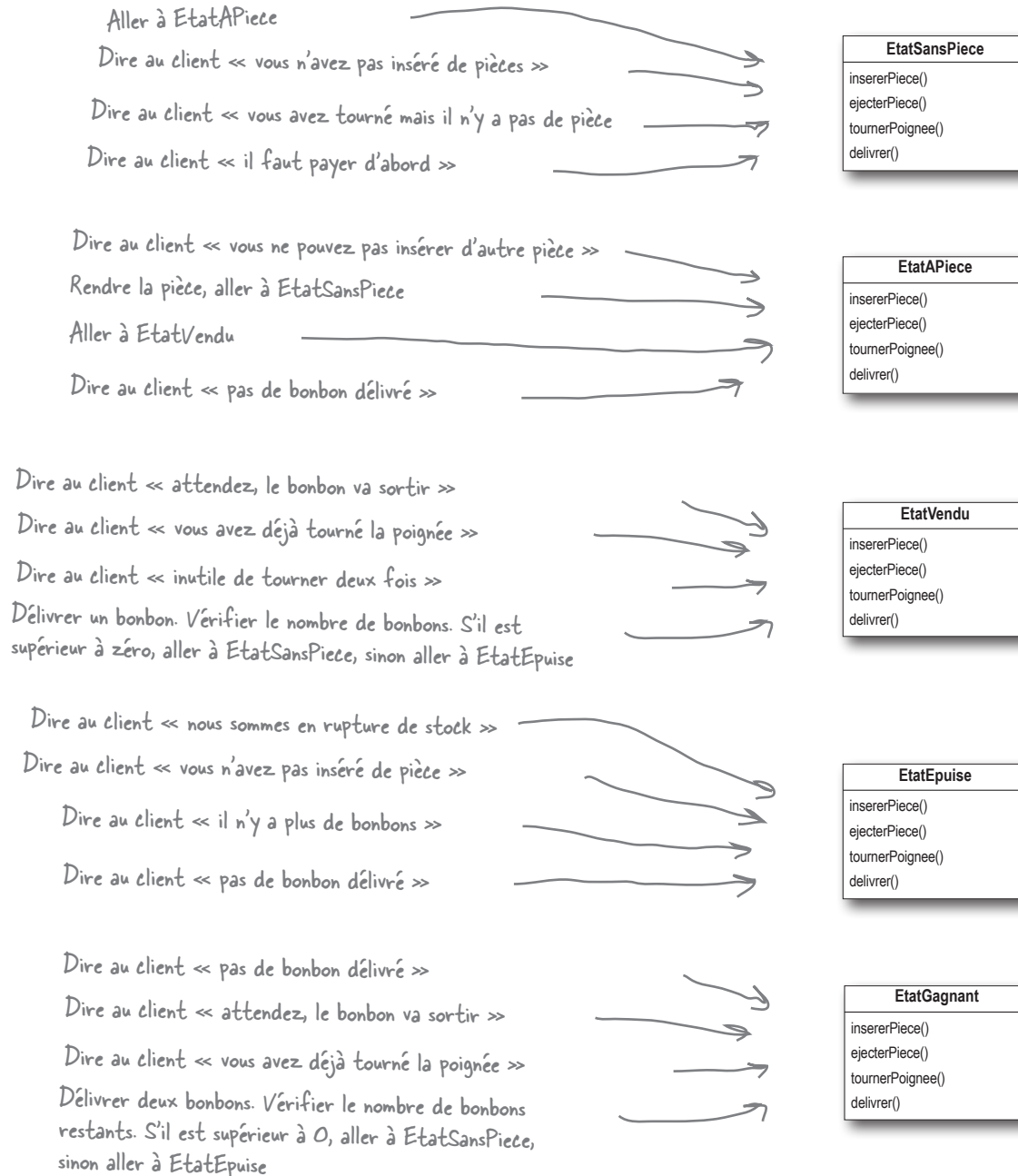
    public void delivrer() {
        System.out.println("Pas de bonbon délivré");
    }

    public String toString() {
        return "sold out";
    }
}
```

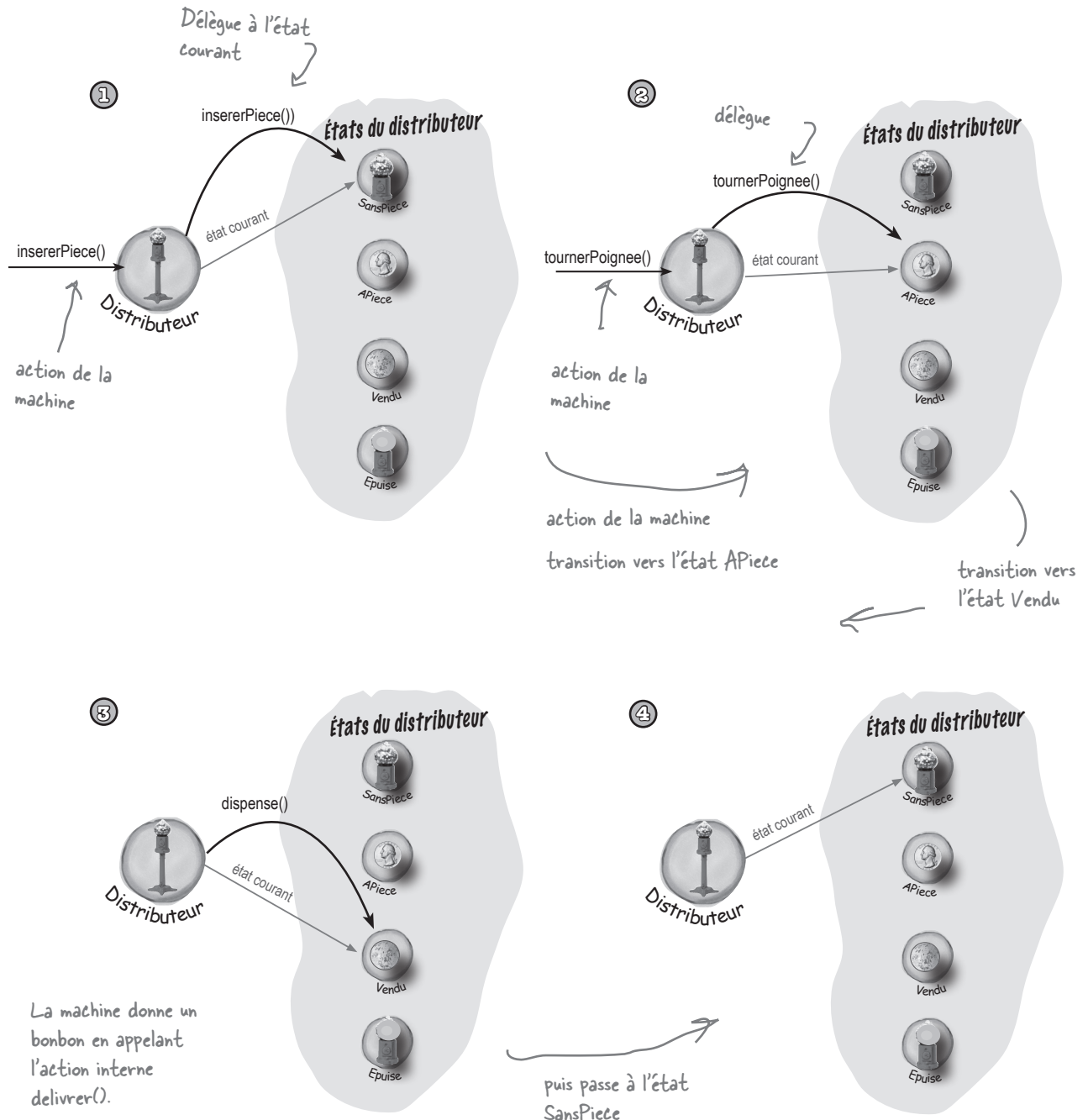

À vos crayons



Pour implémenter nos états, nous devons d'abord spécifier le comportement des classes quand chaque action est appelée. Annotez le diagramme ci-dessous en inscrivant le comportement de chaque action dans chaque classe. Nous en avons déjà indiqué quelques-uns pour vous..



Dans les coulisses : visite libre



Qui fait quoi ?

Faites correspondre chaque pattern à sa description :

Pattern	Description
État	Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
Stratégie	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
Patron de méthode	Encapsule des comportements basés sur des états et délègue le comportement à l'état courant

À vos crayons

Nous avons besoin de vous pour écrire la méthode `remplir()`. Elle n'a qu'un seul argument, le nombre de bonbons que nous allons ajouter dans le distributeur, et elle doit mettre à jour le compteur de bonbons et réinitialiser l'état de la machine.

```
void remplir(int nombre) {
    this.nombre = nombre;
    etat = etatSansPiece;
}
```


11 le pattern Proxy

Contrôler l'accès aux objets

Avec toi comme Proxy, je vais pouvoir racketter trois fois plus d'argent à mes copains à l'heure de la cantine !



Avez-vous déjà joué à « gentil flic, méchant flic » ? Vous êtes le gentil policier et vous rendez volontiers service de bonne grâce, mais comme vous ne voulez pas que qui que ce soit vous sollicite, vous demandez au méchant de servir d'intermédiaire. C'est là le rôle d'un proxy : contrôler et gérer les accès. Comme vous allez le voir, un proxy peut se substituer à un objet de bien des manières. Les proxies sont connus pour transporter des appels de méthodes entiers sur l'Internet à la place d'autres objets. On les connaît également pour remplacer patiemment un certain nombre d'objets bien paresseux.



Vous souvenez-vous du P.D.G.
de Distribon, SARL ?

Bonjour tout le monde.
J'aimerais vraiment bien
disposer d'un meilleur contrôle sur
les distributeurs de bonbons. Pouvez-
vous trouver un moyen de m'afficher
des rapports sur l'état du stock et sur
celui des appareils ?

Cela semble assez facile. Si vous vous en souvenez, le code du Distributeur contient déjà des méthodes qui permettent de connaître le nombre de bonbons (`getNombre()`) et l'état courant de la machine (`getEtat()`).

Il nous suffit donc de créer un rapport que nous pourrions afficher et transmettre au PDG. Mmmm... nous devons également sans doute ajouter un champ emplacement à chaque distributeur. Ainsi, le PDG pourra entretenir ses machines.

Allons-y, il n'y a plus qu'à coder. L'extrême brièveté de nos délais va impressionner le PDG.

Coder le contrôleur

Commençons par ajouter à la classe Distributeur du code qui lui permettra de gérer les emplacements :

```
public class Distributeur {
    // autres variables d'instance
    String emplacement;

    public Distributeur(String emplacement, int nombre) {
        // reste du constructeur
        this.emplacement = emplacement;
    }

    public String getEmplacement() {
        return emplacement;
    }

    // autres méthodes
}
```

Un emplacement n'est autre qu'une chaîne de caractères.

L'emplacement est transmis dans le constructeur et stocké dans la variable d'instance.

Ajoutons également une méthode get pour trouver l'emplacement quand nous en avons besoin.

Créons maintenant une autre classe, ControleurDistrib, qui extrait l'emplacement de la machine, le nombre de bonbons qu'elle a en stock et son état courant, puis qui affiche un petit rapport bien propre :

```
public class ControleurDistrib {
    Distributeur machine;

    public ControleurDistrib(Distributeur machine) {
        this.machine = machine;
    }

    public void rapport() {
        System.out.println("Distributeur : " + machine.getEmplacement());
        System.out.println("Stock Courant : " + machine.getNombre() + " bonbons");
        System.out.println("État courant : " + machine.getEtat());
    }
}
```

Le contrôleur accepte la machine dans son constructeur et l'affecte à la variable d'instance machine.

Notre méthode rapport() affiche simplement l'emplacement, le stock et l'état de la machine.

Tester le contrôleur

Nous avons implémenté cela en un rien de temps. Le PDG va être ravi et stupéfait par notre expertise en développement.

Maintenant, il suffit d'instancier un `ControleurDistrib` et de lui donner une machine à contrôler :

```
public class TestDistributeur {
    public static void main(String[] args) {
        int nombre = 0;

        if (args.length < 2) {
            System.out.println("TestDistributeur <nom> <stock>");
            System.exit(1);
        }

        nombre = Integer.parseInt(args[1]);
        Distributeur distributeur = new Distributeur(args[0], nombre);

        ControleurDistrib controleur = new ControleurDistrib(distributeur);

        // reste du code

        controleur.rapport();
    }
}
```

Transmettre un emplacement et un nombre de bonbons initial en ligne de commande.

Ne pas oublier de donner au constructeur un emplacement et un nombre...

...puis instancier un contrôleur et lui transmettre la machine sur laquelle il doit faire un rapport.

↑ Quand nous avons besoin d'un rapport sur la machine, nous appelons la méthode `rapport()`.

Fichier Édition Fenêtre Aide Bouillabaisse

```
%java TestDistributeur Marseille 112
```

```
Distributeur : Marseille
```

```
Stock Courant : 112 bonbons
```

```
État courant : En attente d'une pièce
```

Et voilà le résultat !



Le résultat a l'air parfait, mais je crois que je n'ai pas été clair. Je veux contrôler mes distributeurs à DISTANCE ! D'ailleurs, nous avons déjà des réseaux en place pour cela. Allons les gars, vous êtes censés être la génération Internet !



Guy : Un quoi distant ?

Paul : Un proxy distant. Réfléchissez : nous avons déjà écrit le code du contrôleur, exact ? Nous donnons à `ContrôleurDistrib` une référence à une machine et il nous retourne un rapport. Le problème est que le contrôleur s'exécute dans la même JVM que le distributeur et que le PDG veut rester tranquillement assis à son bureau et contrôler les machines à distance ! Mais si nous laissons notre classe `ContrôleurDistrib` telle quelle en lui transmettant un proxy à un objet distant ?

Guy : Je ne suis pas sûr de comprendre.

Luc : Moi non plus.

Paul : Commençons par le commencement... un proxy est un remplaçant d'un objet réel. En l'occurrence, le proxy se comporte exactement comme s'il était un objet `Distributeur`, mais, dans les coulisses, il communique sur le réseau et converse avec le vrai `Distributeur` distant.

Luc : Donc, tu dis que nous conservons le code tel quel et que nous passons au contrôleur une référence à une version proxy du `Distributeur`...

Guy : Et ce proxy fait semblant d'être l'objet réel, mais il ne fait que communiquer sur le réseau avec celui-ci.

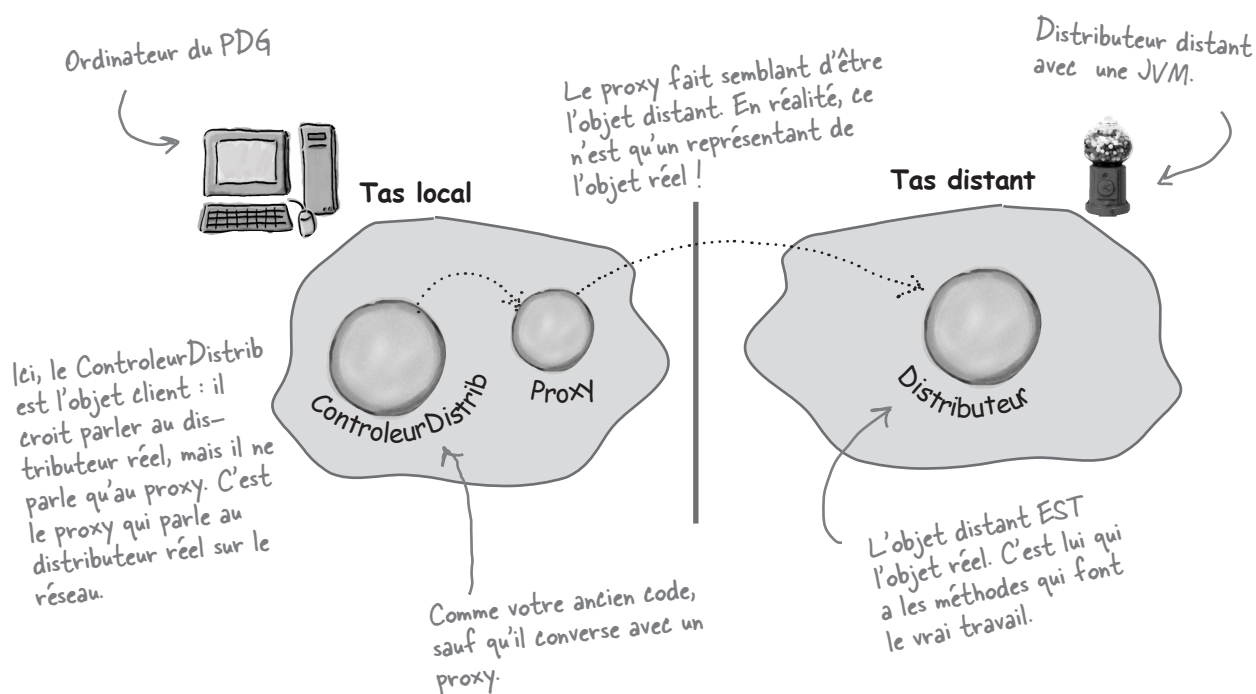
Paul : Oui, cela résume assez bien.

Guy : Cela semble plus facile à dire qu'à faire.

Paul : Peut-être, mais je suis plutôt optimiste. Nous devons nous assurer que le distributeur peut se comporter comme un service et accepter des requêtes sur le réseau. Nous devons également donner à notre contrôleur un moyen d'obtenir une référence à un objet proxy, mais Java contient déjà d'excellents outils intégrés qui vont nous aider. Mais d'abord, parlons un peu plus des proxies distants...

Le rôle du proxy distant

Un « proxy distant » joue le rôle de représentant local d'un objet distant. Qu'est-ce qu'un « objet distant » ? Rien d'autre qu'un objet qui réside sur le tas d'une autre JVM (ou, plus généralement, un objet qui s'exécute dans un autre espace d'adressage). Qu'est-ce qu'un « représentant local » ? C'est un objet sur lequel vous pouvez appeler des méthodes locales qui seront relayées à l'objet distant.



Votre objet client se comporte comme si c'était lui qui exécutait les appels de méthodes à distance. En réalité, il appelle les méthodes sur un objet « proxy » local qui gère tous les détails de bas niveau de la communication sur le réseau.



MUSCLEZ VOS NEURONES

Avant d'aller plus loin, réfléchissez à la façon dont vous concevriez un système qui permettrait les appels de méthode à distance.
Comment faciliteriez-vous le travail du développeur pour qu'il ait à écrire le moins de code possible ?
Comment feriez-vous pour que l'invocation à distance soit transparente ?



MUSCLEZ VOS NEURONES²

L'invocation à distance doit-elle être totalement transparente ? Est-ce une bonne idée ? Quel est le problème potentiel de cette approche ?

Ajouter un proxy distant au code de contrôle du Distributeur

Sur le papier, tout va bien. Mais comment crée-t-on un proxy qui sait comment invoquer une méthode qui réside dans une autre JVM ?

Mmmm... Vous ne pouvez pas obtenir de référence à un objet qui réside sur un autre tas, non ? Autrement dit, vous ne pouvez pas écrire :

Canard c = <objet sur un autre tas>

Ce que la variable c référence doit être dans le même espace d'adressage que le code qui exécute l'instruction. Alors, comment procéder ? Eh bien, c'est là que RMI (Remote Method Invocation) de Java entre en scène... RMI nous fournit un moyen d'accéder aux objets appartenant à une JVM distante et nous permet d'appeler leurs méthodes.

Vous avez peut-être rencontré RMI dans Java tête la première. Si ce n'est pas le cas, nous allons faire un léger détour et présenter RMI avant d'ajouter la prise en charge du proxy au code du Distributeur.

Voici donc ce que nous allons faire :

- 1 Premièrement, nous allons faire un détour et l'étudier un peu. Même si vous connaissez bien RMI, vous voudrez peut-être nous accompagner et admirer le paysage.**
- 2 Puis nous allons reprendre notre Distributeur et le transformer en un service distant fournissant un ensemble de méthodes qui peuvent être appelées à distance.**
- 3 Enfin, nous créerons un proxy capable de converser avec un Distributeur distant, en utilisant RMI, et nous réassemblerons le système de contrôle pour que le PDG puisse surveiller un nombre quelconque de machines distantes.**



Si vous ne connaissez pas RMI, lisez les quelques pages suivantes. Sinon, vous pouvez les parcourir rapidement pour faire une petite révision.

Méthodes distantes : les bases



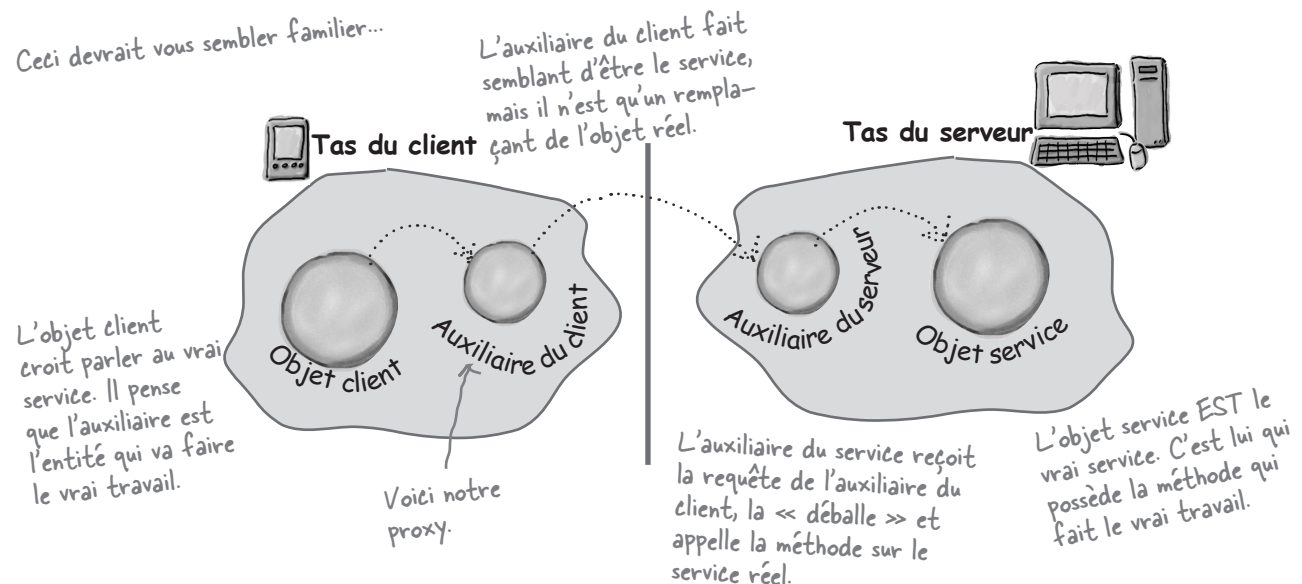
Supposons que nous voulions concevoir un système qui nous permette d'appeler un objet local qui fait suivre chaque requête à un objet distant. Comment procéderions-nous ? Il nous faut une paire d'objets auxiliaires qui vont s'occuper des communications à notre place. Ces objets auxiliaires permettent au client de se comporter comme s'il appelait une méthode sur un objet local (ce qu'il est en fait). Le client appelle une méthode sur l'auxiliaire du client, comme si cet auxiliaire était le service réel. Puis l'auxiliaire se chargera de transmettre la requête pour nous.

Autrement dit, l'objet client pense qu'il appelle une méthode sur le service distant, parce que l'auxiliaire fait semblant d'être l'objet service, comme s'il était l'objet qui possède la méthode que le client veut appeler.

Mais l'auxiliaire du client n'est pas réellement le service distant. Même s'il se comporte comme si c'était le cas (parce qu'il a la même méthode que le celle que le service annonce), il ne possède en réalité rien de la logique à laquelle le client s'attend. En lieu et place, il contacte le serveur, transfère les informations sur l'appel de méthode (nom de la méthode, arguments, etc.) et attend un retour du serveur.

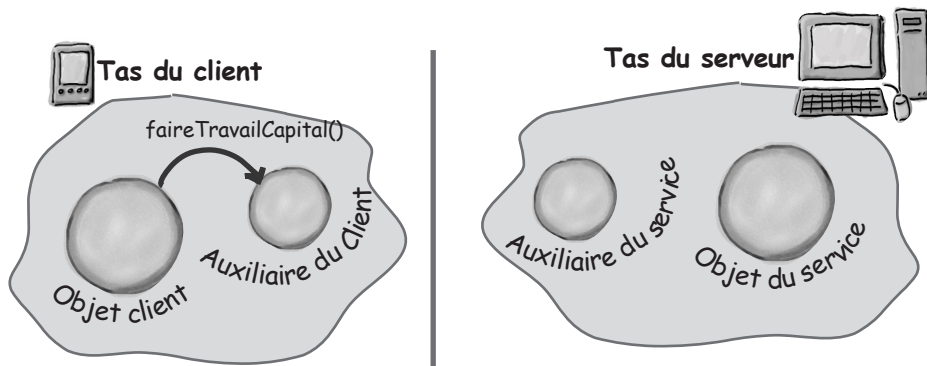
Côté serveur, l'auxiliaire du service reçoit la requête de l'auxiliaire du client (via une connexion Socket), lit les informations sur l'appel et invoque la méthode réelle sur l'objet service réel. Ainsi, pour l'objet service, l'appel est local. Il vient de l'auxiliaire du service, non d'un client distant.

L'auxiliaire du service obtient alors la valeur de retour du service, l'« emballe » et la renvoie (sur le flot de sortie du Socket) à l'auxiliaire du client. Ce dernier « déballe » l'information et retourne la valeur à l'objet client.

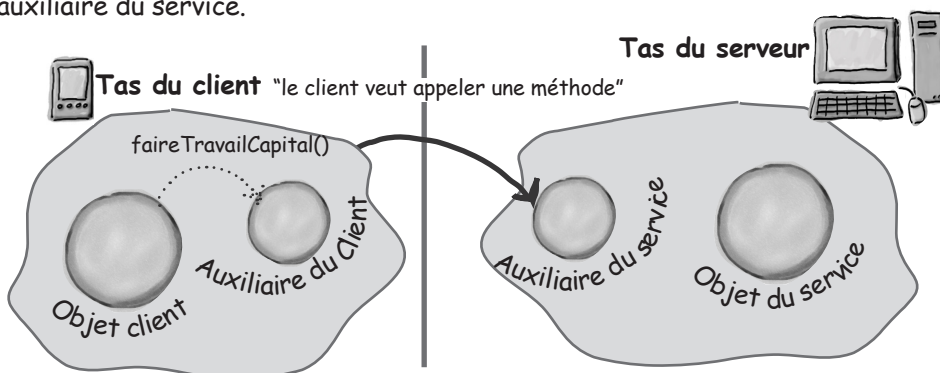


Comment fonctionne l'appel de méthode ?

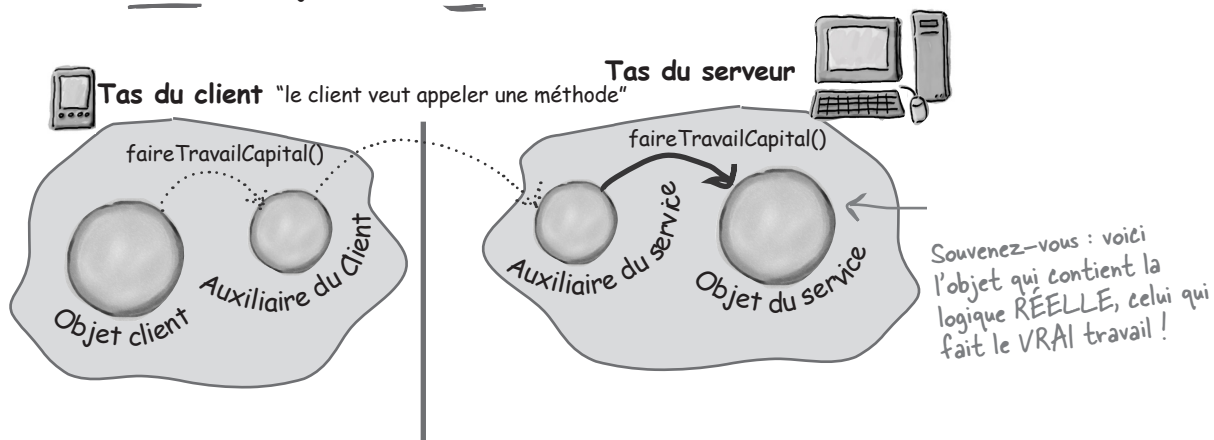
- ① L'objet client appelle faireTravailCapital() sur l'auxiliaire du client.



- ② L'auxiliaire du client emballe les informations sur l'appel (nom de méthode, arguments, etc.) et l'envoie sur le réseau à l'auxiliaire du service.

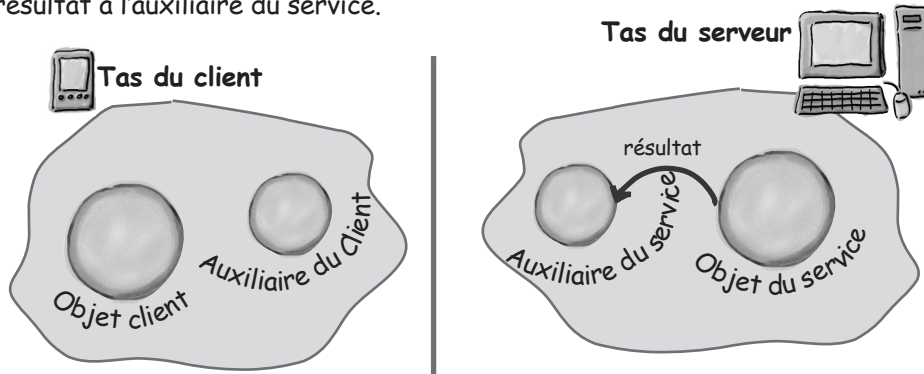


- ③ L'auxiliaire du service déballe les informations qu'il a reçues de l'auxiliaire du client, détermine quelle méthode appeler (et sur quel objet) et invoque la méthode réelle sur l'objet service réel.

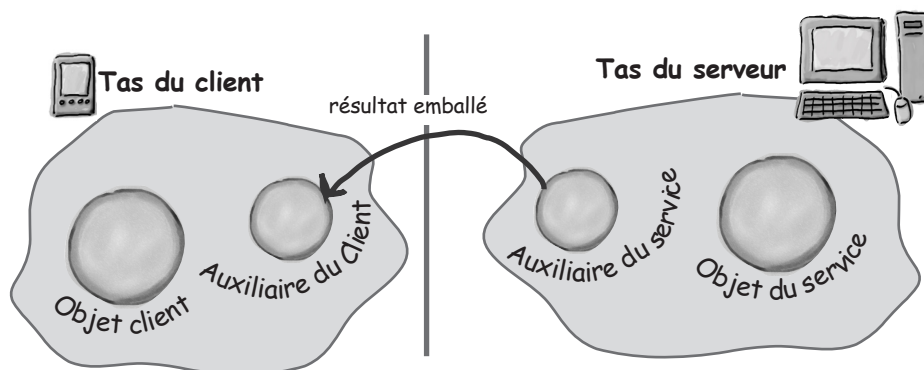




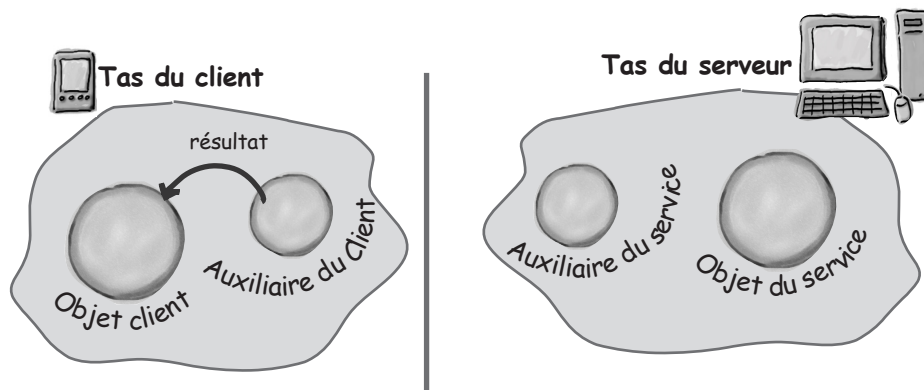
- ④ La méthode est appelée sur l'objet service, qui retourne un résultat à l'auxiliaire du service.



- ⑤ L'auxiliaire du service emballe les informations retournées par l'appel et les renvoie sur le réseau à l'auxiliaire du client.



- ⑥ L'auxiliaire du client déballe les valeurs retournées et les transmet à l'objet client. Pour l'objet client, tout ce processus a été transparent.



RMI : vue d'ensemble

Bien. Vous savez maintenant comment fonctionnent les méthodes distantes. Il ne vous reste plus qu'à apprendre comment utiliser RMI pour effectuer des appels de méthodes à distance.

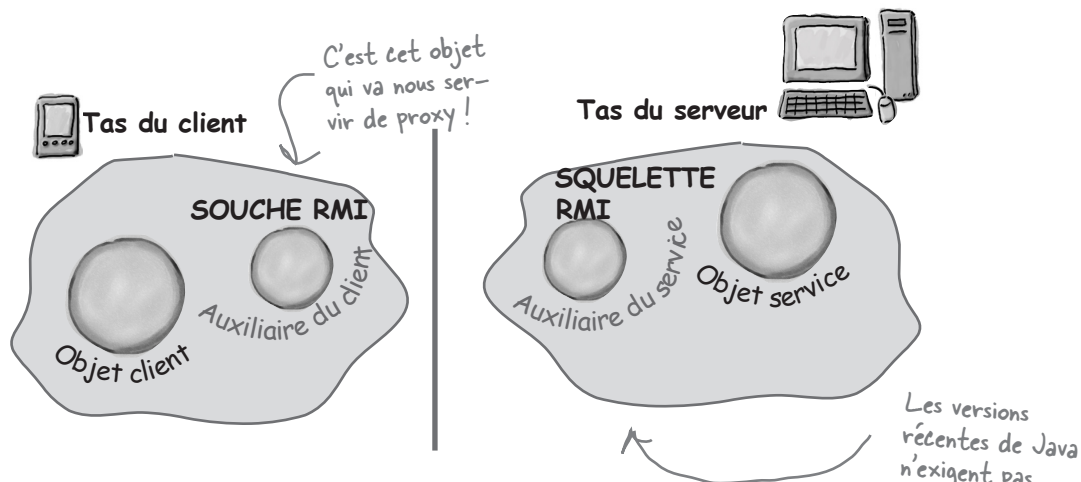
Que fait RMI pour vous ? Il construit les objets auxiliaires du client et du service, allant jusqu'à créer un objet auxiliaire du client ayant les mêmes méthodes que le service distant. La bonne nouvelle avec RMI, c'est que vous n'avez pas besoin d'écrire une seule ligne de code pour le réseau ni pour les entrées/sorties. Votre client peut appeler les méthodes distantes (celles du service réel) exactement comme dans des appels de méthode normaux sur des objets s'exécutant dans leur JVM locale.

RMI fournit également toute l'infrastructure qui assure le fonctionnement de l'ensemble lors de l'exécution, y compris un service de recherche que le client peut utiliser pour localiser les objets distants et y accéder.

Mais les appels RMI et les appels de méthodes locaux (normaux) présentent une différence. N'oubliez pas que même si le client a l'impression que l'appel de méthode est local, l'auxiliaire du client transmet l'appel de méthode sur un réseau. Il faut donc des méthodes pour le réseau et les E/S. Et que savons-nous sur les méthodes réseau et les méthodes d'E/S?

Elles sont risquées ! Elles peuvent échouer ! Et alors elles lancent des exceptions en pagaille. En conséquence, le client doit en accepter le risque. Nous allons voir comment dans quelques pages.

Terminologie RMI : pour RMI, l'auxiliaire du client est une « souche » et l'auxiliaire du service est un « squelette ».



Voyons maintenant les étapes nécessaires pour transformer un objet en service qui accepte des appels distants, et celles qui permettent au client d'effectuer des appels distants.

Attachez vos ceintures : les étapes sont nombreuses et la route parsemée de bosses et de virages. Rien de vraiment inquiétant toutefois.

Créer le service distant

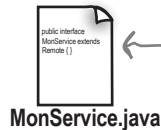


Voici une **vue d'ensemble** des cinq étapes de la création du service distant. Autrement dit, les étapes nécessaires pour prendre un objet ordinaire et faire en sorte qu'il puisse être appelé par un client distant. Nous ferons ceci plus tard dans notre Distributeur. Pour l'instant, nous allons parcourir les étapes que nous expliquerons ensuite chacune en détail.

Étape 1 :

Créer une Interface distante

L'interface distante définit les méthodes qu'un client peut appeler. C'est elle que le client va utiliser comme type pour votre service. La souche et le service réel l'implémentent tous les deux !



Cette interface définit les méthodes distantes que vous voulez que les clients appellent.

Étape 2 :

Créer une Implémentation distante

C'est la classe qui fait le VRAI travail. C'est elle qui possède l'implémentation réelle des méthodes distantes définies dans l'interface distante. C'est l'objet sur lequel le client va appeler les méthodes (par exemple notre Distributeur !).



Le service réel ; la classe qui possède les méthodes qui font le vrai travail. Elle implémente l'interface distante.

Étape 3 :

Générer les souches et les squelettes en exécutant rmic

Ce sont les « auxiliaires » du client et du serveur. Vous n'avez pas à créer ces classes ni même à regarder le code source qui les génère. Tout est géré automatiquement quand vous exécutez l'outil rmic livré avec votre kit de développement Java.



L'exécution de la commande rmic suivi du nom de la classe d'implémentation...

..génère deux nouvelles classes pour les objets auxiliaires



Étape 4 :

Exécuter le registre RMI (rmiregistry)

Le registre RMI est comparable aux pages blanches d'un annuaire téléphonique. C'est là que le client va chercher le proxy (l'objet auxiliaire/souche).



Exécutez ceci dans une fenêtre de commandes séparée.

Étape 5 :

Lancer le service distant

Vous devez mettre l'objet service en... service. Votre classe d'implémentation crée une instance du service et l'enregistre dans le registre RMI. Ce faisant, elle met ce service à la disposition des clients.



Étape 1 : créer une interface distante

① Étendre `java.rmi.Remote`

`Remote` est une interface « marqueur », ce qui veut dire qu'elle n'a pas de méthodes. Toutefois, cela a une signification spéciale pour RMI et vous devez appliquer cette règle. Remarquez que nous avons écrit « `extends` » : une interface est autorisée à étendre une autre interface.

```
public interface MonService extends Remote {
```

Ceci nous indique que nous allons utiliser l'interface pour prendre en charge des appels distants.

② Déclarer que toutes les méthodes lancent une `RemoteException`

L'interface distante est celle que le client utilise comme type pour le service. Autrement dit, le client invoque des méthodes sur une entité qui implémente l'interface distante. Cette entité est la souche, bien sûr, et comme c'est la souche qui se charge du réseau et des E/S, toutes sortes de Gros Problèmes peuvent survenir. Le client doit reconnaître l'existence de ces risques en gérant ou en déclarant les exceptions distantes. Si les méthodes d'une interface déclarent des exceptions, tout code appelant des méthodes sur une référence de ce type (le type de l'interface) doit gérer ou déclarer les exceptions.

```
import java.rmi.* ;

public interface MonService extends Remote {
    public String direBonjour() throws RemoteException;
}
```

L'interface `Remote` est dans `java.rmi`

Tout appel de méthode distant est considéré comme « risqué ». Déclarer une `RemoteException` pour chaque méthode oblige le client à faire attention et à reconnaître qu'il pourrait y avoir un dysfonctionnement.

③ S'assurer que les arguments et les valeurs de retour sont de type primitif ou sérialisables

Les arguments et les valeurs de retour d'une méthode distante doivent être de type primitif ou sérialisable. Réfléchissez : tout argument d'une méthode distante doit être « emballé » et expédié sur le réseau et cela se fait au moyen de la sérialisation. Il en va de même pour les valeurs de retour. Si vous utilisez des types primitifs, des chaînes de caractères et la majorité des types de l'API (y compris les tableaux et les collections), tout ira bien. Si vous transmettez vos propres types, vérifiez que vos classes implémentent `Serializable`.

```
public String direBonjour() throws RemoteException;
```

Comme le serveur va réexpédier cette valeur de retour sur le réseau au client, celle-ci doit être sérialisable. C'est de cette façon que les arguments et les valeurs de retour sont « emballés » et transmis.

Consultez *Java Tête la première* si vous avez besoin de vous rafraîchir le mémoire sur `Serializable`.

Étape 2 : créer une implémentation distante



① Implémenter l'interface distante

Votre service doit implémenter l'interface distante – celle qui possède les méthodes que votre client va appeler.

```
public class MonServiceImpl extends UnicastRemoteObject implements MonService {
    public String direBonjour() {
        return "Le serveur dit, 'Bonjour'";
    }
    // reste du code de la classe
}
```

Le compilateur va vérifier que vous avez implémenté toutes les méthodes de l'interface que vous implémentez. Dans ce cas, il n'y en a qu'une.

② Étendre UnicastRemoteObject

Pour pouvoir fonctionner comme service distant, votre objet doit disposer d'une fonctionnalité liée à la « distance ». Le moyen le plus simple consiste à étendre UnicastRemoteObject (du package java.rmi.server) et à laisser cette classe (votre superclasse) travailler à votre place.

```
public class MonServiceImpl extends UnicastRemoteObject implements MonService {
```

③ Écrire un constructeur sans arguments qui déclare une RemoteException

Votre nouvelle superclasse, UnicastRemoteObject, présente un petit problème : son constructeur lance une RemoteException. Le seul moyen de le résoudre est de déclarer un constructeur pour votre implémentation distante, dans le seul but de disposer d'un endroit pour déclarer une RemoteException. Souvenez-vous : lorsqu'une classe est instanciée, le constructeur de la superclasse est toujours appelé. Si celui-ci lance une exception, vous n'avez pas d'autre choix que de déclarer que votre constructeur lance également une exception.

```
public MonServiceImpl() throws RemoteException { }
```

Vous n'avez pas besoin de placer quoi que ce soit dans le constructeur. Il vous faut simplement un moyen de déclarer que le constructeur de votre superclasse lance une exception.

④ Enregistrer le service dans le registre RMI

Maintenant que vous avez un service distant, vous devez le mettre à la disposition des clients distants. Pour ce faire, vous l'instanciez et vous le placez dans le registre RMI (qui doit être en train de s'exécuter, sinon cette ligne de code échoue).

Quand vous enregistrez l'objet implémentation, le système RMI place la souche dans le registre, puisque c'est d'elle dont le client a besoin en réalité. Enregistrez votre service grâce à la méthode statique rebind() de la classe java.rmi.Naming.

```
try {
    MonService service = new MonServiceImpl();
    Naming.rebind("BonjourDistant", service);
} catch (Exception ex) { ... }
```

Donnez un nom à votre service (pour que le client puisse le rechercher dans le registre) et enregistrez-le dans le registre RMI. Quand vous liez l'objet service, RMI remplace le service par la souche et place celle-ci dans le registre.

Étape 3 : générer des souches et des squelettes

1 Exécutez `rmic` sur la classe de l'implémentation distante (non sur l'interface distante)

L'outil `rmic`, livré avec le kit de développement Java, prend une implémentation de service et crée deux nouvelles classes, la souche et le squelette. Il applique une convention de nommage : le nom de votre implémentation avec soit `_Stub` soit `_Skel` ajouté à la fin. `rmic` dispose d'autres options, notamment ne pas générer de squelettes, lire le code source de ces classes et même utiliser IIOP comme protocole. Ici, nous procédons comme vous procéderez généralement. Les classes atterriront dans le répertoire courant (celui dans lequel vous vous êtes déplacé avec la commande `cd`). Souvenez-vous : puisque `rmic` doit être capable de voir la classe de votre implémentation distante, vous exécuterez probablement `rmic` depuis le répertoire dans lequel elle se trouve. (C'est exprès que nous n'utilisons pas de packages ici pour simplifier. Dans le monde réel, vous devriez prendre en compte les structures de répertoires des packages et les noms qualifiés.)

Remarquez qu'il n'y a pas << .class >> à la fin. Juste le nom de la classe.

```
Fichier Édition Fenêtre Aide Whuffie
%rmic MonServiceImpl
```

RMIC génère deux nouvelles classes pour les objet auxiliaires..



MonServiceImpl_Stub.class



MonServiceImpl_Skel.class

Étape 4 : exécuter `rmiregistry`

1 Ouvrez une fenêtre de commandes et lancez `rmiregistry`.

Vérifiez bien que vous partez d'un répertoire qui a accès à vos classes. Le plus simple consiste à partir de votre répertoire « classes ».

```
Fichier Édition Fenêtre Aide Comment ?
%rmiregistry
```

Étape 5 : démarrer le service

1 Ouvrez une autre fenêtre de commandes et lancez votre service

Vous pouvez le faire à parti de la méthode `main()` de votre implémentation distante ou à partir d'une classe « lanceur » séparée. Dans cet exemple simple, nous avons placé le code qui démarre le service dans la classe de l'implémentation, dans une méthode `main()` qui instancie l'objet et l'enregistre dans le registre RMI.

```
Fichier Édition Fenêtre Aide Comment ?
%java MonServiceImpl
```


Code complet côté serveur



L'interface distante : interface publique

```
import java.rmi.*;
```

RemoteException et l'interface Remote sont dans le package java.rmi.

```
public interface MonService extends Remote {
```

Votre interface DOIT étendre java.rmi.Remote

```
    public String direBonjour() throws RemoteException;
```

Toutes vos méthodes distantes doivent déclarer une RemoteException.

```
}
```

Le service distant (l'implémentation) :

```
import java.rmi.*;
```

UnicastRemoteObject est dans le package java.rmi.server.

```
import java.rmi.server.*;
```

Étendre UnicastRemoteObject est le moyen le plus simple de créer un objet distant.

```
public class MonServiceImpl extends UnicastRemoteObject implements MonService {
```

Vous devez bien sûr implémenter toutes les méthodes de l'interface. Mais remarquez que vous n'avez PAS à déclarer de RemoteException.

```
    public String direBonjour() {
```

Vous DEVEZ implémenter votre interface distante !!

```
        return "Le serveur dit 'Bonjour'";
```

```
    }
```

```
    public MonServiceImpl() throws RemoteException { }
```

```
    public static void main (String[] args) {
```

```
        try {
```

```
            MonService service = new MonServiceImpl();
```

```
            Naming.rebind("BonjourDistant", service);
```

```
        } catch (Exception ex) {
```

Comme le constructeur de votre superclasse (pour UnicastRemoteObject) déclare une exception, VOUS devez écrire un constructeur, parce que cela signifie que votre constructeur appelle du code à risques (son super-constructeur).

```
            ex.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```


Comment le client obtient-il l'objet souche ?

Le client doit obtenir l'objet souche (notre proxy), puisque c'est sur lui que le client va appeler les méthodes. Et c'est là que le registre RMI intervient. Le client effectue une « recherche », comme dans les pages blanches d'un annuaire téléphonique, et dit en substance « Voici un nom, et je voudrais la souche qui va avec ».

Jetons un coup d'œil au code qui permet de rechercher et d'extraire une souche.

Voilà comment ça marche



Code à la loupe

Le client utilise toujours l'interface distante comme type du service. En fait, il n'a jamais besoin de connaître le vrai nom de classe du service distant.

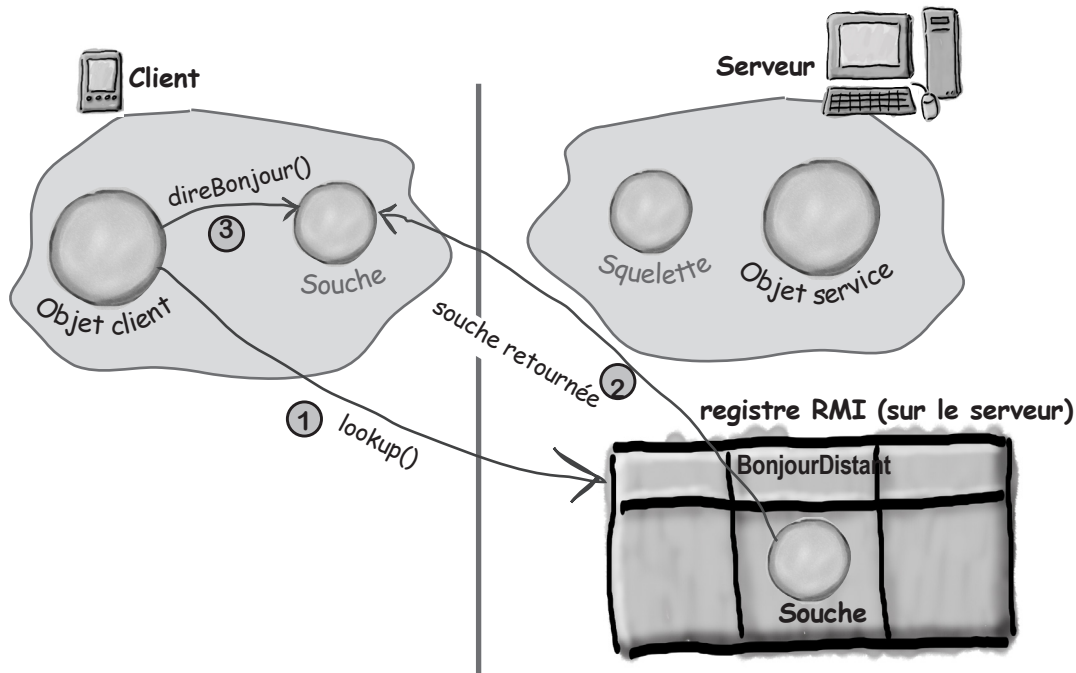
```
MonService service =  
(MonService) Naming.lookup("rmi://127.0.0.1/BonjourDistant");
```

Vous devez donner le type de l'interface, puisque la méthode lookup() retourne un type Object.

lookup() est une méthode statique de la classe Naming.

Le nom d'hôte ou l'adresse IP de la machine sur laquelle le service s'exécute.

Ceci doit être le nom sous lequel le service a été enregistré



Comment ça marche...

- ① **Le client consulte le registre RMI**
`Naming.lookup('rmi://127.0.0.1/BonjourDistant') ;`
- ② **Le registre RMI retourne l'objet souche**
 (comme valeur de retour de la méthode `lookup()`) et RMI le déséréalise automatiquement. Vous DEVEZ avoir la classe de la souche (que `rmic` a générée pour vous) sur client ou la souche ne sera pas déséréalisée.
- ③ **Le client invoque une méthode sur la souche, comme si elle ÉTAIT le service réel**

Code complet côté serveur

```
import java.rmi.*;

public class monClientDistant {
    public static void main (String[] args) {
        new monClientDistant().go();
    }

    public void go() {
        try {
            MonService service = (MonService) Naming.lookup("rmi://127.0.0.1/BonjourDistant");

            String s = service.direBonjour();

            System.out.println(s);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

La classe Naming (pour la recherche dans le registre RMI) est dans le package java.rmi.

De type Object à la sortie du registre : n'oubliez pas de le « caster ».

Adresse IP ou nom d'hôte.

et le nom utilisé pour lier/re-lier le service.

On dirait vraiment un bon vieil appel de méthode ! (sauf qu'il doit déclarer la RemoteException.)



Trucs de geeks

Comment le client obtient-il la classe de la souche ?

Voici maintenant la question intéressante. D'une manière ou d'une autre, le client doit avoir la classe de la souche (que vous avez générée plus tôt avec `rmic`) au moment où il effectue la recherche, sinon la souche ne sera pas désérialisée sur le client et tout explosera. Le client doit également avoir les classes de tous les objets sérialisés retournés par les appels de méthode à l'objet distant. Dans un système simple, vous pouvez vous contenter de fournir ces classes directement au client.

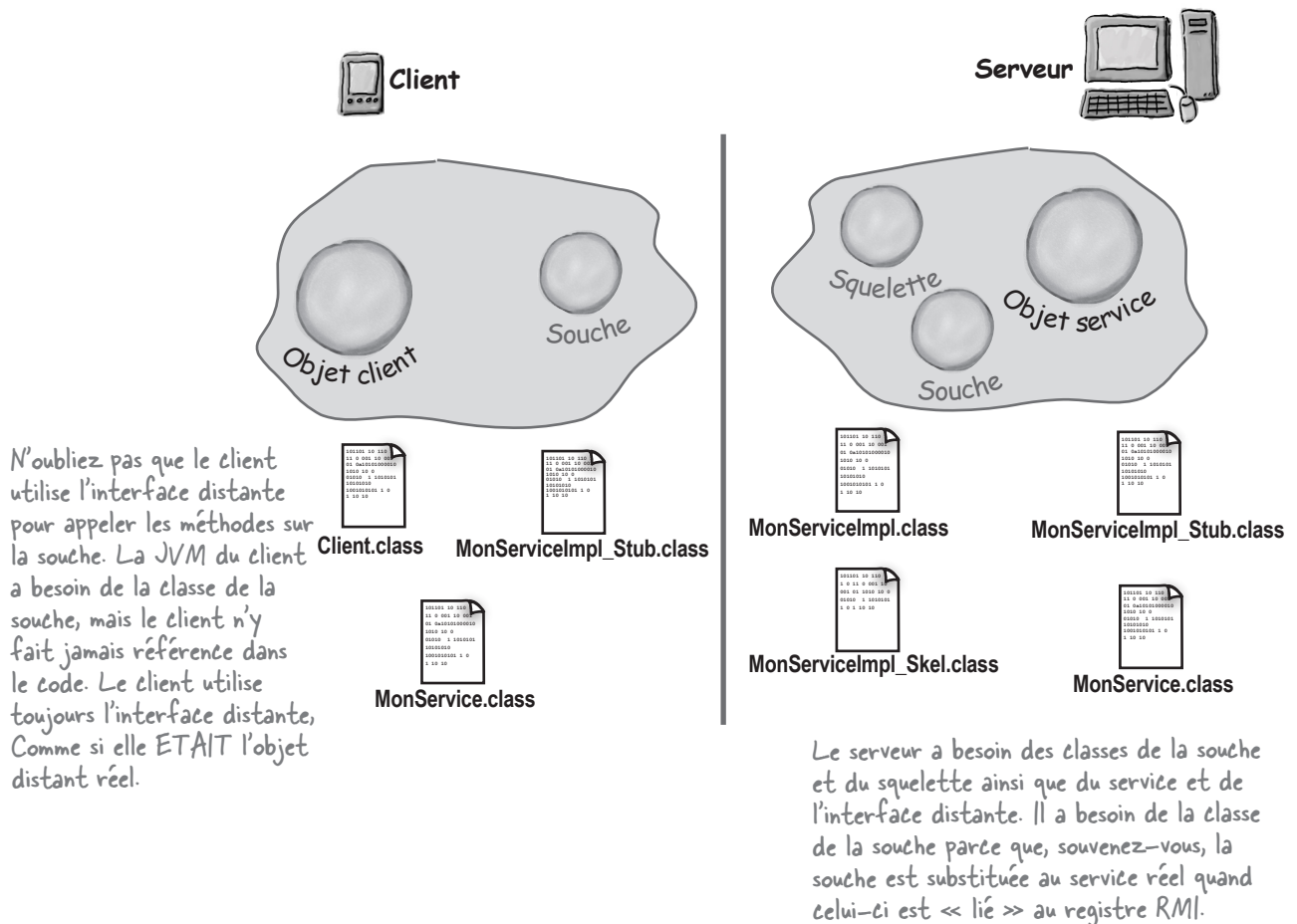
Mais il existe un moyen beaucoup plus cool, bien qu'il dépasse la portée de cet ouvrage. Néanmoins, si vous êtes intéressé, ce moyen beaucoup plus cool se nomme « téléchargement dynamique des classes ». Avec le téléchargement dynamique des classes, les objets sérialisés (comme la souche) sont « estampillés » avec une URL qui indique au système RMI résidant sur le client où trouver le fichier `.class` correspondant à l'objet. Puis, au cours du processus de désérialisation, si RMI ne trouve pas la classe localement, il utilise cette URL pour émettre une requête HTTP GET et récupérer le fichier `.class`. Il vous faut donc un serveur web simple pour servir les fichiers `.class`. Vous devez également modifier certains paramètres de sécurité sur le client. Le téléchargement dynamique des classes présente encore quelques aspects un peu délicats, mais vous savez le principal.

En ce qui concerne spécifiquement l'objet souche, le client dispose d'un autre moyen d'obtenir la classe, mais seulement en Java 5, Nous en parlerons brièvement vers la fin de ce chapitre.



Les trois principales erreurs que les programmeurs commettent avec RMI :

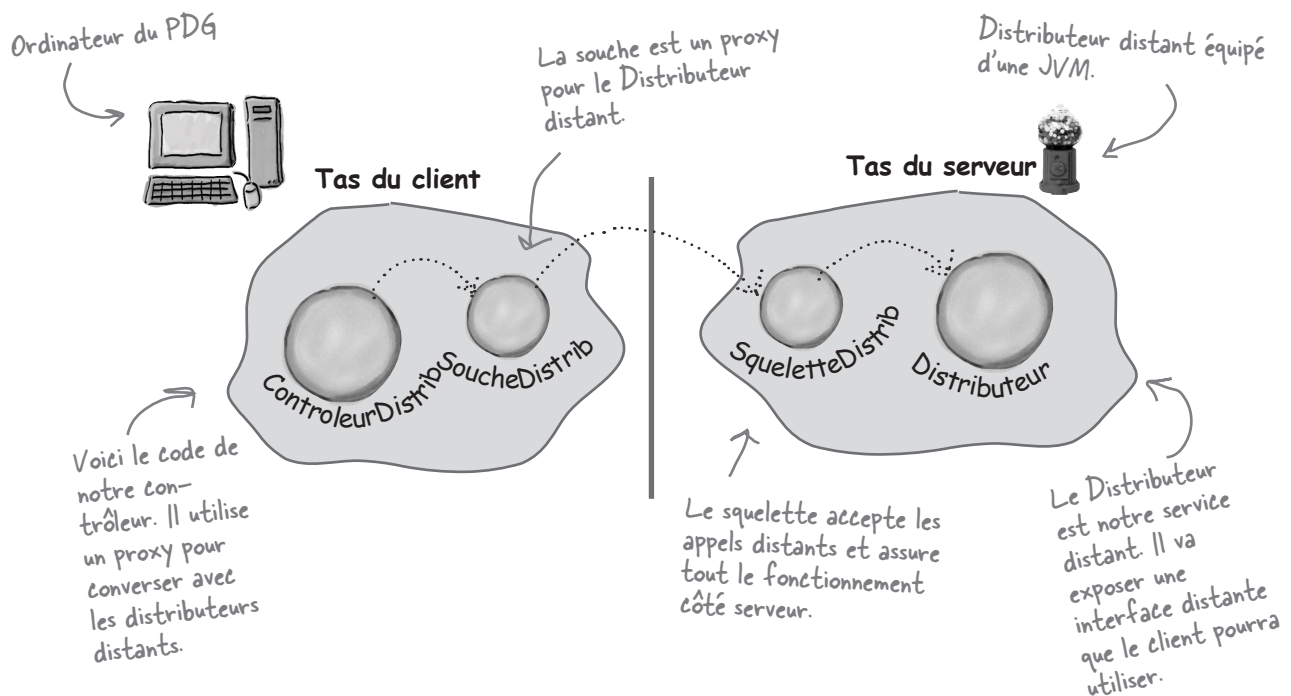
- 1) Oublier de lancer `rmiregistry` avant de démarrer le service distant (quand le service est enregistré au moyen de `Naming.rebind()`, le registre RMI doit être en train de s'exécuter !)
- 2) Oublier de rendre les arguments et les types de retour sérialisables (vous ne le saurez pas avant l'exécution parce que le compilateur ne le détectera pas).
- 3) Oublier de donner la classe souche au client.



Revenons à notre Distributeur et au proxy



Bien. Maintenant que vous possédez les bases de RMI, vous disposez des outils nécessaires pour implémenter le proxy distant du distributeur. Voyons un peu comment le Distributeur s'insère dans ce cadre :



Préparer le Distributeur à être un service distant

Pour convertir notre code afin qu'il puisse utiliser le proxy distant, la première étape consiste à permettre au Distributeur de servir les requêtes distantes des clients. Autrement dit, nous allons le transformer en service. Pour ce faire, nous devons :

- 1) Créer une interface distante pour le Distributeur. Elle va nous fournir un ensemble de méthodes qui pourront être appelées à distance.
- 2) Nous assurer que tous les types de retour de l'interface sont sérialisables.
- 3) Implémenter l'interface dans une classe concrète.

Nous allons commencer par l'interface distante.

N'oubliez pas d'importer `java.rmi.*`

```
import java.rmi.*;
```

Voici l'interface distante.

```
public interface DistributeurDistant extends Remote {
    public int getNombre() throws RemoteException;
    public String getEmplacement() throws RemoteException;
    public Etat getEtat() throws RemoteException;
}
```

Tous les types de retour sont soit primitifs soit sérialisables...

Et voilà les méthodes que nous allons prendre en charge. Chacune d'elles lance une `RemoteException`.

Nous n'avons qu'un seul type de retour qui n'est pas sérialisable : la classe `Etat`. Arrangeons cela...

```
import java.io.*;
```

Serializable est dans le package `java.io`.

```
public interface Etat extends Serializable {
    public void insererPiece();
    public void ejecterPiece();
    public void tournerPoignee();
    public void delivrer();
}
```

Nous nous contentons d'étendre l'interface `Serializable` (qui ne contient aucune méthode). Maintenant, l'état de toutes les sous-classes peut être transféré sur le réseau.

En réalité, nous n'en avons pas encore fini avec Serializable : nous avons un problème avec Etat. Comme vous vous en souvenez peut-être, chaque objet Etat maintient une référence à un distributeur pour pouvoir appeler ses méthodes et modifier son état. Mais nous ne voulons pas que tout le distributeur soit sérialisé et transféré avec l'objet Etat. La solution est simple :

```
public class EtatSansPiece implements Etat {  
    transient Distributeur distributeur;  
  
    // toutes les autres méthodes  
}
```

Dans chaque implémentation d'Etat, nous ajoutons le mot-clé transient à la déclaration de la variable d'instance distributeur. Nous indiquons ainsi à la JVM qu'elle ne doit pas sérialiser ce champ.

Nous avons déjà implémenté notre Distributeur, mais nous devons nous assurer qu'il peut se comporter en service et gérer les requêtes qui lui parviennent sur le réseau. Pour ce faire, nous devons vérifier que le Distributeur fait tout ce qu'il faut pour implémenter l'interface DistributeurDistant.

Comme vous l'avez vu dans notre détour par RMI, l'opération est très simple : il suffit d'ajouter deux ou trois lignes de code...

```
import java.rmi.*;  
import java.rmi.server.*;  
  
public class Distributeur  
    extends UnicastRemoteObject implements DistributeurDistant  
{  
    // variables d'instance  
  
    public Distributeur(String emplacement, int nombreBonbons) throws RemoteException {  
        // reste du code du constructeur  
    }  
  
    public int getNombre() {  
        return nombre;  
    }  
  
    public Etat getEtat() {  
        return etat;  
    }  
  
    public String getEmplacement() {  
        return emplacement;  
    }  
  
    // autres méthodes  
}
```

Tout d'abord, il faut importer les packages rmi.

Le Distributeur va sous-classer UnicastRemoteObject, ce qui va lui permettre de se comporter en service distant.

Le Distributeur doit également implémenter l'interface distante...

...et le constructeur doit lancer une exception, parce que la superclasse le fait.

C'est tout ! Rien d'autre ne change !

Enregistrer le service dans le registre RMI...

C'est terminé pour le service. Il suffit maintenant de faire en sorte qu'il puisse recevoir des requêtes. Tout d'abord, nous devons l'enregistrer dans le registre RMI, ce qui permettra aux clients de le localiser.

Nous allons ajouter à la classe de test un peu de code qui va s'en charger pour nous :

```
public class TestDistributeur {

    public static void main(String[] args) {
        DistributeurDistant distributeur = null;
        int nombre;
        if (args.length < 2) {
            System.out.println("TestDistributeur <nom> <stock>");
            System.exit(1);
        }

        try {
            nombre = Integer.parseInt(args[1]);

            Distributeur distributeur =
                new Distributeur(args[0], count);
            Naming.rebind("//" + args[0] + "/distributeur", distributeur);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Nous devons d'abord ajouter un bloc try/catch autour de l'instanciation du distributeur, parce que notre constructeur peut maintenant lancer des exceptions.

Nous ajoutons également l'appel de Naming.rebind(), qui publie la souche du Distributeur sous le nom distributeur.

Exécutons le test...

Exécutez ceci d'abord.

Cette commande met en service le registre RMI.

```
% rmiregistry
```

Exécutez ceci en second.

Cette commande met le Distributeur en fonction et l'enregistre dans le registre RMI.

```
% java Distributeur "Marseille" 100
```


Et maintenant, le client ControleurDistrib...

Vous souvenez-vous de ControleurDistrib ? Nous voulions le réutiliser sans devoir tout récrire pour qu'il fonctionne sur un réseau. Eh bien nous y sommes presque, mais au prix de quelques petites modifications.

```
import java.rmi.*;
```

Nous devons importer le package RMI parce que nous utilisons la classe RemoteException ci-dessous...

```
public class ControleurDistrib {  
    DistributeurDistant machine;
```

Maintenant, nous nous reposons sur l'interface distante et non sur la classe Distributeur concrète.

```
    public ControleurDistrib(DistributeurDistant machine) {  
        this.machine = machine;  
    }
```

```
    public void rapport() {
```

```
        try {
```

```
            System.out.println("Distributeur : " + machine.getEmplacement());
```

```
            System.out.println("Stock Courant : " + machine.getNombre() + " bonbons");
```

```
            System.out.println("État courant : " + machine.getEtat());
```

```
        } catch (RemoteException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

Nous devons également intercepter toutes les exceptions distantes qui pourraient survenir quand nous essayons d'invoquer des méthodes qui s'exécutent finalement sur le réseau.

Paul avait raison.
Tout va très bien
marcher !



Écrire le test du contrôleur

Nous avons maintenant tout ce qu'il faut. Il suffit d'écrire un peu de code pour que le PDG puisse contrôler quelques distributeurs :

Voici le test du contrôleur. C'est le PDG qui va l'exécuter !

Voici tous les emplacements que nous allons contrôler.

Nous créons un tableau d'emplacements, un pour chaque machine.

```
import java.rmi.*;

public class TestControleurDistrib {

    public static void main(String[] args) {
        String[] emplacement = {"rmi://rennes.distribon.com/distributeur",
                                "rmi://lille.distribon.com/distributeur",
                                "rmi://marseille.distribon.com/distributeur"};
        ControleurDistrib[] controleur = new ControleurDistrib[emplacement.length];

        for (int i=0; i < emplacement.length; i++) {
            try {
                DistributeurDistant machine =
                    (DistributeurDistant) Naming.lookup(emplacement[i]);
                controleur[i] = new ControleurDistrib(machine);
                System.out.println(controleur[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for(int i=0; i < controleur.length; i++) {
            controleur[i].rapport();
        }
    }
}
```

Nous créons aussi un tableau de contrôleurs.

Il nous faut maintenant un proxy pour chaque machine distante.

Puis nous itérons sur chaque machine et nous affichons un rapport.



Code à la loupe

```
try {
    DistributeurDistant machine =
        (DistributeurDistant) Naming.lookup(emplacement[i]);

    controleur[i] = new ControleurDistrib(machine);
} catch (Exception e) {
    e.printStackTrace();
}
```

Ce code retourne un proxy au Distributeur distant (ou lance une exception s'il n'en trouve pas).

Souvenez-vous : `Naming.lookup()` est une méthode statique du package RMI qui prend un emplacement et un nom de service et recherche dans à cet emplacement dans le registre RMI.

Une fois que nous avons un proxy pour la machine distante, nous créons un nouveau `ControleurDistrib` et nous lui transmettons la machine à contrôler.

Une autre démo pour le PDG de Distribon...

Bien. Il est temps d'assembler tout cela et de présenter une autre démo. Vérifions d'abord que quelques distributeurs exécutent le nouveau code :

Sur chaque machine, exécutez `rmiregistry` en arrière-plan ou dans une fenêtre de commandes séparée....

...puis exécuter le Distributeur, en lui donnant un emplacement et un nombre initial de bonbons.

```
Fichier Édition Fenêtre Aide Comment ?
% rmiregistry &

% java TestDistributeur rennes.distribon.com 100

Fichier Édition Fenêtre Aide Comment ?
% rmiregistry &

% java TestDistributeur lille.distribon.com 100

Fichier Édition Fenêtre Aide Comment ?
% rmiregistry &

% java TestDistributeur marseille.distribon.com 250

distributeur très populaire ! ↗
```


Et maintenant, mettons le contrôleur entre les mains du PDG.
Espérons que cette fois il sera content...

```
Fichier Édition Fenêtre Aide BonbonsCaramelsEsquimauxChocolats
% java TestControleurDistrib
Distributeur: rennes.distribon.com
Stock courant : 99 bonbons
Etat courant : en attente de pièce

Distributeur: lille.distribon.com
Stock courant : 44 bonbons
Etat courant : en attente de poignée tournée

Distributeur: marseille.distribon.com
Stock courant : 187 bonbons
Etat courant : en attente de pièce
%
```

Le contrôleur itère sur chaque machine distante et appelle ses méthodes `getEmplacement()`, `getNombre()` et `getEtat()`.

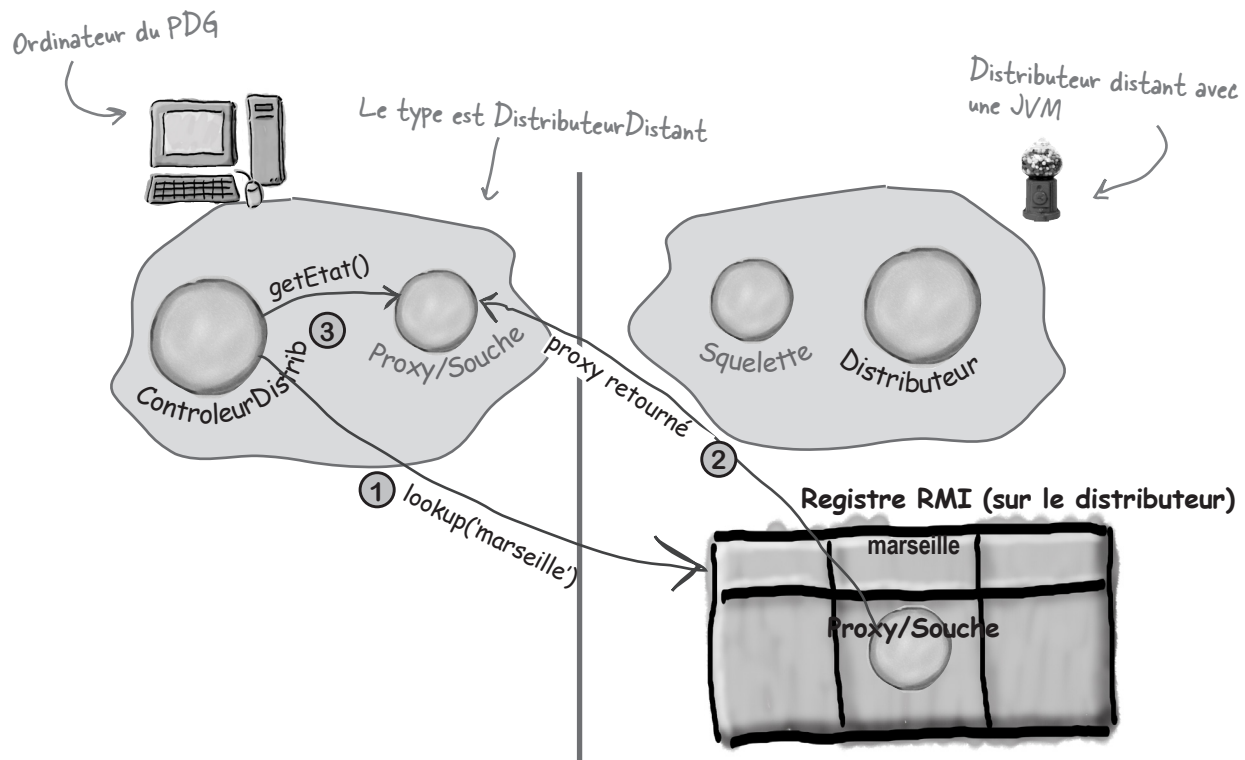
C'est stupéfiant ! Ce programme va révolutionner notre métier et balayer la concurrence !



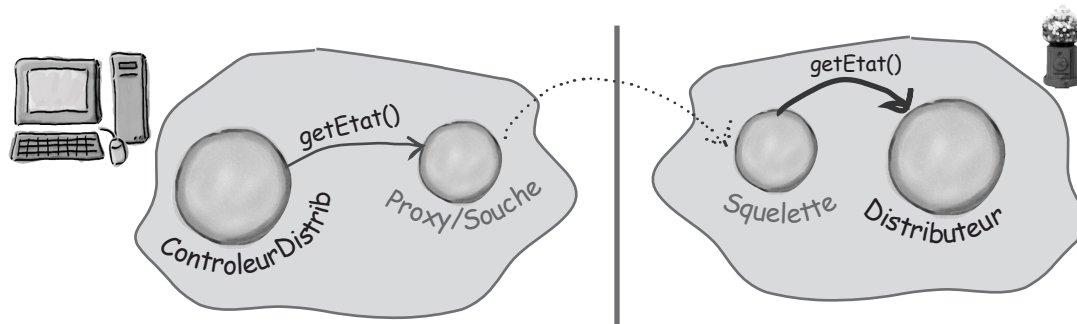
Quand on invoque des méthodes sur le proxy, un appel distant est transmis sur le réseau. Il retourne une chaîne de caractères, un entier et un objet Etat. Comme nous utilisons un proxy, le `ControleurDistrib` ne sait pas, et n'a pas besoin de savoir, que les appels sont distants (mais il doit quand même se soucier des exceptions).



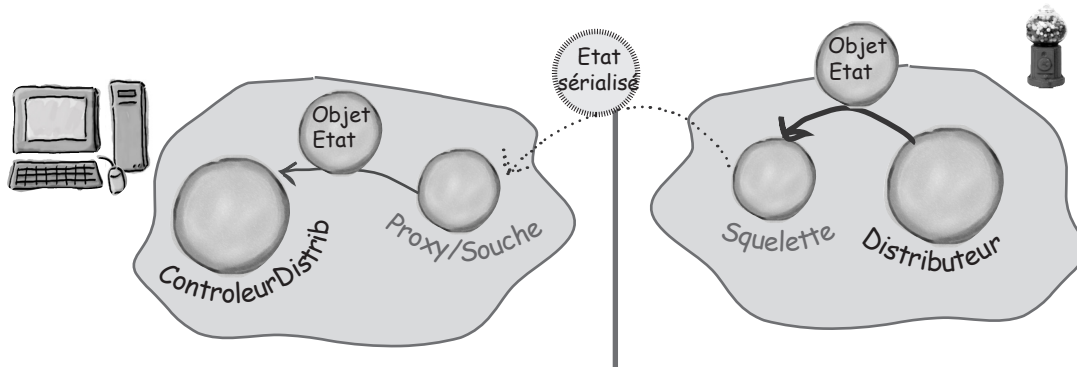
- 1 Le PDG exécute le contrôleur. Celui-ci accède d'abord aux proxies des distributeurs distants, puis il appelle `getEtat()` sur chacun (ainsi que `getNombre()` et `getEmplacement()`).



- 2 `getEtat()` est appelée sur le proxy, qui transmet l'appel au service distant. Le squelette reçoit la requête, puis la fait suivre au distributeur.



- 3 Le Distributeur retourne l'état au squelette, qui le sérialise et le retransfère au proxy via le réseau. Le proxy le désérialise et le retourne sous forme d'objet au contrôleur.



Le contrôleur n'a pas du tout changé du tout, sauf qu'il sait qu'il peut rencontrer des exceptions distantes. Il utilise également l'interface `DistributeurDistant` au lieu d'une implémentation concrète.

De même, le Distributeur implémente une autre interface et peut lancer une exception distante dans son constructeur. Ceci mis à part, le code n'a pas changé.

Il faut également un petit fragment de code pour enregistrer et localiser les souches en utilisant le registre RMI. Mais, en tout état de cause, si nous écrivions quelque chose qui doive fonctionner sur l'Internet, il nous faudrait une forme ou une autre de service de localisation.

Le pattern Proxy : définition

Nous avons déjà un certain nombre de pages derrière nous et vous avez pu constater que l'explication du Proxy distant était assez complexe. Pourtant, vous allez voir que la définition et le diagramme de classes du pattern Proxy sont en fait relativement simples. Notez que le Proxy distant est une implémentation parmi d'autres du pattern Proxy générique. Ce dernier présente en réalité un assez grand nombre de variantes dont nous parlerons plus tard. Pour l'instant, voyons les détails de la forme canonique.

Voici la définition du pattern Proxy :

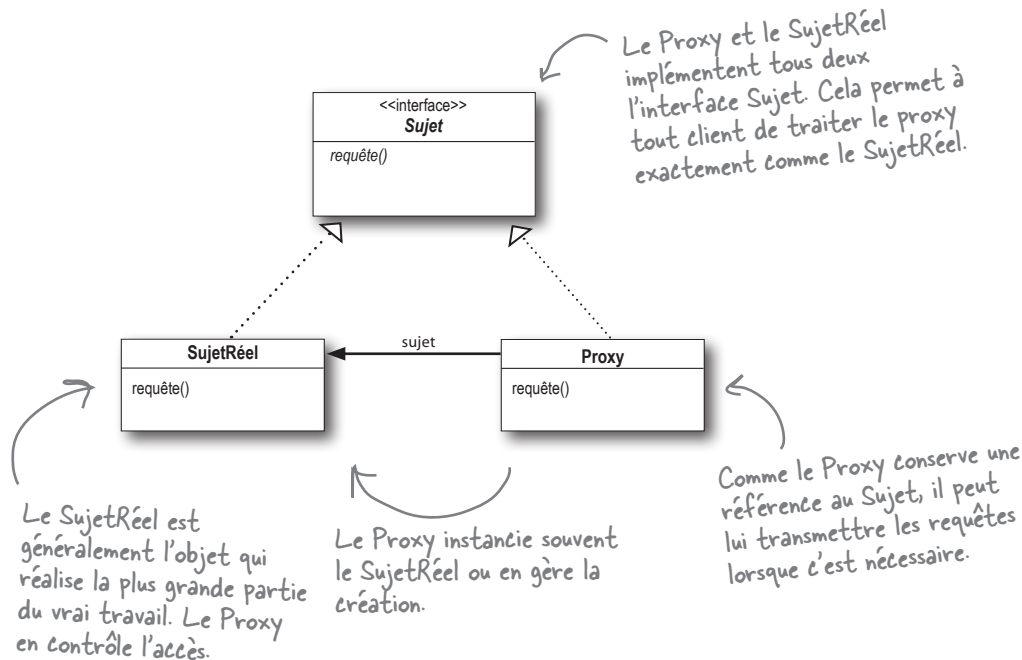
Le pattern Proxy fournit un remplaçant à un autre objet, pour en contrôler l'accès. .

Bien. Nous avons vu comment le pattern Proxy fournissait un remplaçant à un autre objet. Nous avons également décrit un proxy comme un « représentant » d'un autre objet. Mais qu'en est-il du contrôle d'accès ? Un peu étrange, non ? Ne vous inquiétez pas. Dans le cas du distributeur de bonbons, pensez simplement que le proxy contrôle l'accès à l'objet distant. Il doit contrôler l'accès parce que notre client, le contrôleur, ne sait pas comment s'adresser à un objet distant. Donc, dans un certain sens, le proxy distant contrôle l'accès pour pouvoir gérer à notre place les détails liés au réseau. Comme nous l'avons mentionné, il existe de nombreuses variantes du pattern Proxy et ces variantes tournent généralement autour de la façon dont le proxy « contrôle l'accès ». Nous y reviendrons plus tard. Pour l'instant, voici un certain nombre de types de contrôle d'accès :

- Comme nous l'avons vu, un proxy distant contrôle l'accès à un objet distant.
- Un proxy virtuel contrôle l'accès à une ressource dont la création est coûteuse.
- Un proxy de protection contrôle l'accès à une ressource en fonction de droits d'accès.

Maintenant que vous avez l'esprit du pattern, étudions le diagramme de classes...

Utilisez le pattern Proxy pour créer un objet remplaçant qui contrôle l'accès à un autre objet qui peut être distant, coûteux à créer ou qui doit être sécurisé.



Parcourons le diagramme...

Tout d'abord, nous avons un Sujet qui fournit une interface pour le SujetRéal et le Proxy. Puisqu'ils implémentent la même interface, le Proxy peut toujours être substitué au SujetRéal.

Le SujetRéal est l'objet qui fait le vrai travail. C'est lui que le Proxy représente et auquel il contrôle l'accès.

Le Proxy contient une référence au SujetRéal. Dans certains cas, le Proxy peut être responsable de la création et de la destruction du SujetRéal. Les clients interagissent avec le SujetRéal via le Proxy. Comme le Proxy et le SujetRéal implémentent la même interface (Sujet), le Proxy peut être substitué au sujet partout où ce dernier peut être utilisé. Le Proxy contrôle également l'accès au SujetRéal ; ce contrôle peut être nécessaire si le sujet s'exécute sur une machine distante, si le sujet est coûteux à créer ou si l'accès au sujet doit être protégé pour une raison quelconque.

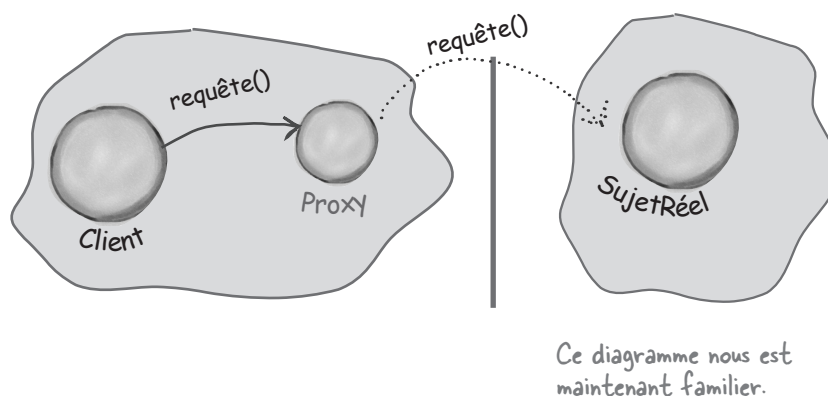
Maintenant que vous comprenez le pattern générique, voyons un certain nombre d'autres façons d'utiliser un proxy en dehors du Proxy Distant...

Prêts pour le Proxy Virtuel ?

Jusqu'ici, vous avez vu la définition du pattern Proxy et vous avez eu un aperçu d'un exemple spécifique : le *Proxy Distant*. Nous allons maintenant considérer un autre type de proxy, le *Proxy Virtuel*. Comme vous allez le découvrir, le pattern Proxy peut se manifester sous de nombreuses formes, et pourtant toutes ces formes observent approximativement la même structure. Mais pourquoi tant de formes ? Parce que le pattern Proxy peut s'appliquer à quantité de situations différentes. Voyons le Proxy Virtuel et comparons-le au Proxy Distant :

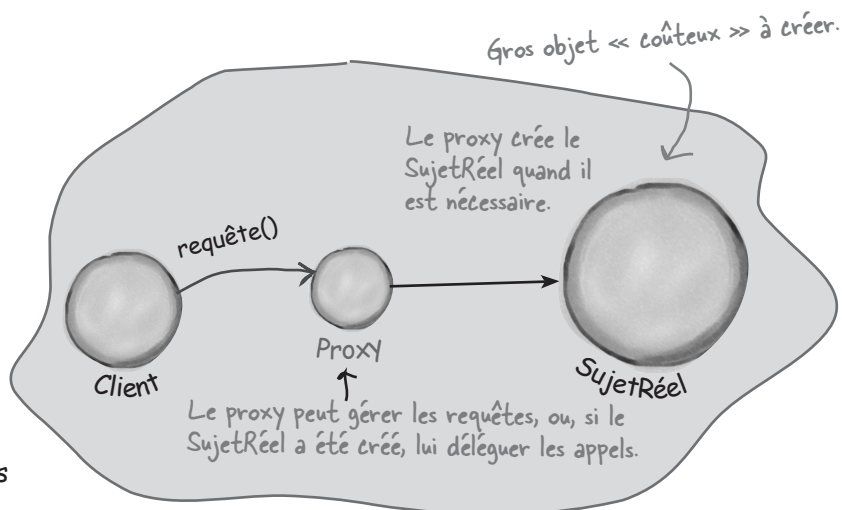
Proxy Distant

Avec Proxy Distant, le proxy se comporte comme un représentant local d'un objet qui réside dans une JVM différente. Quand on appelle une méthode sur le proxy, l'appel est transféré sur le réseau, la méthode est invoquée à distance et le résultat est retourné au proxy, puis au Client.



Proxy Virtuel

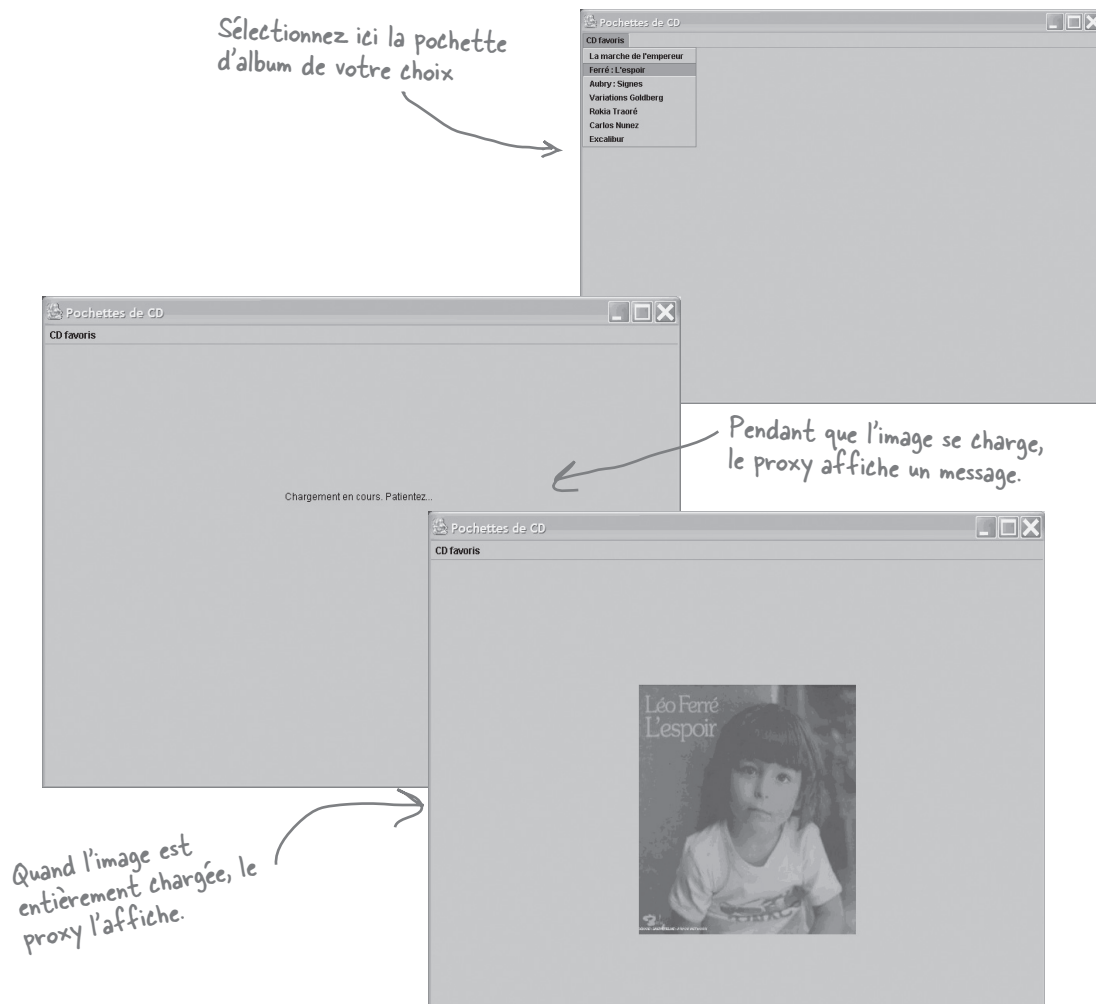
Proxy Virtuel agit en tant que représentant d'un objet dont la création peut être coûteuse. Il diffère souvent la création de l'objet jusqu'au moment où elle est nécessaire. Il sert également de substitut à l'objet avant sa création. Après quoi, le proxy délègue les requêtes directement au SujetRéel.



Afficher des pochettes de CD

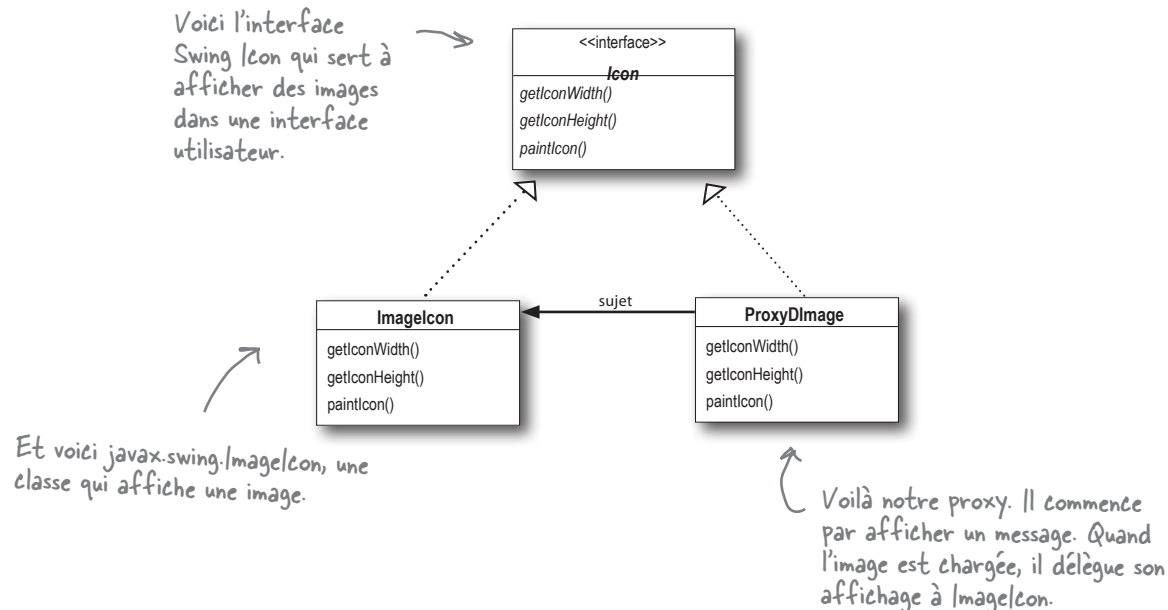
Supposons que vous vouliez écrire une application qui affiche les pochettes de vos CD favoris. Vous pouvez créer un menu des titres de CD puis récupérer les images sur un service en ligne comme Amazon.com. Si vous utilisez Swing, vous pouvez créer une icône (avec la classe `Icon`) et lui demander de télécharger l'image sur le réseau. Un seul problème : selon la charge du réseau et le débit de votre connexion, le chargement d'une pochette de CD peut prendre un certain temps. Votre application doit donc afficher quelque chose pendant que vous attendez que l'image apparaisse. De plus, vous ne voulez pas que toute l'application se mette à « ramer » pendant qu'elle attend l'image.

Une fois l'image chargée, le message doit disparaître et vous devez voir l'image. Pour ce faire, un moyen pratique consiste à utiliser un proxy virtuel. Le proxy virtuel peut prendre la place de l'icône, gérer le chargement en arrière-plan et, tant que l'image n'a pas été entièrement récupérée via le réseau, afficher « Chargement en cours, patientez... ». Une fois l'image chargée, le proxy déléguera l'affichage à l'icône.



Concevoir le proxy virtuel pour les pochettes

Avant d'écrire le code de l'affichage des pochettes, regardons le diagramme de classes. Vous allez voir qu'il est similaire à celui de notre Proxy Distant, sauf qu'ici nous utilisons le proxy pour masquer un objet dont la création est coûteuse (parce que les données destinées à l'icône transitent sur le réseau) et non un objet qui réside réellement autre part sur le réseau.



Comment ProxyDImage va fonctionner :

- ❶ **ProxyDImage crée d'abord une ImageIcon et commence à la charger à partir d'une URL donnée.**
- ❷ **Pendant qu'il récupère les octets qui composent l'image, ProxyDImage affiche « Chargement en cours, patientez... ».**
- ❸ **Lorsque l'image est entièrement chargée, ProxyDImage délègue tous les appels de méthode à ImageIcon, notamment paintIcon(), getWidth() et getHeight().**
- ❹ **Si l'utilisateur demande une nouvelle image, nous créons un nouveau proxy et nous répétons le processus.**

Écrire ProxyDImage

```

class ProxyDImage implements Icon {
    ImageIcon image;
    URL urlImage;
    Thread threadChargement;
    boolean chargement = false;

    public ProxyDImage(URL url) { urlImage = url; }

    public int getIconWidth() {
        if (image != null) {
            return image.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (image != null) {
            return image.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (image != null) {
            image.paintIcon(c, g, x, y);
        } else {
            g.drawString("Chargement en cours. Patientez...", x+300, y+190);
            if (!chargement) {
                chargement = true;
                threadChargement = new Thread(new Runnable() {
                    public void run() {
                        try {
                            image = new ImageIcon(urlImage, "Pochette de CD");
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                threadChargement.start();
            }
        }
    }
}

```

ProxyDImage implémente l'interface Icon

<<interface>> Icon
getIconWidth()
getIconHeight()
paintIcon()

image est l'icône RÉELLE, celle que nous voulons voir s'afficher quand elle aura été chargée.

Nous transmettons l'URL de l'image dans le constructeur. C'est l'image que nous voulons télécharger et afficher !

Nous retournons une largeur et une hauteur par défaut tant que l'image n'est pas chargée. Ensuite, nous retournons les valeurs réelles.

C'est là que cela devient intéressant. Cette portion de code « peint » l'image à l'écran (en déléguant à image). Mais si nous n'avons pas d'ImageIcon complètement créée, nous en créons une. Nous allons voir cela de plus près page suivante...



Code à la loupe

→ Cette méthode est appelée quand il est temps d'afficher l'image à l'écran.

```
public void paintIcon(final Component c, Graphics g, int x, int y) {
    if (image != null) {
        image.paintIcon(c, g, x, y);
    } else {
        g.drawString("Chargement en cours. Patientez...", x+300, y+190);
        if (!chargement) {
            chargement = true;
            threadChargement = new Thread(new Runnable() {
                public void run() {
                    try {
                        image = new ImageIcon(urlImage, "Pochette de CD");
                        c.repaint();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            threadChargement.start();
        }
    }
}
```

← Si nous avons déjà une image, nous lui demandons de s'afficher.

← Sinon, nous affichons un message de « chargement ».

C'est là que nous chargeons l'image RÉELLE. Notez que le chargement d'une image avec `ImageIcon` est synchrone : le constructeur d'`ImageIcon` ne se termine pas tant que l'image n'est pas chargée. Comme cela ne nous laisse aucune chance de mettre à jour les écrans et d'afficher notre message, nous allons procéder de manière asynchrone. Tous les détails dans « Code au microscope » page suivante...



Code au microscope

Si nous ne sommes pas déjà en train de récupérer l'image...

```

if (!chargement) {
    chargement = true;

    threadChargement = new Thread(new Runnable() {
        public void run() {
            try {
                image = new ImageIcon(urlImage, "Pochette de CD");
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });

    threadChargement.start();
}

```

...il est temps de commencer à le faire (au cas où vous poseriez la question, un seul thread appelle paint() : nous pouvons donc être tranquilles sur l'ordre d'exécution des threads !).

Comme nous ne voulons pas que l'interface utilisateur risque de se « planter », nous allons utiliser un autre thread pour extraire l'image.

Dans notre thread, nous instancions l'objet Icon. Son constructeur ne se terminera pas tant que l'image ne sera pas chargée.

Quand nous avons l'image, nous disons à Swing qu'il faut la « repeindre ».

Ainsi, la prochaine fois que l'affichage changera après l'instanciation d'ImageIcon, la méthode paint() affichera l'image, pas le message.

Problème de conception

Il semblerait que la classe ProxyDImage possède deux états qui sont contrôlés par des instructions conditionnelles. Connaissez-vous un autre pattern qui pourrait permettre d'arranger ce code ? Comment pourriez-vous revoir la conception de ProxyDImage ?

```
class ProxyDImage implements Icon {
    // variables d'instance et constructeur
```

```
    public int getIconWidth() {
        if (image != null) {
            return image.getIconWidth();
        } else {
            return 800;
        }
    }
```

Deux états

```
    public int getIconHeight() {
        if (image != null) {
            return image.getIconHeight();
        } else {
            return 600;
        }
    }
```

Deux états

```
    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (image != null) {
            image.paintIcon(c, g, x, y);
        } else {
            g.drawString("Chargement en cours. Patientez...", x+300, y+190);
            // reste du code
        }
    }
}
```

Deux états

Tester le programme



Code prêt à l'emploi

Bien. Il est temps de tester ce nouveau proxy virtuel dernier cri. Dans les coulisses, nous avons concocté un nouveau programme, `TestProxyDImage`, qui crée une fenêtre, puis un cadre, puis les menus et enfin notre proxy. Nous n'allons pas entrer dans tous les détails ici, mais vous pouvez toujours télécharger le code source et y jeter un coup d'œil ou vous rendre à la fin de ce chapitre où vous trouverez l'intégralité du code du Proxy Virtuel.

Voici une vue partielle du code de test :

```
public class TestProxyDImage {
    ComposantDImage composant;
    public static void main (String[] args) throws Exception {
        TestProxyDImage test = new TestProxyDImage();
    }

    public TestProxyDImage() throws Exception{

        // installer le cadre et les menus
        Icon icone = new ProxyDImage(urlInitiale);
        composant = new ComposantDImage(icone);
        cadre.getContentPane().add(composant);
    }
}
```

Ici, nous créons un proxy pour l'image et nous lui affectons une URL initiale. Chaque fois que vous effectuez une sélection dans le menu, vous obtenez un nouveau proxy.

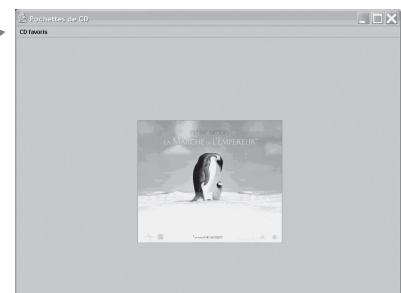
Puis nous enveloppons notre proxy dans un composant pour pouvoir l'ajouter au cadre. Le composant se chargera de la taille du proxy et autres détails similaires.

Enfin, nous ajoutons le proxy au cadre pour pouvoir l'afficher.

Et maintenant, exécutons le test :



L'exécution de `TestProxyDImage` doit afficher cette fenêtre.



Essayez ceci...

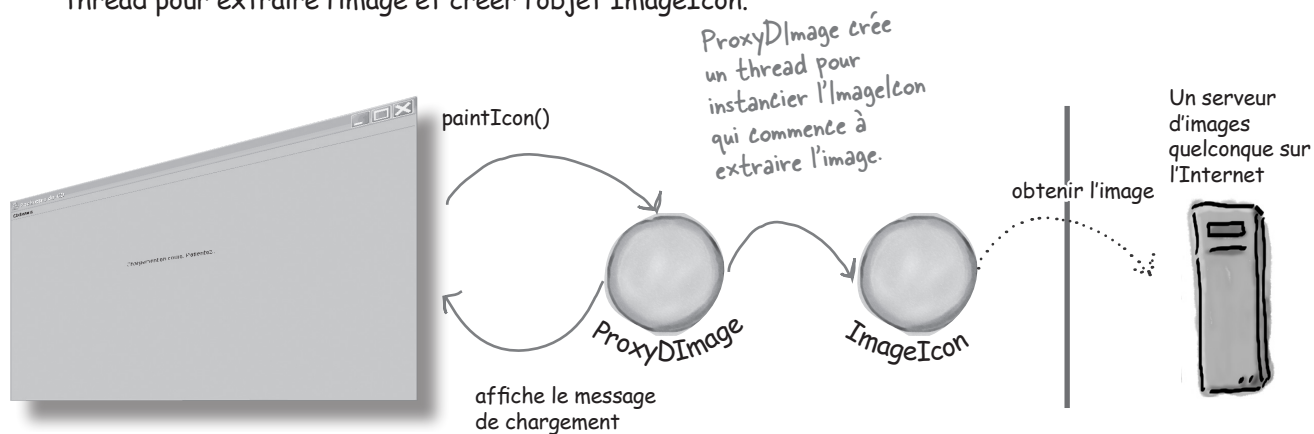
- 1 Utilisez le menu pour charger différentes pochettes de CD ; regardez le proxy afficher le message jusqu'à l'arrivée de l'image.
- 2 Redimensionnez la fenêtre pendant l'affichage du message. Remarquez que le proxy gère le chargement sans « planter » la fenêtre Swing.
- 3 Ajoutez vos propres CD favoris à `TestProxyDImage`.

Qu'avons-nous fait ?

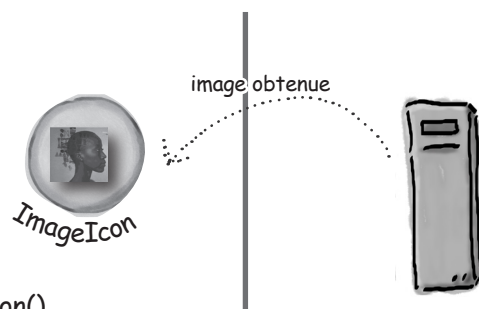
Dans les
coulisses



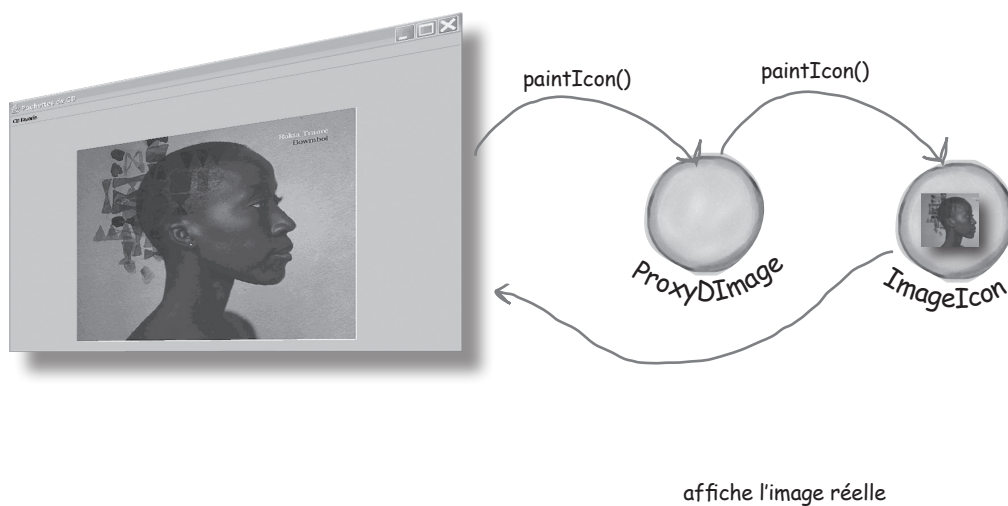
- 1 Nous avons créé un ProxyDImage pour l'affichage. La méthode paintIcon() est appelée et ProxyDImage initialise un thread pour extraire l'image et créer l'objet ImageIcon.



- 2 À un moment donné, l'image est retournée et l'ImageIcon est complètement instanciée.



- 3 Quand l'ImageIcon est créée, la prochaine fois que paintIcon() est appelée, le proxy délègue à l'ImageIcon.



il n'y a pas de questions stupides

Q: Le Proxy Distant et le Proxy Virtuel me semblent vraiment très différents ; s'agit-il vraiment du MÊME pattern ?

R: Vous trouverez beaucoup de variantes du pattern Proxy dans le monde réel. Ils présentent tous un point commun : ils interceptent une invocation de méthode que le client effectue sur le sujet. Ce niveau d'indirection nous permet de faire beaucoup de choses, notamment transmettre des requêtes à un sujet distant, procurer un représentant à un objet dont la création est coûteuse, ou, comme nous allons le voir, fournir un certain niveau de protection capable de déterminer quels clients peuvent appeler certaines méthodes. Et ce n'est qu'un début : le pattern Proxy a un grand nombre d'applications différentes et nous en aborderons quelques unes à la fin de ce chapitre.

Q: Pour moi, ProxyDImage ressemble tout à fait à un Décorateur. Je veux dire qu'en substance nous enveloppons un objet dans un autre, puis que nous déléguons les appels à l'Imagelcon. Est-ce que j'ai loupé quelque chose ?

R: Il arrive que Proxy et Décorateur aient l'air très similaires, mais leurs objectifs sont différents. Un décorateur ajoute un comportement à une classe, tandis qu'un proxy en contrôle l'accès. Vous pourriez rétorquer que le message de « chargement » ajoute un comportement.

C'est vrai en un sens, mais ce n'est pas le plus important. Le plus important, c'est que ProxyDImage contrôle l'accès à une Imagelcon. Et comment contrôle-t-il l'accès ? Vous pouvez penser en ces termes : le proxy découple le client de l'Imagelcon. S'ils étaient couplés, le client devrait attendre que l'image soit chargée avant d'afficher toute son interface. Le proxy contrôle l'accès à l'Imagelcon en affichant une autre représentation à l'écran tant qu'elle n'est pas entièrement créée. Une fois l'Imagelcon créée, le proxy autorise l'accès.

Q: Et comment faire pour que le client utilise le Proxy et non le SujetRéel ?

R: Bonne question. Une technique courante consiste à fournir un objet fabrique qui instancie et retourne le sujet. Comme il s'agit d'une méthode de fabrique, nous pouvons envelopper l'objet dans un proxy avant de le retourner. Le client ne sait jamais qu'il utilise un proxy à la place du sujet réel, et il ne s'en soucie même pas.

Q: Dans l'exemple de ProxyDImage, j'ai remarqué qu'on créait toujours un nouvel objet Imagelcon pour obtenir l'image, même si l'image avait déjà été chargée. Pourrait-on implémenter quelque chose de similaire à ProxyDImage pour récupérer les chargements antérieurs ?

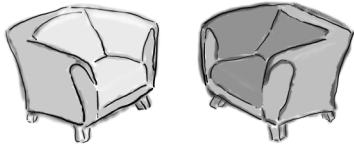
R: Vous parlez là d'une forme spécialisée de Proxy virtuel nommée Proxy de mise en cache. Un tel proxy mémorise tous les objets créés antérieurement et, si c'est possible, retourne l'objet en cache quand une requête est émise. Nous verrons cette variante du pattern Proxy ainsi que quelques autres à la fin du chapitre.

Q: Je vois bien le rapport entre Décorateur et Proxy, mais qu'en est-il d'Adaptateur ? Un adaptateur semble également très similaire.

R: Proxy et Adaptateur s'interposent tous deux entre le client et un objet et lui transmettent les requêtes. Mais n'oubliez pas qu'un adaptateur modifie l'interface de l'objet qu'il adapte alors qu'un proxy implémente la même interface.

Il existe aussi une autre similarité en rapport avec le Proxy de Protection. Un proxy de protection peut autoriser ou interdire à un client d'accéder à certaines méthodes d'un objet, en fonction du rôle de ce client. Il peut donc se contenter d'offrir une interface partielle à un client, de façon assez similaire à celle de certains adaptateurs. Nous allons jeter un coup d'œil à un proxy de protection dans quelques pages.

Face à face :



Le face à face de ce soir : **Proxy et Décorateur s'envoient des fleurs.**

Proxy

Bonjour, Décorateur. Je présume que vous êtes là parce qu'on nous confond parfois ?

Moi ? Copier vos idées ? S'il vous plaît ! Je contrôle l'accès aux objets. Vous, vous vous contentez de les décorer. Mon rôle est tellement plus important que le vôtre que ce n'est même pas drôle.

D'accord, vous n'êtes peut-être pas entièrement frivole... Mais je ne comprends toujours pas pourquoi vous pensez que je copie toutes vos idées. Je sers uniquement à représenter mes sujets, pas à les décorer.

J'ai l'impression que vous n'avez pas très bien saisi, Décorateur. Je ne me contente pas d'ajouter des comportements : je remplace mes sujets. Les clients m'utilisent comme un substitut du sujet réel, parce que je les protège contre les accès indésirables ou parce que j'empêche leur IHM de « ramer » pendant qu'ils attendent le chargement d'un gros objet, ou encore parce que je masque le fait que leurs sujets s'exécutent sur des machines distantes. Je dirais que c'est là un objectif très différent du vôtre !

Décorateur

Eh bien je crois que les gens nous confondent parce que vous vous baladez en prétendant être un pattern totalement différent alors que vous n'êtes en fait qu'un Décorateur déguisé. Et je pense que vous devriez arrêter de copier toutes mes idées.

Me *contenter* de décorer ? Vous croyez que la décoration est une tâche frivole et sans importance ? Laissez-moi vous dire quelque chose, mon pote : j'ajoute des *comportements*. C'est la chose la plus importante pour un objet – ce qu'il *fait* !

Vous avez beau parler de « représentation ». Mais si ça ressemble à un canard et que ça marche comme un canard... Regardez donc votre Proxy Virtuel : c'est juste une autre façon d'ajouter un comportement pour faire quelque chose pendant qu'on charge un gros objet coûteux, et votre Proxy Distant est un moyen de communiquer avec des objets distants pour que vos clients n'aient pas besoin de s'en occuper eux-mêmes. Comme je viens de le dire, il n'y a là que des comportements.

Dites ce que vous voudrez. J'implémente la même interface que les objets que j'enveloppe, et vous aussi.

Proxy

D'accord. Je retire ce que j'ai dit. Vous enveloppez un objet. Mais si je dis parfois de façon informelle qu'un proxy enveloppe son sujet, ce n'est pas vraiment le terme exact.

Pensez à un proxy distant... Quel est l'objet que j'enveloppe ? L'objet que je représente et dont je contrôle l'accès réside sur une autre machine ! Qu'est-ce que vous en dites ?

Bien sûr. Prenez les proxies virtuels... Pensez à l'exemple du programme qui affiche des pochettes de CD. Quand le client m'utilise comme proxy, le sujet n'existe même pas ! Qu'est-ce que je pourrais bien envelopper ?

Je ne savais pas que les décorateurs étaient aussi stupides ! Bien sûr qu'il m'arrive de créer des objets. Comment croyez-vous qu'un proxy virtuel obtient son sujet ? Et puis vous venez de souligner une autre grosse différence entre nous : nous savons tous les deux que les décorateurs se contentent de faire de l'habillage et qu'ils n'instancient jamais rien.

Eh, après cette conversation je suis convaincu que vous n'êtes qu'un proxy passif !

C'est bien rare qu'on voit un proxy s'aventurer à envelopper un objet plusieurs fois. En fait, si vous enveloppez quelque chose dix fois, vous feriez peut-être bien de réexaminer votre conception.

Décorateur

Ah oui ? Pourquoi ?

D'accord, mais on sait bien que les proxies distants sont un peu bizarres. Vous avez un autre exemple ? Cela m'étonnerait.

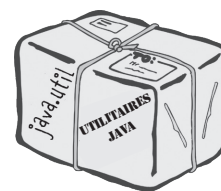
Ah ah ! Et vous allez bientôt me dire qu'en fait vous créez des objets.

Ah oui ? Instanciez-moi donc ça !

Un proxy *passif* !? J'aimerais bien vous voir envelopper récursivement un objet avec dix décorateurs et garder la tête droite en même temps.

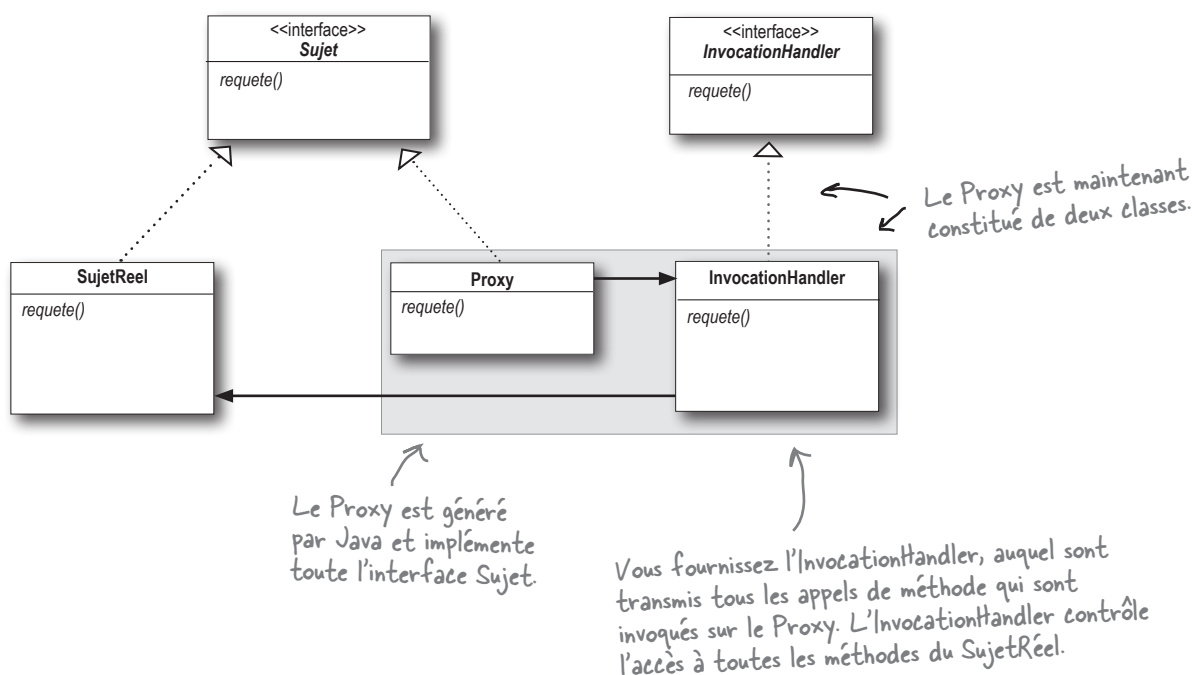
Et que dire d'un proxy qui fait semblant d'être réel alors qu'il ne fait que servir d'intermédiaire avec les objets qui font le vrai travail ? Tenez, vous me faites de la peine.

Utiliser la classe Proxy de l'API Java pour créer un proxy de protection



Java possède sa propre implémentation de Proxy dans le package `java.lang.reflect`. Grâce à ce package, Java vous permet de créer à *la volée* une classe proxy qui implémente une ou plusieurs interfaces et transmet les invocations de méthode à une classe que vous spécifiez. Comme la vraie classe proxy est créée lors de l'exécution, nous qualifions cette technologie Java de *proxy dynamique*.

Nous allons utiliser le proxy dynamique de Java pour créer notre prochaine implémentation de Proxy (un proxy de protection). Mais auparavant, jetons un rapide coup d'œil au diagramme de classes qui représente la façon dont les proxies dynamiques s'articulent. Comme la plupart des entités du monde réel, ce Proxy diffère légèrement de la définition officielle du pattern :



Comme Java crée à *votre place* la classe Proxy, il vous faut un moyen d'indiquer à celle-ci ce qu'elle doit faire. Vous ne pouvez pas placer ce code dans la classe Proxy comme nous l'avons fait tout à l'heure, parce que nous ne l'implémentons pas directement. Alors, si vous ne pouvez pas insérer ce code dans la classe Proxy, où allez-vous le placer ? Dans un InvocationHandler. La tâche de l'InvocationHandler consiste à répondre à tous les appels de méthode envoyés au proxy. Représentez-vous l'InvocationHandler comme l'objet auquel le Proxy demande de faire tout le vrai travail après qu'il a reçu les appels de méthode.

Voyons maintenant quelles sont les étapes nécessaires pour utiliser le proxy dynamique...

Rencontres à Objectville



Toute ville digne de ce nom a besoin d'un service de rencontres, non ? Vous vous êtes attelé à la tâche et vous en avez programmé un pour Objectville. Vous avez également essayé d'innover en incluant une fonctionnalité « Sexy ou non » qui offre aux participants la possibilité de se noter mutuellement. Vous pensez que cela aura pour effet de fidéliser vos clients. Et puis, cela rend les choses beaucoup plus amusantes.

Votre service s'appuie sur un *bean* *Personne* qui vous permet d'accéder aux informations concernant un individu et de les modifier :

Voici l'interface. Nous allons voir l'implémentation dans une seconde...

```
public interface BeanPersonne {

    String getNom();
    String getSexe();
    String getInterets();
    int getSexyOuNon();

    void setNom(String nom);
    void setSexe(String sexe);
    void setInterets(String interets);
    void setSexyOuNon(int note);
}
```

Voilà où nous obtenons les informations sur le nom de la personne, son sexe, ses centres d'intérêt et son niveau SexyOuNon (de 1 à 10).

setSexyOuNon() accepte un entier et l'ajoute à la moyenne actuelle de la personne.

Nous pouvons également modifier les mêmes informations au moyen de leurs appels de méthode respectifs.

Regardons maintenant l'implémentation...

L'implémentation de BeanPersonne

BeanPersonneImpl implémente l'interface BeanPersonne



```
public class BeanPersonneImpl implements BeanPersonne {  
    String nom;  
    String sexe;  
    String interets;  
    int note;  
    int nombreDeNotes = 0;
```

← Les variables d'instance.

```
    public String getNom() {  
        return nom;  
    }
```

```
    public String getSexe() {  
        return sexe;  
    }
```

← Toutes les méthodes get. Chacune d'elles retourne le contenu de la variable d'instance appropriée...

```
    public String getInterets() {  
        return interets;  
    }
```

```
    public int getSexyOuNon() {  
        if (nombreDeNotes == 0) return 0;  
        return (note/nombreDeNotes);  
    }
```

← ...sauf getSexyOuNon() qui calcule la moyenne en divisant les notes par le nombreDeNotes.

```
    public void setNom(String nom) {  
        this.nom = nom;  
    }
```

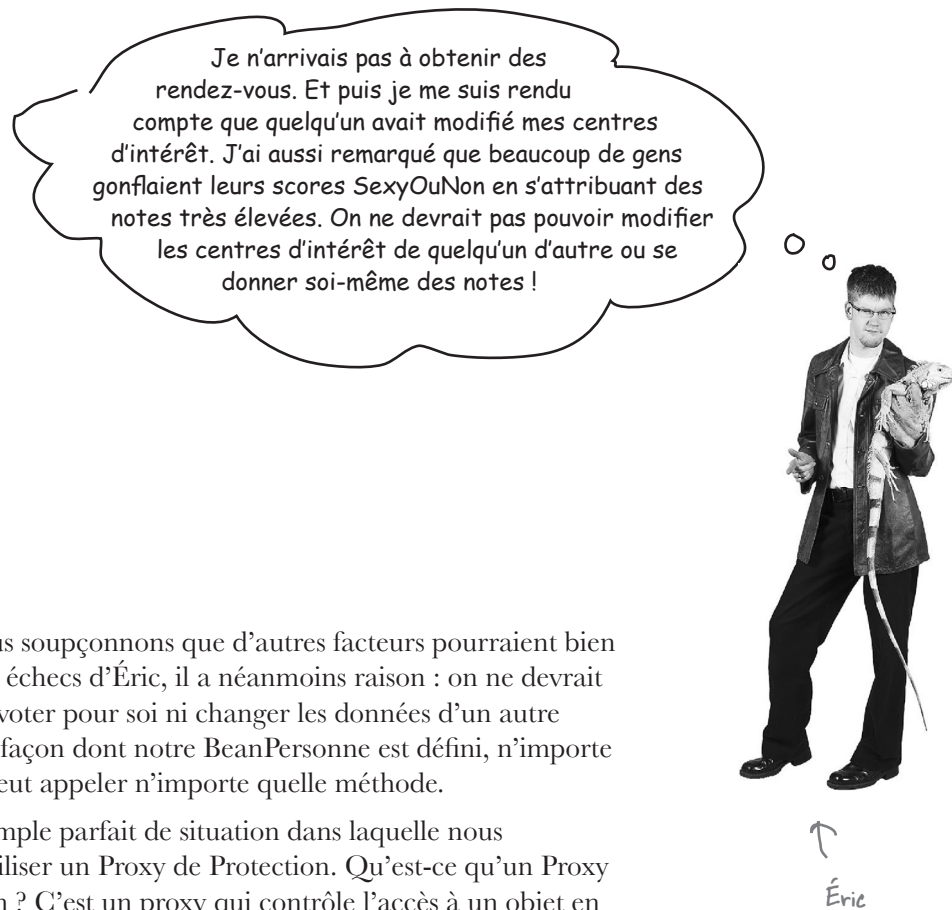
```
    public void setSexe(String sexe) {  
        this.sexe = sexe;  
    }
```

```
    public void setInterets(String interets) {  
        this.interets = interets;  
    }
```

← Et voici toutes les méthodes set qui modifient les variables d'instance correspondantes.

```
    public void setSexyOuNon(int note) {  
        this.note += note;  
        nombreDeNotes++;  
    }
```

← Enfin, la méthode setSexyOuNon() incrémente nombreDeNotes et ajoute la note au total courant.



Même si nous soupçonnons que d'autres facteurs pourraient bien expliquer les échecs d'Éric, il a néanmoins raison : on ne devrait pas pouvoir voter pour soi ni changer les données d'un autre client. De la façon dont notre `BeanPersonne` est défini, n'importe quel client peut appeler n'importe quelle méthode.

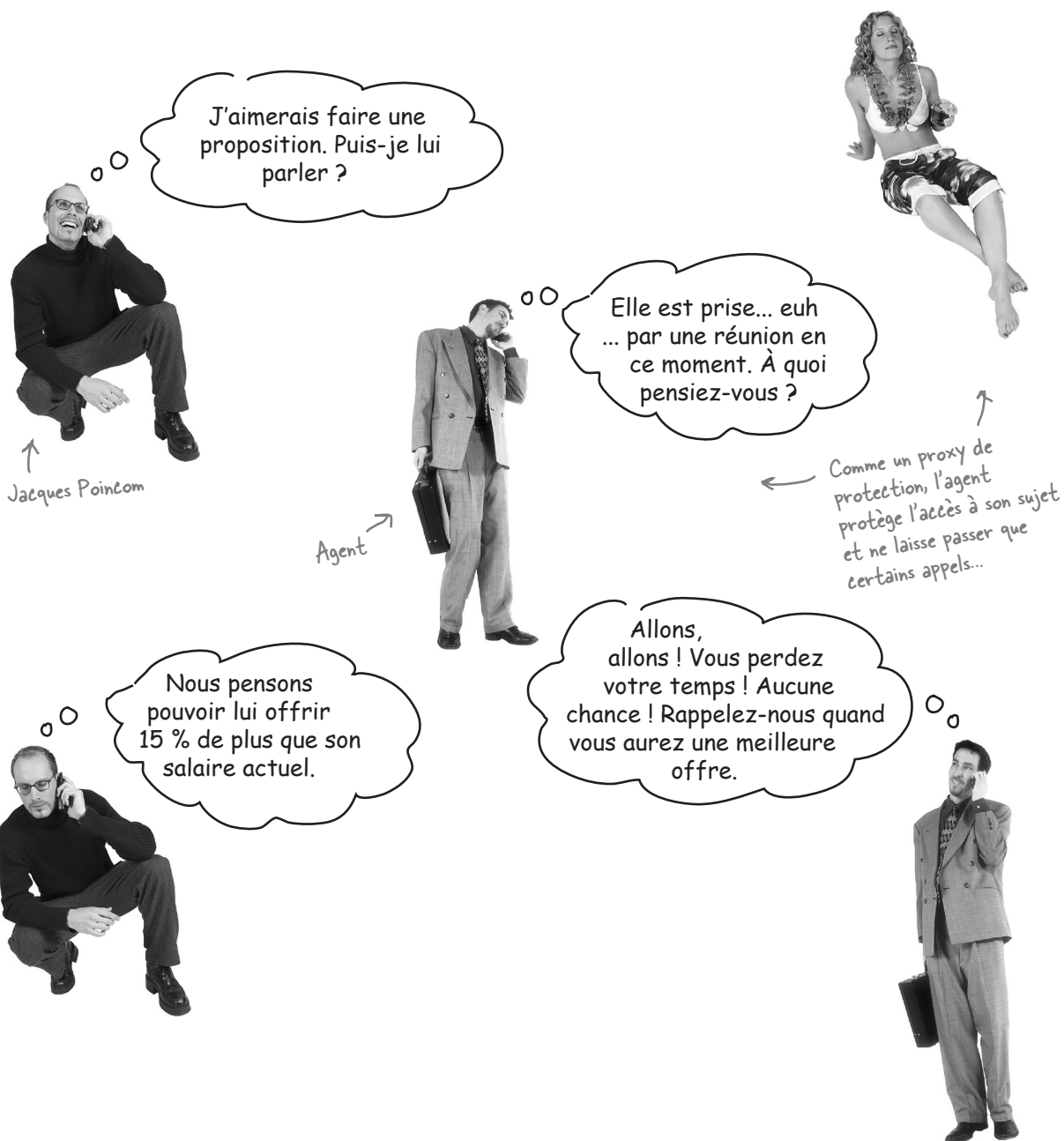
Voilà un exemple parfait de situation dans laquelle nous pourrions utiliser un Proxy de Protection. Qu'est-ce qu'un Proxy de Protection ? C'est un proxy qui contrôle l'accès à un objet en fonction de droits d'accès. Si par exemple nous avons un objet `Employé`, un proxy de protection pourrait permettre à l'employé d'appeler certaines méthodes tandis qu'un manager pourrait appeler des méthodes supplémentaires (comme `setSalaire()`) et qu'un agent des ressources humaines pourrait appeler toutes les méthodes de l'objet.

Dans notre service de rencontres, nous voulons être certains qu'un client peut modifier les informations qui le concernent tout en empêchant les autres d'en faire autant. Et nous voulons exactement le contraire avec les notes `SexyOuNon` : les autres clients doivent pouvoir attribuer la note, mais pas ce client particulier. Le `BeanPersonne` possède également un certain nombre de méthodes *get*, et, comme aucune d'entre elles ne retourne d'informations confidentielles, tout client doit pouvoir les appeler.



Comédie express : des sujets sous protection

La bulle Internet ne semble plus qu'un lointain souvenir. C'était l'époque où il suffisait de traverser la rue pour trouver un poste plus intéressant et mieux payé. Les développeurs de logiciels avaient même des agents...

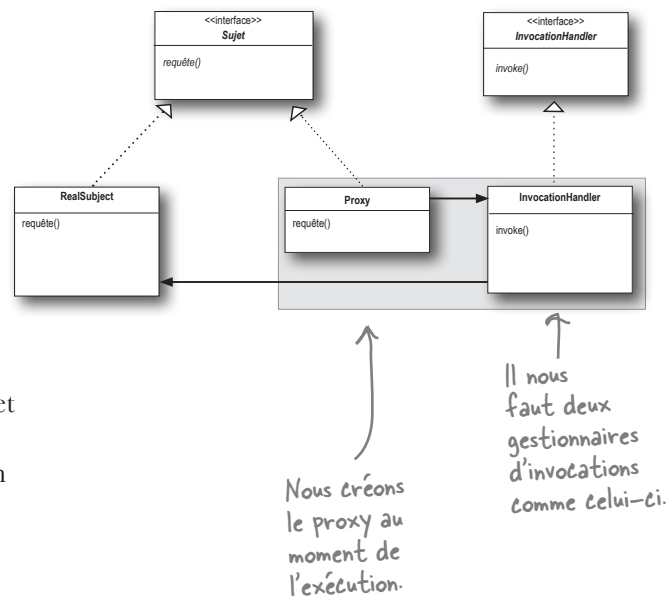


Vue d'ensemble : créer un Proxy dynamique pour le BeanPersonne

Nous avons deux problèmes à résoudre : les clients ne doivent pas pouvoir changer leur note SexyOuNon, et ils ne doivent pas pouvoir changer non plus les informations personnelles des autres clients. Pour ce faire, nous allons créer deux proxies : l'un pour accéder à votre propre objet BeanPersonne et l'autre pour accéder à l'objet BeanPersonne d'un autre client. Ainsi, les proxies pourront contrôler les requêtes qui sont émises dans chaque circonstance.

Pour créer ces proxies, nous allons utiliser le proxy dynamique de l'API Java que vous avez vu il y a quelques pages. Java va créer deux proxies pour nous et il nous suffira de fournir des « gestionnaires » qui sachent quoi faire quand une méthode est appelée sur un proxy.

Souvenez-vous de ce diagramme. Nous l'avons vu il y a quelques pages...



Étape 1 :

Créer deux **InvocationHandler**, autrement dit deux gestionnaires d'invocations.

Les InvocationHandler implémentent le comportement du proxy. Comme vous allez le voir, Java se chargera de créer la classe et l'objet proxy, et il nous faudra seulement fournir le gestionnaire qui sait ce qu'il faut faire lors d'un appel de méthode.

Étape 2 :

Écrire le code qui crée les proxies dynamiques.

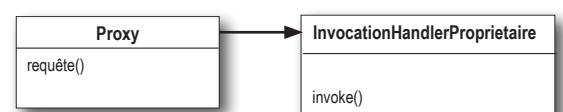
Nous devons écrire un peu de code pour générer la classe proxy et l'instancier. Nous étudierons ce code dans quelques instants.

Étape 3 :

Envelopper chaque objet BeanPersonne dans le proxy qui convient.

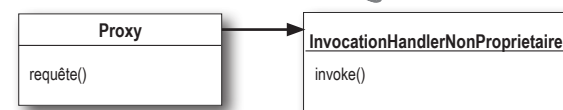
Quand nous avons besoin d'utiliser un objet BeanPersonne, soit il s'agit de l'objet du client lui-même (auquel cas nous l'appellerons le « propriétaire »), soit il s'agit de celui d'un autre client du service (et nous l'appellerons « non-propriétaire »).

Dans les deux cas, nous créons le proxy approprié pour le BeanPersonne.



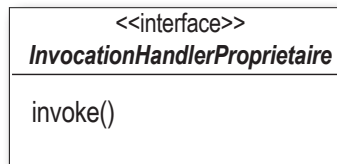
Quand un client voit son propre bean

Quand un client voit le bean de quelqu'un d'autre.



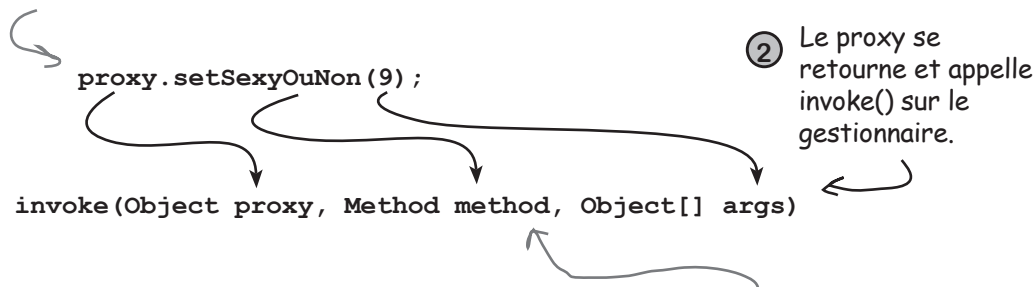
Étape 1 : créer les gestionnaires d'invocations

Nous savons que nous devons écrire deux gestionnaires d'invocations, l'un pour le propriétaire et l'autre pour le non-propriétaire. Mais qu'est-ce qu'un gestionnaire d'invocations ? Voici une façon de se le représenter : quand une méthode est appelée sur le proxy, le proxy transmet l'appel à votre gestionnaire d'invocations mais il n'appelle pas la méthode correspondante de ce dernier. Alors qu'appelle-t-il ? Jetons un coup d'œil à l'interface `InvocationHandler` :

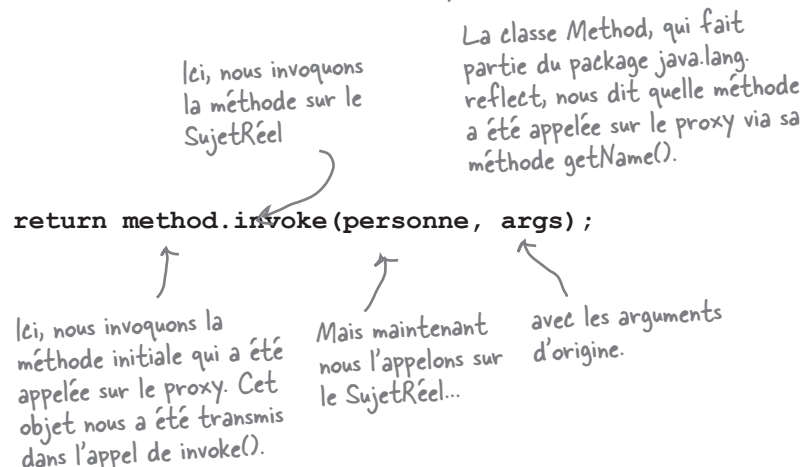


Cette interface ne contient qu'une seule méthode, `invoke()`, et, quelle que soit la méthode appelée sur le proxy, c'est cette méthode `invoke()` qui est appelée sur le gestionnaire. Voyons comment cela fonctionne :

- ① Supposons que la méthode `setSexyOuNon()` est appelée sur le proxy.



- ③ Le gestionnaire détermine ce qu'il doit faire de la requête et la transmet éventuellement au – Sujet Réel. Comment le gestionnaire décide-t-il ? Nous allons bientôt le savoir.



Créer des gestionnaires d'invocations (suite)...

Quand `invoke()` est appelée par le proxy, comment savez-vous ce qu'il faut faire de l'appel ? En général, vous examinez la méthode appelée et vous prenez des décisions en fonction de son nom et éventuellement de ses arguments. Implémentons `InvocationHandlerProprietaire` pour voir comment il fonctionne :

Comme `InvocationHandler` fait partie du package `java.lang.reflect`, nous devons l'importer.

Tous les gestionnaires d'invocations implémentent l'interface `InvocationHandler`.

Le sujet réel nous est transmis dans le constructeur et nous mémorisons une référence à celui-ci.

```
import java.lang.reflect.*;

public class InvocationHandlerProprietaire implements InvocationHandler {
    BeanPersonne personne;

    public InvocationHandlerProprietaire(BeanPersonne personne) {
        this.personne = personne;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(personne, args);
            } else if (method.getName().equals("setSexyOuNon")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(personne, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Voici la méthode `invoke()` qui est appelée chaque fois qu'une méthode est invoquée sur le proxy.

Si c'est une méthode `get`, nous continuons et nous l'appelons sur le sujet réel.

Sinon, si c'est la méthode `setSexyOuNon()`, nous empêchons de l'appeler en lançant une `IllegalAccessException`.

Comme nous sommes le propriétaire, nous avons le droit d'appeler n'importe quelle méthode `set`. Nous continuons et nous l'appelons sur le sujet réel.

Voici ce qui arrive si le sujet réel lance une exception.

Si une autre méthode quelconque est appelée, nous préférons retourner `null` au lieu de prendre un risque.



Exercice

InvocationHandlerNonProprietaire fonctionne exactement comme InvocationHandlerProprietaire, excepté qu'il *autorise* les appels de `setSexyOuNon()` et qu'il interdit ceux de toutes les autres méthodes `set`.
Allez-y ! Écrivez ce gestionnaire.

Étape 2 : créer la classe Proxy et instancier l'objet Proxy

Maintenant, il ne nous reste plus qu'à créer dynamiquement la classe proxy et à instancier l'objet proxy. Commençons par écrire une méthode qui accepte un `BeanPersonne` et qui crée un proxy « propriétaire » pour lui. Autrement dit, créons le type de proxy qui transmet ses appels de méthode à `InvocationHandlerProprietaire`. Voici le code :

Cette méthode accepte un objet `personne` (le sujet réel) et retourne un proxy qui va le remplacer. Comme le proxy a la même interface que le sujet, nous retournons un `BeanPersonne`.

Cette portion de code crée le proxy. Comme elle est un peu compliquée, nous allons l'étudier soigneusement.

Pour créer un proxy, nous utilisons la méthode statique `newProxyInstance()` de la classe `Proxy`...

← Nous lui transmettons le chargeur de classe pour notre sujet...
...et l'ensemble d'interfaces que le proxy doit implémenter...

Nous transmettons le sujet réel dans le constructeur du gestionnaire d'invocations. Si vous vous reportez deux pages en arrière, vous verrez que c'est ainsi que le gestionnaire accède au sujet réel.

...et un gestionnaire d'invocations, en l'occurrence notre `InvocationHandlerProprietaire`.

```

BeanPersonne getProxyProprietaire(BeanPersonne personne) {
    return (BeanPersonne) Proxy.newProxyInstance(
        personne.getClass().getClassLoader(),
        personne.getClass().getInterfaces(),
        new InvocationHandlerProprietaire(personne));
}

```

À vos crayons

Même si elle est un peu compliquée, la création d'un proxy dynamique n'est pas bien difficile. Pourquoi n'écrieriez-vous pas `getProxyNonProprietaire()`, qui retourne un proxy pour `InvocationHandlerNonProprietaire` ?

Encore plus fort : pouvez-vous écrire une méthode `getProxy()` qui accepte en argument un gestionnaire et une personne et qui retourne un proxy qui utilise ce gestionnaire ?

Tester le service de rencontres

Testons le service de rencontres et voyons comment il contrôle l'accès aux méthodes set selon le proxy utilisé.

```
public class TestRencontres {
    // variables d'instance
```

```
    public static void main(String[] args) {
        TestRencontres test = new TestRencontres();
        test.test();
    }
```

```
    public TestRencontres() {
        initialiserBD();
    }
```

```
    public void tester() {
        BeanPersonne luc = getPersonneDepuisBD("Luc Javabine");
        BeanPersonne proxyProprietaire = getProxyProprietaire(luc);
        System.out.println("Nom = " + proxyProprietaire.getNom());
        proxyProprietaire.setInterets("tennis, jeu de go");
        System.out.println("Intérêts fixés par le proxy propriétaire");
        try {
            proxyProprietaire.setSexyOuNon(10);
        } catch (Exception e) {
            System.out.println("Le proxy propriétaire ne peut pas modifier la note");
        }

        System.out.println("Note = " + proxyProprietaire.getSexyOuNon());
        BeanPersonne proxyNonProprietaire = getProxyNonProprietaire(luc);
        System.out.println("Nom = " + proxyNonProprietaire.getNom());
        try {
            proxyNonProprietaire.setInterets("tennis, jeu de go");
        } catch (Exception e) {
            System.out.println("Le proxy non propriétaire ne peut pas modifier les intérêts");
        }
        proxyNonProprietaire.setSexyOuNon(3);
        System.out.println("Note modifiée par le proxy non propriétaire");
        System.out.println("Note = " + proxyNonProprietaire.getSexyOuNon());
    }

    // autres méthodes comme getProxyProprietaire et getProxyNonProprietaire
}
```

La méthode main() crée simplement le test et appelle sa méthode tester().

Le constructeur initialise la base de données des personnes abonnées à notre service de rencontres.

Extrayons une personne de la base de données

...et créons un proxy propriétaire.

Appelons une méthode get puis une méthode set

puis essayons de modifier la note

Ceci ne doit pas marcher !

Créons maintenant un proxy non-propriétaire

...et appelons une méthode get suivie d'une méthode set

Ceci ne doit pas marcher !

Puis essayons de changer la note

Ceci doit marcher !

Exécuter le code...

```

Fichier Édition Fenêtre Aide Dynamiser
% java TestRencontres
Nom = Luc Javabine
Intérêts fixés par le proxy propriétaire
Le proxy propriétaire ne peut pas modifier la note
Note = 7
Nom = Luc Javabine
Le proxy non propriétaire ne peut pas modifier les intérêts
Note modifiée par le proxy non propriétaire
Note = 5
%
```

Notre proxy propriétaire autorise les accès et les modifications, sauf pour la note SexyOuNon.

Notre proxy non-propriétaire n'autorise que les accès, mais il permet également de modifier la note SexyOuNon.

il n'y a pas de questions stupides

Q: Que signifie donc le mot « dynamique » dans l'expression proxy dynamique ? Est-ce lié au fait que j'instancie le proxy et que je l'affecte à un gestionnaire au moment de l'exécution ?

R: Non, le proxy est dynamique parce que sa classe est créée au moment de l'exécution. Réfléchissez : avant que votre code ne s'exécute, il n'y a pas de classe proxy. Elle est créée à la demande à partir de l'ensemble d'interfaces que vous lui transmettez.

Q: Mon InvocationHandler m'a l'air d'un proxy bien étrange : il n'implémente aucune des méthodes de la classe à laquelle il se substitue.

R: C'est parce que l'InvocationHandler n'est pas un proxy : c'est une classe à laquelle le proxy transmet les appels de méthode pour qu'elle les gère. Le proxy lui-même est créé dynamiquement lors de l'exécution par la méthode statique Proxy.newProxyInstance().

Q: Existe-t-il un moyen quelconque de savoir si une classe est une classe Proxy ?

R: Oui. La classe Proxy a une méthode statique nommée isProxy-Class(). L'appel de cette méthode avec une classe retourne « true » si la classe est un proxy dynamique. À part cela, la classe proxy se comportera comme toute autre classe qui implémente un ensemble particulier d'interfaces.

Q: Y a-t-il des restrictions sur le type d'interfaces que je peux transmettre dans newProxyInstance() ?

R: Oui, il y en a quelques unes. Tout d'abord, il est intéressant de souligner que nous passons toujours à newProxyInstance() un tableau d'interfaces – seules les interfaces sont autorisées, pas les classes. La restriction la plus importante est que toutes les interfaces non publiques doivent appartenir au même package. De plus, vous ne pouvez pas avoir des interfaces dont les noms de méthode soient conflictuels (autrement dit, deux interfaces ayant des méthodes dont la signature est la même). Il y a encore quelques nuances mineures et vous devrez donc à un moment donné lire ce qui est écrit en petits caractères dans la javadoc.

Q: Pourquoi utilisez-vous des squelettes ? Je croyais qu'on s'en était débarrassé depuis Java 1.2.

R: Vous avez raison, nous n'avons pas réellement besoin de générer des squelettes. À partir de Java 1.2, RMI peut transmettre directement les appels des clients au service distant en utilisant l'introspection au moment de l'exécution. Mais nous préférons montrer les squelettes, parce que, du point de vue conceptuel, cela aide à comprendre qu'il y a quelque chose dans les coulisses qui permet à la souche du client et au service distant de communiquer.

Q: J'ai entendu dire qu'en Java 5, je n'avais même plus besoin de générer des souches. Est-ce que c'est vrai ?

R: Certainement. En Java 5, RMI et le Proxy dynamique sont fusionnés, et les souches sont maintenant générées dynamiquement en utilisant le Proxy dynamique. La souche de l'objet distant est une instance de java.lang.reflect.Proxy (avec un gestionnaire d'invocations) qui est générée automatiquement et qui gère tous les détails de la transmission des appels de méthodes locaux du client à l'objet distant. En conséquence, vous n'avez plus du tout besoin d'exécuter rmic : tout ce qu'il faut pour que le client converse avec l'objet distant est géré de manière transparente à votre place.

?

Qui fait quoi ?

Faites correspondre chaque pattern avec sa description :

Pattern	Description
Décorateur	Enveloppe un autre objet et lui fournit une interface différente
Façade	Enveloppe un autre objet et lui fournit un comportement supplémentaire
Proxy	Enveloppe un autre objet pour en contrôler l'accès
Adaptateur	Enveloppe un groupe d'objets pour simplifier leur interface

Le zoo des proxies

Bienvenue au zoo d'Objectville !

Vous connaissez maintenant les proxies distants, virtuels et de protection, mais vous rencontrerez aussi dans la nature toutes sortes de mutations de ce pattern. Ici, dans le coin Proxy de notre zoo, nous conservons une jolie collection de proxies sauvages que nous avons capturés pour que vous puissiez les étudier.

Notre tâche n'est pas terminée : nous sommes convaincus que vous verrez d'autres variantes de ce pattern dans le monde réel et vous pouvez nous aider à ajouter d'autres proxies au catalogue. Mais jetons un coup d'œil à la collection existante :



Proxy pare-feu
contrôle l'accès
à un ensemble
de ressources

réseau et protège le sujet des
clients « malveillants ».

Habitat : souvent observé aux environs de systèmes
pare-feu d'entreprise.

Aidez-le à trouver un habitat

Référence intelligente
réalise des opérations
supplémentaires lorsqu'un
objet est référencé, telles
que le comptage du nombre
de références.



Proxy de mise en cache
permet de stocker
temporairement les résultats
d'opérations coûteuses. Il peut

également permettre à plusieurs clients de
partager les résultats afin de réduire les
temps de calcul ou de remédier à la latence
du réseau.

Habitat : souvent rencontré dans les serveurs web ainsi que
dans les systèmes de publication et de gestion de contenus.

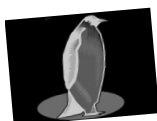
Proxy de synchronisation
permet à plusieurs threads d'accéder de façon sécurisée à un sujet.



Vu en train de traîner dans des JavaSpaces, où il contrôle l'accès synchronisé à un ensemble sous-jacent d'objets dans un environnement distribué.

Aidez-le à trouver un habitat

Proxy de masquage de la complexité.
Comme son nom l'indique, masque la complexité d'un ensemble de classes et en contrôle l'accès. Parfois appelé Proxy Façade pour des raisons évidentes. Il diffère du pattern Façade en ce que le proxy contrôle l'accès alors que la Façade se contente de fournir une autre interface.



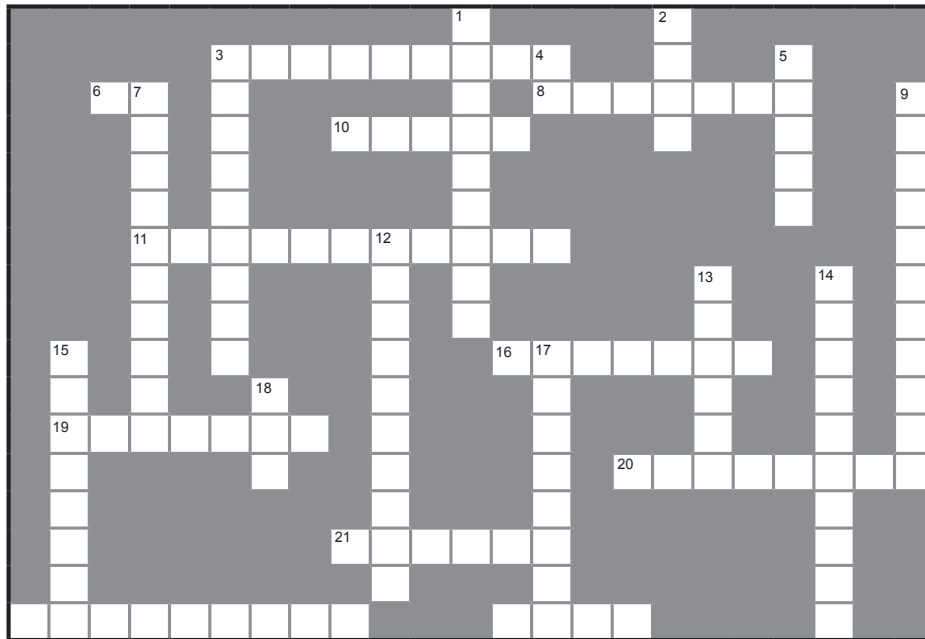
Proxy « Copy-On-Write »
(copie-calque) contrôle la copie d'un objet en la différant jusqu'à ce qu'un client en ait besoin. C'est une variante du Proxy Virtuel.

Habitat : observé au voisinage du CopyOnWriteArrayList de Java 5.

Notes de terrain : ajoutez ici vos observations d'autres proxies à l'état sauvage :



C'était un très LONG chapitre. Pourquoi ne pas vous détendre en faisant un mots-croisés avant de le terminer ?

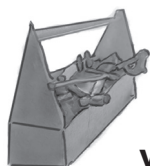


Horizontalement

3. La classe du proxy _____ est créée au moment de l'exécution.
6. Vous pouvez en afficher les pochettes.
8. Sans _____, pas de services.
10. Ce que contrôle un proxy.
11. Il faut l'exécuter pour enregistrer un service.
16. Toute _____ est transmise au gestionnaire d'invocations.
19. Pas vraiment réel.
20. Fournis par un serveur.
21. C'est ainsi que RMI nomme le proxy.
22. Certains constructeurs n'en ont pas.
23. Manque de rendez-vous.

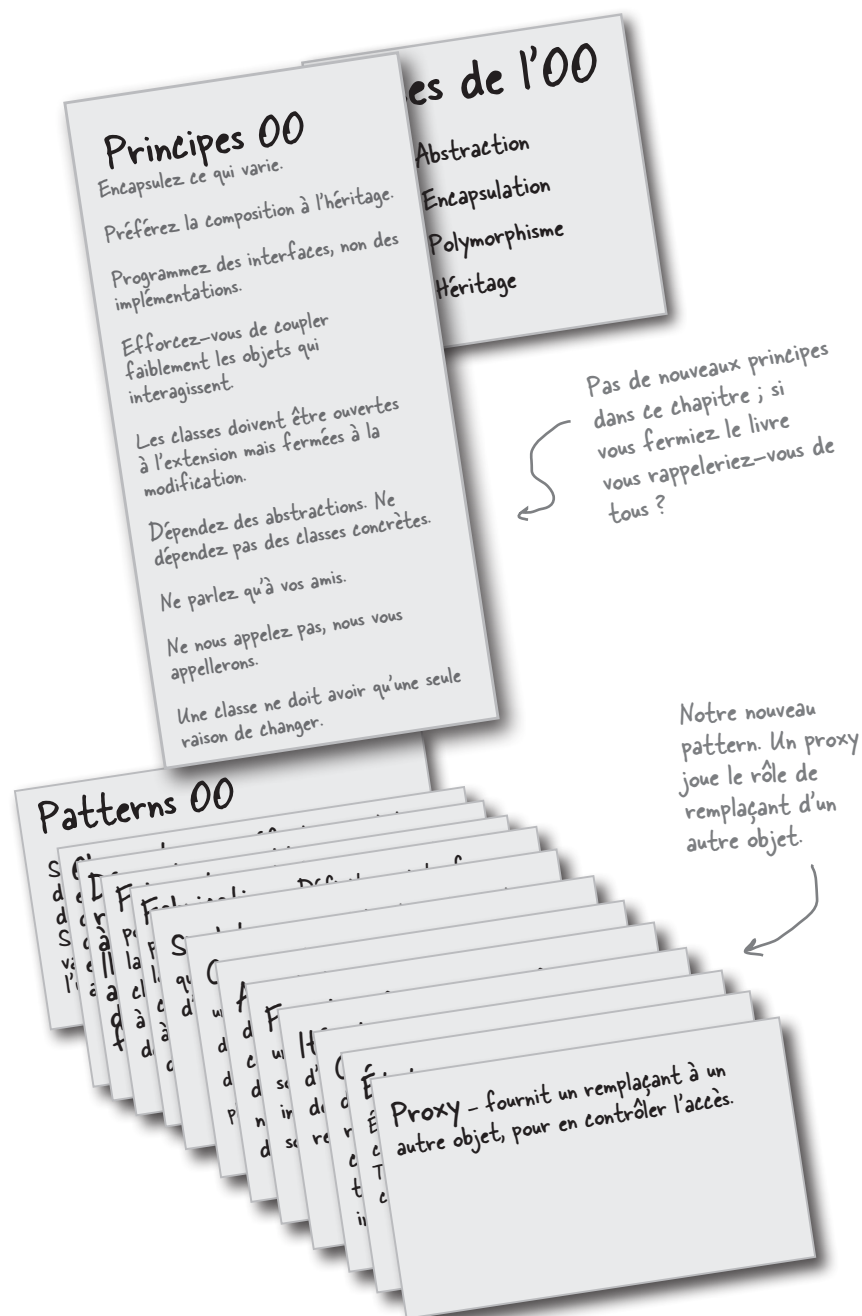
Verticalement

1. Dans RMI, auxiliaire du serveur.
2. Pas C++, pas C#, _____.
3. Distributeur de distributeurs.
4. Entrées et sorties familières.
5. Substitut.
7. Similaire à Proxy, son intention est différente.
9. Rencontres.
12. Remote Method _____.
13. Nous en avons fait un pour découvrir RMI.
14. Protège les méthodes des appels non autorisés.
15. Appeler.
17. On en voit un page 469.
18. On y rencontre des foules de proxies.



Votre boîte à outils de concepteur

Votre boîte à outils est presque pleine. Vous êtes maintenant prêt à résoudre presque tous les problèmes de conception qui pourraient se présenter.



POINTS D'IMPACT

- Le pattern Proxy fournit un représentant à un autre objet pour contrôler les accès du client. Il existe de nombreuses façons de gérer ces accès.
- Un Proxy Distant gère les interactions entre un client et un objet distant.
- Un Proxy Virtuel contrôle l'accès à un objet qui est coûteux à instancier.
- Un Proxy de protection contrôle l'accès aux méthodes d'un objet en fonction de l'appelant.
- Il existe de nombreuses autres variantes du pattern Proxy, notamment des proxies de mise en cache, des proxies de synchronisation, des proxies pare-feu, des proxies « copy-on-write », etc.
- Du point de vue structurel, Proxy est similaire à Décorateur, mais leurs objectifs diffèrent.
- Le pattern Décorateur ajoute un comportement à un objet, alors qu'un Proxy contrôle l'accès.
- Java dispose d'un support intégré de Proxy. Il permet de créer une classe proxy dynamique à la demande et de transmettre tous les appels qui lui sont envoyés à un gestionnaire de votre choix.
- Comme tous les wrappers, les proxies augmentent le nombre de classes et d'objets dans vos conceptions.



Solutions des exercices



Exercice

InvocationHandlerNonProprietaire fonctionne exactement comme InvocationHandlerProprietaire, excepté qu'il autorise les appels de setSexyOuNon() et qu'il interdit ceux de toutes les autres méthodes set.

Allez-y ! Écrivez ce gestionnaire :

```
import java.lang.reflect.*;

public class InvocationHandlerNonProprietaire implements InvocationHandler {
    BeanPersonne personne;

    public InvocationHandlerNonProprietaire(BeanPersonne personne) {
        this.personne = personne;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(personne, args);
            } else if (method.getName().equals("setSexyOuNon")) {
                return method.invoke(personne, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Problème de conception

Il semblerait que la classe ProxyDImage possède deux états qui sont contrôlés par des instructions conditionnelles. Connaissez-vous un autre pattern qui pourrait permettre d'arranger ce code ? Comment pourriez-vous revoir la conception de ProxyDImage ?

Utiliser le pattern État, implémenter deux états, ImageChargée et ImageNonChargée. Puis placer le code des instructions conditionnelles dans leurs états respectifs. Commencer dans l'état ImageNonChargée puis effectuer la transition vers ImageChargée une fois que l'ImageIcon a été récupérée.



Solutions des exercices



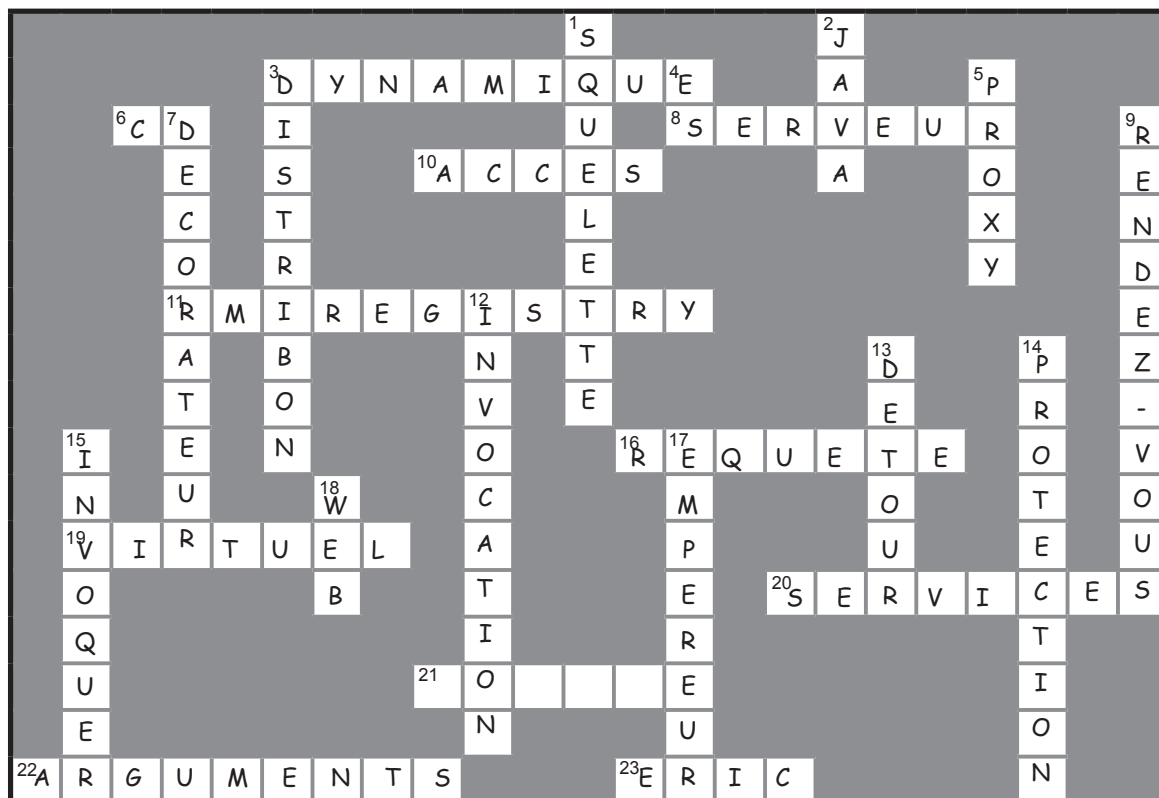
À vos crayons

Même si elle est un peu compliquée, la création d'un proxy dynamique n'est pas bien difficile. Pourquoi n'écririez-vous pas `getProxyNonProprietaire()`, qui retourne un proxy pour `InvocationHandlerNonProprietaire` ?

```
BeanPersonne getProxyNonProprietaire(BeanPersonne personne) {

    return (BeanPersonne) Proxy.newProxyInstance(
        personne.getClass().getClassLoader() ,
        personne.getClass().getInterfaces() ,
        new InvocationHandlerNonProprietaire(personne)) ;

}
```





Code prêt à
l'emploi

Le code qui affiche les pochettes de CD

```
package tetepremiere.proxy.proxyvirtuel;
import java.net.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class TestProxyDImage {
    ComposantDImage composant;
    JFrame cadre = new JFrame("Pochettes de CD");
    JMenuBar barreDeMenus;
    JMenu menu;
    Hashtable cd = new Hashtable();

    public static void main (String[] args) throws Exception {
        TestProxyDImage test = new TestProxyDImage();
    }

    public TestProxyDImage() throws Exception{
        cd.put("La marche de l'empereur", " http://images-eu.amazon.com/images/P/
B00070FX3M.08.LZZZZZZZ.jpg");
        cd.put("Ferré : L'espoir", " http://images-eu.amazon.com/images/P/B00008ZPD3.08.
LZZZZZZZ.jpg ");
        cd.put("Excalibur", "http://images-eu.amazon.com/images/P/B00004VRKV.08.LZZZZZZZ.
jpg");
        cd.put("Carlos Nunez", "http://images-eu.amazon.com/images/P/B000063WSL.08.
LZZZZZZZ.jpg");
        cd.put("Variations Goldberg", "http://images-eu.amazon.com/images/P/B000025NYA.08.
LZZZZZZZ.jpg");
        cd.put("Aubry : Signes", " http://images-eu.amazon.com/images/P/B0000085FR.08.
LZZZZZZZ.jpg ");
        cd.put("Rokia Traoré", "http://images-eu.amazon.com/images/P/B0002M5T9I.01.
LZZZZZZZ.jpg");

        URL urlInitiale = new URL((String)cd.get("La marche de l'empereur"));
        barreDeMenus = new JMenuBar();
        menu = new JMenu("CD favoris");
        barreDeMenus.add(menu);
        cadre.setJMenuBar(barreDeMenus);
    }
}
```



```

for(Enumeration e = cd.keys(); e.hasMoreElements();) {
    String nom = (String)e.nextElement();
    JMenuItem menuItem = new JMenuItem(nom);
    menu.add(menuItem);
    menuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            composant.setIcon(new ProxyDImage(getUrlCD(event.getActionCommand())));
            cadre.repaint();
        }
    });
}

// installer le cadre et les menus

Icon icone = new ProxyDImage(urlInitiale);
composant = new ComposantDImage(icone);
cadre.getContentPane().add(composant);
cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
cadre.setSize(800,600);
cadre.setVisible(true);

}
URL getUrlCD(String nom) {
    try {
        return new URL((String)cd.get(nom));
    } catch (MalformedURLException e) {
        e.printStackTrace();
        return null;
    }
}
}
}

```




Code prêt à l'emploi

Le code qui affiche les pochettes de CD, suite...

```
package tetepremiere.proxy.proxyvirtuel;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ProxyDImage implements Icon {
    ImageIcon image;
    URL urlImage;
    Thread threadChargement;
    boolean chargement = false;

    public ProxyDImage(URL url) { urlImage = url; }

    public int getIconWidth() {
        if (image != null) {
            return image.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (image != null) {
            return image.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (image != null) {
            image.paintIcon(c, g, x, y);
        } else {
            g.drawString("Chargement en cours. Patientez...", x+300, y+190);
            if (!chargement) {
                chargement = true;

                threadChargement = new Thread(new Runnable() {
                    public void run() {
                        try {
                            image = new ImageIcon(urlImage, "Pochette de CD");
                            c.repaint();
                        } catch (Exception e) {
```



```

        e.printStackTrace();
    }
}
});
threadChargement.start();
}
}
}
}
}

```

```

package tetepremiere.proxy.proxyvirtuel;

import java.awt.*;
import javax.swing.*;

class ComposantDImage extends JComponent {
    private Icon icone;

    public ComposantDImage(Icon icone) {
        this.icone = icone;
    }

    public void setIcon(Icon icone) {
        this.icone = icone;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icone.getIconWidth();
        int h = icone.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icone.paintIcon(this, g, x, y);
    }
}

```


12 patterns composés

✦ ***Patterns*** ✦ ***de patterns*** ✦



Qui aurait cru que les patterns étaient capables de collaborer ? Vous avez déjà été témoin de l'animosité qui règne dans les face-à-face (et vous n'avez pas vu le combat à mort que l'éditeur nous a forcés à retirer de ce livre *). Alors qui aurait pu penser que des patterns pourraient finir par s'entendre ? Eh bien, croyez-le ou non, certaines des conceptions OO les plus puissantes font appel à plusieurs patterns. Apprêtez-vous à passer à la vitesse supérieure : il est temps d'étudier les patterns composés.

* si vous êtes intéressé, envoyez-nous un e-mail.

Collaboration



L'une des meilleures façons d'utiliser les patterns est de les sortir de leur cage afin qu'ils puissent interagir avec leurs congénères. Plus vous utiliserez les patterns, plus vous les verrez apparaître ensemble dans vos conceptions. Nous disposons d'un nom spécial pour désigner un ensemble de patterns qui collaborent dans une conception qui peut s'appliquer à de nombreux problèmes : nous disons que ce sont des *patterns composés*. C'est juste, nous allons maintenant parler de patterns composés de patterns !

Vous rencontrerez beaucoup de patterns composés dans le monde réel. Maintenant que vous avez assimilé les concepts, vous allez voir qu'il s'agit simplement de patterns qui collaborent, ce qui les rend plus faciles à comprendre.

Nous allons commencer ce chapitre en retrouvant les sympathiques canards de notre simulateur de canards, SuperCanard. Ce n'est que justice qu'ils réapparaissent au moment où nous allons combiner des patterns. Après tout, ils nous ont accompagnés tout au long de ce livre, et ils se sont montrés bons joueurs en prenant part à de nombreuses démonstrations de patterns. Les canards vont vous aider à comprendre comment les patterns peuvent collaborer pour parvenir à une solution commune. Mais le seul fait de combiner des patterns ne signifie pas pour autant que notre solution qualifie un pattern composé. Pour cela, cette solution doit être suffisamment généraliste pour être applicable à de nombreux problèmes. C'est pourquoi nous verrons dans la seconde moitié de ce chapitre un vrai pattern composé : Monsieur Modèle-Vue-Contrôleur en personne. Si vous n'en avez jamais entendu parler, ce sera l'occasion de faire connaissance et vous verrez que c'est l'un des patterns les plus puissants que vous puissiez ajouter à votre boîte à outils de concepteur.

On combine souvent des patterns pour obtenir une solution de conception.

Un pattern composé combine deux ou plusieurs patterns pour mettre au point la solution d'un problème général ou récurrent.

Réunion de famille

Comme vous venez de l'apprendre, nous allons de nouveau travailler avec les canards. Cette fois, ils vont vous montrer comment des patterns peuvent coexister et même coopérer au sein d'une même solution.

Nous allons reconstruire notre simulateur de canards *ex nihilo* et lui ajouter un certain nombre de fonctionnalités intéressantes en utilisant un groupe de patterns. OK, allons-y..

① D'abord, nous allons créer une interface **Cancaneur**.

Comme nous vous l'avons dit, nous partons de zéro. Cette fois, les Canards vont implémenter une interface **Cancaneur**. Ainsi, nous saurons quels sont les objets du simulateurs qui peuvent `cancaner()`, comme les colverts, les mandarins, les appelants... et il se pourrait même que le canard en plastique viennent pointer son nez.

```
public interface Cancaneur {
    public void cancaner();
}
```

Les Cancaneurs ne font qu'une chose et le font bien : ils cancanent !

② Maintenant, créons des Canards qui implémentent **Cancaneur**.

À quoi bon une interface sans classes pour l'implémenter ? Il est temps de créer quelques canards concrets (mais pas du genre « nains de jardin », si vous voyez ce que nous voulons dire).

```
public class Colvert implements Cancaneur {
    public void cancaner() {
        System.out.println("Coincoin");
    }
}
```

Notre Colvert standard.

```
public class Mandarin implements Cancaneur {
    public void cancaner() {
        System.out.println("Coincoin");
    }
}
```

Il faut bien varier les espèces si nous voulons que ce simulateur soit intéressant.

Ce ne serait pas drôle si nous n'ajoutions pas quelques types de Canards.

Vous souvenez-vous de la dernière fois ? Nous avions des appelants (comme ceux que les chasseurs utilisent et qui, de toute évidence, cancanent) et des canards en plastique.

```
public class Appelant implements Cancaneur {
    public void cancaner() {
        System.out.println("Couincouin");
    }
}
```

Un Appelant qui cancaner mais qui n'a pas l'air tout à fait vrai.

```
public class CanardEnPlastique implements Cancaneur {
    public void cancaner() {
        System.out.println("Couic");
    }
}
```

Un CanardEnPlastique qui couine quand il cancaner.

③ Bien. Nous avons nos canards, il nous faut maintenant le simulateur.

Concoctons un simulateur qui crée quelques canards et vérifions qu'ils cancanent...

```
public class SimulateurDeCanards {
    public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        simulateur.simuler();
    }
}
```

Voici notre méthode main() qui déclenche l'exécution..

```
void simuler() {
    Cancaneur colvert = new Colvert();
    Cancaneur mandarin = new Mandarin();
    Cancaneur appelant = new Appelant();
    Cancaneur canardEnPlastique = new CanardEnPlastique();
}
```

Comme il nous faut quelques canards, nous créons un Cancaneur de chaque sorte...

```
System.out.println("\nSimulateur de canards");
```

```
simuler(colvert);
simuler(mandarin);
simuler(appelant);
simuler(canardEnPlastique);
}
```

... puis nous simulons chacun d'eux.

Ici, nous surchargeons la méthode simuler() pour ne simuler qu'un seul canard.

```
void simuler(Cancaneur canard) {
    canard.cancaner();
}
```

Et là, nous laissons la magie du polymorphisme opérer : quel que soit le genre de Cancaneur transmis, la méthode simuler() lui demande de cancaner.

Rien de bien transcendant, mais
attendez qu'on ajoute des patterns !



```
Fichier Édition Fenêtre Aide PeutMieuxFaire
% java SimulateurDeCanards

Simulateur de canards
Coincoin
Coincoin
Couincouin
Couic

%
```

Ils implémentent tous la même interface
Caneaneur, mais leurs implémentations leur
permettent de cancaner à leur guise.

On dirait que tout fonctionne. Jusque là, c'est un sans-faute.

④ Quand il y a des canards, les oies ne sont pas loin.

Un palmipède peut en cacher un autre. Voici une Oie qui était en train de traîner autour du simulateur.

```
public class Oie {
    public void cacarder() {
        System.out.println("Ouinc");
    }
}
```



Une oie ne cancaner
pas, elle cacarde.



MUSCLEZ VOS NEURONES

Supposez que nous voulions pouvoir utiliser une Oie partout où nous voulons utiliser un Canard. Après tout, les oies émettent des sons, les oies volent, les oies nagent. Pourquoi n'y aurait-il pas des oies dans le simulateur ?

Quel est le pattern qui nous permettrait de mélanger facilement des Oies avec des Canards ?

⑤ Il nous faut un adaptateur d'oies.

Notre simulateur s'attend à voir des interfaces `Cancaneur`. Comme les oies ne cancanent pas (elles cacardent), nous pouvons utiliser un adaptateur pour adapter une oie à un canard.

```
public class AdaptateurDOie implements Cancaneur {
    Oie oie;

    public AdaptateurDOie(Oie oie) {
        this.oie = oie;
    }

    public void cancaner() {
        oie.cacarder();
    }
}
```

Souvenez-vous : un `Adaptateur` implémente l'interface cible, qui est en l'occurrence `Cancaneur`.

Le constructeur accepte l'oie que nous allons adapter.

Quand `cancaner()` est appelée, l'appel est délégué à la méthode `cacarder()` de l'oie.

⑥ Maintenant, les oies devraient pouvoir jouer leur rôle dans le simulateur.

Il suffit de créer une `Oie`, de l'envelopper dans un adaptateur qui implémente `Cancaneur` et normalement le tour est joué.

```
public class SimulateurDeCanards {
    public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        simulateur.simuler();
    }

    void simuler() {
        Cancaneur colvert = new Colvert();
        Cancaneur mandarin = new Mandarin();
        Cancaneur appellant = new Appellant();
        Cancaneur canardEnPlastique = new CanardEnPlastique();
        Cancaneur canardOie = new AdaptateurDOie(new Oie());

        System.out.println("\nSimulateur de canards : avec adaptateur d'oies");

        simuler(colvert);
        simuler(mandarin);
        simuler(appellant);
        simuler(canardEnPlastique);
        simuler(canardOie);
    }

    void simuler(Cancaneur canard) {
        canard.cancaner();
    }
}
```

Nous créons une `Oie` qui se comporte comme un `Canard` en l'enveloppant dans l'`AdaptateurDOie`.

Une fois l'`Oie` enveloppée, nous pouvons la traiter comme n'importe quel autre `Cancaneur`.

7 Et maintenant, une petite exécution rapide....

Cette fois, quand nous exécutons le simulateur, la liste des objets transmis à la méthode `simuler()` comprend une Oie enveloppée dans un adaptateur. Le résultat ? Cela devrait cacarder !

Voici notre oie ! Maintenant, elle peut cancaner avec le reste des canards.

```
Fichier Édition Fenêtre Aide PouleAuxOeufsDOR
% java SimulateurDeCanards

Simulateur de canards : avec adaptateur d'oies
Coincoin
Coincoin
Couincouin
Couic
Ouinc
%
```



Cancanologie

Les cancanologues sont fascinés par tous les aspects du comportement des Cancaneurs. Et il y a une chose qu'ils rêvent d'étudier depuis toujours : le nombre de couacs que fait une troupe de canards.

Comment pouvons-nous ajouter la capacité de compter les couacs sans modifier les classes Canard ?

Voyez-vous un pattern qui pourrait nous aider ?

Jules Magret,
Garde forestier
et cancanologue



⑧ Nous allons faire le bonheur de ces cancanologues et leur permettre de compter des couacs.

Comment ? Attribuons aux canards un nouveau comportement (la capacité de compter) en les enveloppant dans un objet décorateur. Il n'y aura absolument rien à changer dans le code de Canard.

CompteurDeCouacs est un décorateur.

Comme pour Adaptateur, nous devons implémenter l'interface cible.

Nous avons une variable d'instance pour le cancanneur que nous décorons.

Et comme nous comptons **TOUS** les couacs, nous utilisons une variable statique pour les mémoriser.

Nous plaçons la référence au Cancaneur que nous décorons dans le constructeur.

Quand `cancaner()` est appelée, nous déléguons l'appel au Cancaneur que nous décorons...

... puis nous incrémentons le nombre de couacs.

Nous ajoutons une autre méthode au décorateur. Cette méthode statique se contente de retourner le nombre de couacs émis par tous les Cancaneurs.

```

public class CompteurDeCouacs implements Cancaneur {
    Cancaneur canard;
    static int nombreDeCouacs;

    public CompteurDeCouacs (Cancaneur canard) {
        this.canard = canard;
    }

    public void cancaner() {
        canard.cancaner();
        nombreDeCouacs++;
    }

    public static int getCouacs() {
        return nombreDeCouacs;
    }
}

```


⑨ Nous devons mettre à jour le simulateur pour créer des canards décorés.

Nous devons maintenant envelopper chaque objet Cancaneur que nousinstancions dans un décorateur CompteurDeCouacs. Sinon, nous ne parviendrons jamais à compter tous ces couacs.

```
public class SimulateurDeCanards {
    public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        simulateur.simuler();
    }
    void simuler() {
        Cancaneur colvert = new CompteurDeCouacs(new Colvert());
        Cancaneur mandarin = new CompteurDeCouacs(new Mandarin());
        Cancaneur appelant = new CompteurDeCouacs(new Appelant());
        Cancaneur canardEnPlastique = new CompteurDeCouacs(new CanardEnPlastique());
        Cancaneur canardOie = new AdaptateurDOie(new Oie());

        System.out.println("\nSimulateur de canards : avec Décorateur");

        simuler(colvert);
        simuler(mandarin);
        simuler(appelant);
        simuler(canardEnPlastique);
        simuler(canardOie);

        System.out.println("Nous avons compté " +
            CompteurDeCouacs.getCouacs() + " couacs");
    }

    void simuler(Cancaneur canard) {
        canard.cancaner();
    }
}
```

Chaque fois que nous créons un Cancaneur, nous l'enveloppons dans un nouveau décorateur.

Comme le garde nous a dit qu'il ne voulait pas compter les cris des oies, nous ne les décorons pas.

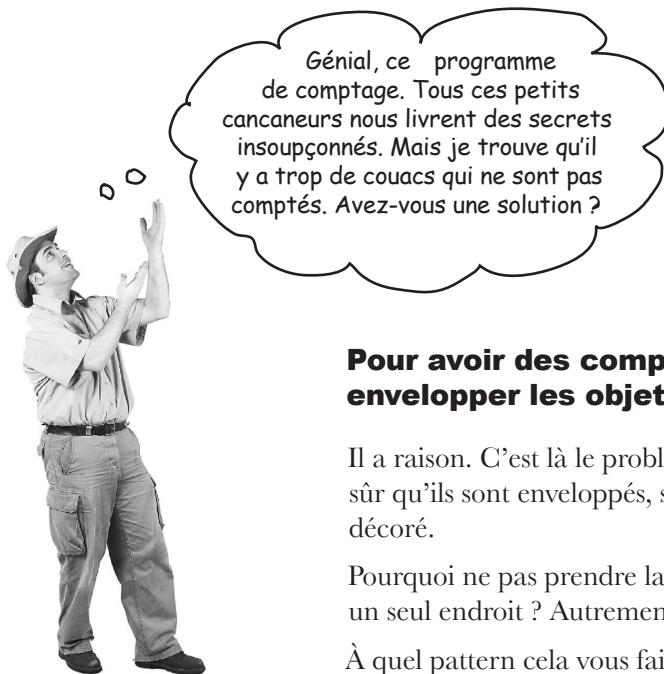
Voilà où nous recueillons les comportements de cancanement pour les Cancanologues.

Ici, rien ne change. Les objets décorés sont toujours des Cancaneurs.

Voilà le résultat!

Souvenez-vous : nous ne comptons pas les oies.

```
Fichier Édition Fenêtre Aide ŒufsDePâques
% java SimulateurDeCanards
Simulateur de canards : avec Décorateur
Coincoin
Coincoin
Couincouin
Couic
Ouinc
Nous avons compté 4 couacs
%
```

Pour avoir des comportements décorés, il faut envelopper les objets.

Il a raison. C'est là le problème des objets décorés : vous devez être sûr qu'ils sont enveloppés, sinon ils n'adoptent pas le comportement décoré.

Pourquoi ne pas prendre la création des canards et la concentrer dans un seul endroit ? Autrement dit, décorer cette création et l'encapsuler.

À quel pattern cela vous fait-il penser ?

⑩ Il nous faut une fabrique pour produire des canards !

Un peu de contrôle qualité nous permettrait d'être certains que nos canards sont bien enveloppés. Nous allons construire une fabrique uniquement destinée à les créer. Comme la fabrique devra produire une famille de produits composée de différents types de canards, nous allons employer le pattern Fabrique abstraite.

Commençons par la définition de la `FabriqueDeCanardsAbstraite` :

```
public abstract class FabriqueDeCanardsAbstraite {  
  
    public abstract Cancaneur creerColvert();  
    public abstract Cancaneur creerMandarin();  
    public abstract Cancaneur creerAppelant();  
    public abstract Cancaneur creerCanardEnPlastique();  
}
```

Nous définissons une fabrique abstraite que les sous-classes vont implémenter pour créer différentes familles.

Chaque méthode crée un type de canard.

Commençons par une fabrique qui crée des canards sans décorateurs, pour bien comprendre le principe :

```
public class FabriqueDeCanards extends FabriqueDeCanardsAbstraite {

    public Cancaneur creerColvert() {
        return new Colvert();
    }

    public Cancaneur creerMandarin() {
        return new Mandarin();
    }

    public Cancaneur creerAppelant() {
        return new Appelant();
    }

    public Cancaneur creerCanardEnPlastique() {
        return new CanardEnPlastique();
    }
}
```

FabriqueDeCanards étend la fabrique abstraite.

Chaque méthode crée un produit : un type particulier de Cancaneur. Le simulateur ne connaît pas le produit réel – il sait seulement qu'il reçoit un Cancaneur.

Maintenant, créons la fabrique dont nous avons vraiment besoin, la FabriqueDeComptage :

```
public class FabriqueDeComptage extends FabriqueDeCanardsAbstraite {

    public Cancaneur creerColvert() {
        return new CompteurDeCouacs(new Colvert());
    }

    public Cancaneur creerMandarin() {
        return new CompteurDeCouacs(new Mandarin());
    }

    public Cancaneur creerAppelant() {
        return new CompteurDeCouacs(new Appelant());
    }

    public Cancaneur creerCanardEnPlastique() {
        return new CompteurDeCouacs(new CanardEnPlastique());
    }
}
```

FabriqueDeComptage étend aussi la fabrique abstraite.

Chaque méthode enveloppe le Cancaneur dans le décorateur qui va permettre de compter les couacs. Le simulateur ne verra jamais la différence : il reçoit toujours un Cancaneur. Mais maintenant, nos gardes forestiers peuvent être sûrs que tous les couacs seront comptés.

11 Faisons en sorte que le simulateur utilise la fabrique.

Vous souvenez-vous du fonctionnement de Fabrique Abstraite ? On crée une méthode polymorphe qui accepte une fabrique et l'utilise pour créer des objets. En transmettant différentes fabriques, on parvient à utiliser différentes familles de produits dans la méthode.

Nous allons modifier la méthode `simuler()` pour qu'elle accepte une fabrique et l'utilise pour créer des canards.

```
public class SimulateurDeCanards {
    public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        FabriqueDeCanardsAbstraite fabriqueDeCanards = new FabriqueDeComptage();

        simulateur.simuler(fabriqueDeCanards);
    }

    void simuler(FabriqueDeCanardsAbstraite fabriqueDeCanards) {
        Cancaneur colvert = fabriqueDeCanards.creerColvert();
        Cancaneur mandarin = fabriqueDeCanards.creerMandarin();
        Cancaneur appellant = fabriqueDeCanards.creerAppelant();
        Cancaneur canardEnPlastique = fabriqueDeCanards.creerCanardEnPlastique();
        Cancaneur canardOie = new AdaptateurDOie(new Oie());

        System.out.println("\nSimulateur de canards : avec Fabrique abstraite");

        simuler(colvert);
        simuler(mandarin);
        simuler(appellant);
        simuler(canardEnPlastique);
        simuler(canardOie);
        System.out.println("Nous avons compté " +
            CompteurDeCouacs.getCouacs() + " couacs");
    }

    void simuler(Cancaneur canard) {
        canard.cancaner();
    }
}
```

Nous créons d'abord la fabrique que nous allons transmettre dans la méthode `simuler()`.

La méthode `simuler()` accepte `FabriqueDeCanardsAbstraite` et l'utilise pour créer des canards au lieu de les instancier directement.

Pas de changements ici ! Toujours le même bon vieux code.

Voilà le résultat obtenu en utilisant la fabrique...

Comme tout à l'heure,
mais, cette fois, nous
sommes sûrs que les
canards sont tous
décorés parce que
nous utilisons la
FabriqueDeComptage.

```
Fichier Edition Fenêtre Aide FabriqueDCEufs
% java SimulateurDeCanards
Simulateur de canards : avec Fabrique abstraite
Coincoin
Coincoin
Couincouin
Couic
Ouinc
Nous avons compté 4 couacs
%
```



À vos crayons

Nous continuons à instancier directement les Oies en nous appuyant sur des classes concrètes. Sauriez-vous écrire une Fabrique Abstraite pour les Oies ? Comment gèrerait-elle la création de canards qui seraient des oies ?

Nous commençons à avoir du mal à gérer tous ces types de canards séparément. Auriez-vous un moyen de nous aider à les traiter comme un tout, et peut-être même de nous permettre de gérer quelques familles de canards que nous voulons suivre plus particulièrement ?



Ah, Il veut gérer une troupe de canards.

Encore une bonne question de notre cancanologue : pourquoi gérons-nous les canards individuellement ?

Voilà qui n'est pas très facile à gérer !

```
Cancaneur colvert = fabriqueDeCanards.creerColvert();
Cancaneur mandarin = fabriqueDeCanards.creerMandarin();
Cancaneur appelant = fabriqueDeCanards.creerAppelant();
Cancaneur canardEnPlastique = fabriqueDeCanards.creerCanardEnPlastique();
Cancaneur canardOie = new AdaptateurDOie(new Oie());

simuler(colvert);
simuler(mandarin);
simuler(appelant);
simuler(canardEnPlastique);
simuler(canardOie);
```

Ce qu'il nous faut, c'est une façon de gérer des collections, et même des sous-collections de canards (pour satisfaire à la requête de notre garde forestier). Ce serait également agréable de pouvoir appliquer des opérations à tout l'ensemble de canards.

Quel est le pattern qui peut nous aider ?

⑫ Créons une troupe de canards (en réalité une troupe de Cancaneurs).

Vous souvenez-vous du pattern Composite qui nous permet de traiter une collection d'objets de la même façon que des objets individuels ? Quel meilleur composite y a-t-il qu'une troupe de Cancaneurs ?

Voyons un peu comment cela va fonctionner :

Souvenez-vous : le composite doit implémenter la même interface que les éléments feuilles. Nos éléments feuilles sont les Cancaneurs.

```
public class Troupe implements Cancaneur {
    ArrayList cancaneurs = new ArrayList();

    public void add(Cancaneur cancaneur) {
        cancaneurs.add(cancaneur);
    }

    public void cancaner() {
        Iterator itérateur = cancaneurs.iterator();
        while (itérateur.hasNext()) {
            Cancaneur cancaneur = (Cancaneur)itérateur.next();
            cancaneur.cancaner();
        }
    }
}
```

Dans chaque Troupe, nous utilisons une ArrayList pour contenir les Cancaneurs qui appartiennent à la Troupe.

La méthode add() ajoute un Cancaneur à la Troupe.

Maintenant, la méthode cancaner() – après tout, la Troupe est aussi un Cancaneur. La méthode cancaner() de Troupe doit travailler sur la Troupe entière. Ici, nous parcourons l'ArrayList et nous appelons cancaner() sur chaque élément.



Code à la loupe

Avez-vous remarqué que nous avons essayé d'introduire un design pattern en douce ?

```
public void cancaner() {
    Iterator itérateur = cancaneurs.itérateur();
    while (itérateur.hasNext()) {
        Cancaneur cancaneur = (Cancaneur)itérateur.next();
        cancaneur.cancaner();
    }
}
```

C'est cela ! Le pattern Itérateur en pleine action !

13 Maintenant, nous devons modifier le simulateur.

Notre composite est prêt ; il suffit d'un peu de codes pour intégrer les canards à la structure composite.

```
public class SimulateurDeCanards {
    // méthode main()

    void simuler(FabriqueDeCanardsAbstraite fabriqueDeCanards) {
        Cancaneur mandarin = fabriqueDeCanards.creerMandarin();
        Cancaneur appelant = fabriqueDeCanards.creerAppelant();
        Cancaneur canardEnPlastique = fabriqueDeCanards.creerCanardEnPlastique();
        Cancaneur canardOie = new AdaptateurDOie(new Oie());
        System.out.println("\nSimulateur de canards : avec Composite - Troupes");

        Troupe troupeDeCanards = new Troupe();

        troupeDeCanards.add(mandarin);
        troupeDeCanards.add(appelant);
        troupeDeCanards.add(canardEnPlastique);
        troupeDeCanards.add(canardOie);

        Troupe troupeDeColverts = new Troupe();

        Cancaneur colvertUn = fabriqueDeCanards.creerColvert();
        Cancaneur colvertDeux = fabriqueDeCanards.creerColvert();
        Cancaneur colvertTrois = fabriqueDeCanards.creerColvert();
        Cancaneur colvertQuatre = fabriqueDeCanards.creerColvert();

        troupeDeColverts.add(colvertUn);
        troupeDeColverts.add(colvertDeux);
        troupeDeColverts.add(colvertTrois);
        troupeDeColverts.add(colvertQuatre);

        troupeDeCanards.add(troupeDeColverts);

        System.out.println("\nSimulateur de canards : Toute la troupe");
        simuler(troupeDeCanards);

        System.out.println("\nSimulateur de canards : Troupe de colverts");
        simuler(troupeDeColverts);
        System.out.println("\nNous avons compté " +
            CompteurDeCouacs.getCouacs() + " couacs");
    }

    void simuler(Cancaneur canard) {
        canard.cancaner();
    }
}
```

Créons tous les Cancaneurs, exactement comme avant.

Nous créons d'abord une Troupe, puis nous la remplissons de Cancaneurs.

Puis nous créons une nouvelle troupe de colverts.

Ici, nous créons une petite famille de colverts...

...et nous l'ajoutons à la troupe de colverts.

Puis nous ajoutons la troupe de colverts à la troupe principale.

Testons sur toute la Troupe !

Puis testons uniquement sur la troupe de colverts.

Enfin, transmettons les données au Cancanologue.

Rien à changer ici : une Troupe est un Cancaneur !

Lançons l'exécution...

```
Fichier Édition Fenêtre Aide LaMeilleureFaçonDeMarcher
% java SimulateurDeCanards
Simulateur de canards : avec Composite - Troupes
Simulateur de canards : Toute la troupe
Coincoin
Couincouin
Couic
Ouinc
Coincoin
Coincoin
Coincoin
Coincoin

Simulateur de canards : Troupe de colverts
Coincoin
Coincoin
Coincoin
Coincoin
Nous avons compté 11 couacs
```

Voici la première troupe.

Puis les colverts.

Les données semblent correctes (n'oubliez pas que nous ne comptons pas les oies).



Sécurité ou transparence ?

Vous vous souvenez peut-être qu'au chapitre consacré au pattern Composite, les composites (les Menus) et les nœuds feuilles (les Plats) avaient exactement le *même* ensemble de méthodes, y compris la méthode `add()`. En conséquence, nous pouvions appeler sur les plats des méthodes qui n'avaient pas vraiment de sens (comme essayer d'ajouter quelque chose à un plat en appelant `add()`). Cette approche présentait un avantage : la distinction entre les feuilles et les composites était *transparente*. Le client n'avait pas besoin de savoir s'il avait à faire à une feuille ou à un composite et il appelait simplement les mêmes méthodes sur les deux.

Ici, nous avons décidé de séparer les méthodes de maintenance des enfants du composite des nœuds feuilles. Autrement dit, seule les Troupes ont la méthode `add()`. Nous savons que cela ne veut rien dire d'ajouter quelque chose à un Canard, et, dans cette implémentation, c'est impossible. Vous ne pouvez ajouter un élément qu'à une Troupe. En conséquence, cette conception est plus *sûre* – vous ne pouvez pas appeler de méthodes qui n'ont pas de sens sur les composants – mais elle est moins transparente. Maintenant, le client doit savoir qu'un Cancaneur est une Troupe pour pouvoir lui ajouter des cancanes.

Comme toujours, la conception OO oblige à faire des choix, et vous devrez considérer les avantages et les inconvénients quand vous créez vos propres composites.



Le Composite est parfait ! Merci ! Maintenant, nous avons la demande inverse : nous devons également étudier les canards individuellement. Pouvez-vous nous fournir un moyen de compter les couacs de chaque canard en temps réel ?

Avez-vous dit « observateur » ?

On dirait que le Cancanologue aimerait observer le comportement des canards individuels. Cela nous conduit tout droit à un pattern fait exprès pour observer le comportement des objets : le pattern Observateur.

14 Il nous faut d'abord une interface Observable.

Souvenez-vous : l'Observable est l'objet qu'on observe. Un Observable doit posséder des méthodes permettant d'enregistrer les observateurs et de leur notifier des changements. Nous pourrions également avoir une méthode pour supprimer des observateurs, mais nous allons l'omettre pour que notre implémentation reste simple.

```
public interface CouacObservable {
    public void enregistrerObservateur(Observateur observateur);
    public void notifierObservateurs();
}
```

CouacObservable est l'interface que les Cancaneurs doivent implémenter pour qu'on puisse les observer.

Elle dispose également d'une méthode pour tenir les observateurs au courant : `notifierObservateurs()`.

Elle possède une méthode permettant d'enregistrer des observateurs. Tout objet implémentant l'interface `Observateur` peut écouter les couacs. Nous définirons l'interface `Observateur` dans une seconde.

Maintenant, nous devons faire en sorte que tous les Cancaneurs implémentent cette interface...

```
public interface Cancaneur extends CouacObservable {
    public void cancaner();
}
```

Nous étendons donc l'interface `Cancaneur` avec `CouacObservable`.

15

Maintenant, nous devons faire en sorte que toutes les classes concrètes qui implémentent Cancaneur puissent être un CouacObservable.

Pour ce faire, nous pourrions implémenter l'enregistrement et la notification dans chaque classe (comme nous l'avons fait au chapitre 2). Mais nous préférons cette fois procéder un peu différemment : nous allons encapsuler l'enregistrement et la notification dans une autre classe, appelons-la Observable, et la composer avec un CouacObservable. Ainsi, nous n'écrirons le code réel qu'une seule fois et CouacObservable n'aura besoin que du code nécessaire pour déléguer à la classe auxiliaire, Observable.

Commençons par la classe auxiliaire Observable...



Observable implémente toutes les fonctionnalités dont un Cancaneur a besoin pour être observable. Il suffit de l'insérer dans une classe et de faire en sorte que cette classe délègue à l'Observable.

Observable doit implémenter CouacObservable parce que ce sont les mêmes appels de méthode qui vont lui être délégués.

```
public class Observable implements CouacObservable {
    ArrayList observateurs = new ArrayList();
    CouacObservable canard;
```

```
    public Observable(CouacObservable canard) {
        this.canard = canard;
    }
```

```
    public void enregistrerObservateur(Observateur observateur) {
        observateurs.add(observateur);
    }
```

```
    public void notifierObservateurs() {
        Iterator itérateur = observateurs.iterator();
        while (itérateur.hasNext()) {
            Observateur observateur = (Observateur) itérateur.next();
            observateur.actualiser(canard);
        }
    }
}
```

Dans le constructeur, nous transmettons le CouacObservable qui utilise cet objet pour gérer son comportement observable. Regardez la méthode `notifierObservateurs()` ci-dessous : vous verrez que quand une notification a lieu, l'Observable transmet cet objet pour que l'observateur sache quel est l'objet qui cancanne.

Voici le code qui permet d'enregistrer un observateur.

Et celui qui envoie les notifications.

Voyons maintenant comment une classe Cancaneur utilise cet auxiliaire...

⑩ Intégrer l'auxiliaire Observable avec les classes Cancaneur.

Cela semble prometteur. Il nous suffit de faire en sorte que les classes Cancaneur soient composées avec un Observable et qu'elles sachent comment lui déléguer. Après quoi, elles seront prêtes à être des observables. Voici l'implémentation de Colvert ; celle des autres canards est identique.

```
public class Colvert implements Cancaneur {  
    Observable observable;
```

Chaque Cancaneur a une variable d'instance de type Observable.

```
    public Colvert() {  
        observable = new Observable(this);  
    }
```

Dans le constructeur, nous créons un Observable et nous lui transmettons une référence à l'objet Colvert.

```
    public void cancaner() {  
        System.out.println("Coincoin");  
        notifierObservateurs();  
    }
```

Quand nous cancanons, nous devons en informer les observateurs.

```
    public void enregistrerObservateur(Observateur observateur) {  
        observable.enregistrerObservateur(observateur);  
    }
```

```
    public void notifierObservateurs() {  
        observable.notifierObservateurs();  
    }  
}
```

Voici nos deux méthodes de CouacObservable. Remarquez que nous déléguons simplement à l'auxiliaire.



À vos crayons

Il y a un Cancaneur dont nous n'avons pas modifié l'implémentation, le décorateur CompteurDeCouacs. Nous devons également en faire un Observable. Pourquoi ne l'écririez-vous pas ?

⑪ Nous y sommes presque ! Il n'y a plus qu'à s'occuper du côté Observateur du pattern.

Nous avons implémenté tout ce qu'il faut pour les observables. Maintenant, il nous faut quelques observateurs. Nous allons commencer par l'interface Observer :

L'interface Observateur n'a qu'une méthode, `actualiser()`, à laquelle on transmet le `CouacObservable` qui vient de cancaner.

```
public interface Observateur {
    public void actualiser(CouacObservable canard);
}
```

Maintenant, il nous faut un observateur. Mais où sont les cancanologues ?!

Nous devons implémenter l'interface Observateur, sinon nous ne pourrions pas nous enregistrer auprès d'un `CouacObservable`.

```
public class Cancanologue implements Observateur {

    public void actualiser(CouacObservable canard) {
        System.out.println("Cancanologue : " + canard + " vient de cancaner.");
    }
}
```

Le Cancanologue est simple : il n'a qu'une méthode, `actualiser()`, qui affiche le Cancaneur qui vient de cancaner

À vos crayons



Et si un Cancanologue veut observer une troupe entière ? Et d'abord, qu'est-ce que cela veut dire ? Vous pouvez voir les choses ainsi : si nous observons un composite, nous observons tout ce qu'il contient. Si vous vous enregistrez auprès d'une troupe, le composite (la troupe) s'assure que vous êtes enregistré auprès de tous ses enfants (pardon, tous ses petits cancaneurs), ce qui peut comprendre d'autres troupes.

Allez-y. Écrivez le code de l'observateur de troupe avant d'aller plus loin...

- 18 Nous sommes prêts à observer. Mettons le simulateur à jour et testons-le :

```
public class SimulateurDeCanards {
    public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        FabriqueDeCanardsAbstraite fabriqueDeCanards = new FabriqueDeComptage();

        simulateur.simuler(fabriqueDeCanards);
    }

    void simuler(FabriqueDeCanardsAbstraite fabriqueDeCanards) {

        // créer les fabriques de canards et les canards

        // créer les troupes
        System.out.println("\nSimulateur de canards : avec Observateur");
        Cancanologue cancanologue = new Cancanologue();
        troupeDeCanards.enregistrerObservateur(cancanologue);
        simuler(troupeDeCanards);
        System.out.println("\nNous avons compté " +
            CompteurDeCouacs.getCouacs() + " couacs");
    }

    void simuler(Cancaneur canard) {
        canard.cancaner();
    }
}
```

← Ici, nous créons simplement un Cancanologue et nous l'enregistrons pour qu'il observe la troupe.

← Cette fois, nous nous contentons de simuler la troupe.

Essayons et voyons si cela fonctionne !

Et voici l'apothéose. Cinq, non, six patterns se sont réunis pour créer cette prodigieuse simulation. Sans plus de cérémonie, nous avons le plaisir de vous présenter...
SimulateurDeCanards !

```
Fichier Édition Fenêtre Aide RienQueDesCanards

% java SimulateurDeCanards
Simulateur de canards : avec Observateur
Coincoin
Cancanologue : Mandarin vient de cancaner.
Couincouin
Cancanologue : Appelant vient de cancaner.
Couic
Cancanologue : Canard en plastique vient de cancaner.
Ouinc
Cancanologue : Oie déguisée en Canard vient de cancaner.
Coincoin
Cancanologue : Colvert vient de cancaner.
Coincoin
Cancanologue : Colvert vient de cancaner.
Coincoin
Cancanologue : Colvert vient de cancaner.
Coincoin
Cancanologue : Colvert vient de cancaner.
Nous avons compté 7 couacs.

%
```

Après chaque couac, quel que soit son type, l'observateur reçoit une notification.

Et le cancanologue a toujours son compte.

il n'y a pas de questions stupides

Q: Ainsi, c'était un pattern composé ?

R: Non, c'était juste un ensemble de patterns qui collaboraient. Un pattern composé est formé de quelques patterns que l'on combine pour résoudre un problème récurrent. Dans quelques instants, nous allons jeter un coup d'œil à un pattern composé, le pattern Modèle-Vue-Contrôleur. C'est une collection de patterns qui a été utilisée maintes et maintes fois dans de nombreuses solutions de conception.

Q: C'est donc cela la vraie beauté des Design Patterns : je peux prendre un problème et commencer à lui appliquer des patterns jusqu'à ce que j'aie une solution ?

R: Non. Nous avons mis en scène tous ces canards pour vous montrer comment des patterns peuvent collaborer, mais vous n'aurez jamais besoin d'approcher une conception de cette façon. En fait, l'application de certains patterns à certains aspects de la solution du problème des canards relève peut-être de l'artillerie lourde.

Il suffit parfois de se contenter d'appliquer de bons principes de conception OO pour résoudre un problème de façon satisfaisante.

Nous détaillerons ce point au chapitre suivant, mais sachez qu'il ne faut employer de patterns que où et quand ils ont un sens. On ne démarre jamais avec l'intention d'employer des patterns juste pour le plaisir. Vous pouvez considérer la conception du SimulateurDeCanards comme forcée et artificielle. Mais bon, c'était amusant et cela nous a donné une bonne idée de la façon dont plusieurs patterns peuvent s'insérer dans une solution.

Qu'avons-nous fait ?

Nous avons commencé par une poignée de Cancaneurs...

Puis une oie est arrivée qui voulait se comporter comme un Cancaneur.

Nous avons donc utilisé le *pattern Adaptateur* pour adapter l'oie à un Cancaneur. Maintenant, vous pouvez appeler `cancaner()` sur une oie enveloppée dans un adaptateur et elle cacardera !

Ensuite, les Cancanologues ont décidé qu'ils voulaient compter

les couacs. Nous avons donc employé le *pattern Décorateur* pour ajouter un décorateur, `CompteurDeCouacs`, qui mémorise le nombre de fois où `cancaner()` est appelée puis délègue le couac au Cancaneur qu'il enveloppe.

Mais les Cancanologues avaient peur d'avoir oublié d'ajouter le

décorateur `CompteurDeCouacs`. Nous avons donc appliqué le *pattern Fabrique abstraite* pour créer des canards. Maintenant, chaque fois qu'ils veulent un canard, ils en demandent un à la fabrique et elle leur retourne un canard décoré. (Et n'oubliez pas qu'ils peuvent aussi faire appel à une autre fabrique s'ils veulent un canard non décoré !)

Nous avons des problèmes de gestion avec toutes ces oies, tous ces

canards et tous ces cancaneurs. C'est pourquoi nous avons employé le *pattern Composite* pour les grouper en troupes. Ce pattern permet également au cancanologue de créer des sous-ensembles pour gérer des familles de canards. Nous avons également introduit le *pattern Itérateur* dans notre implémentation en utilisant `Iterator` de `java.util` dans une `ArrayList`.

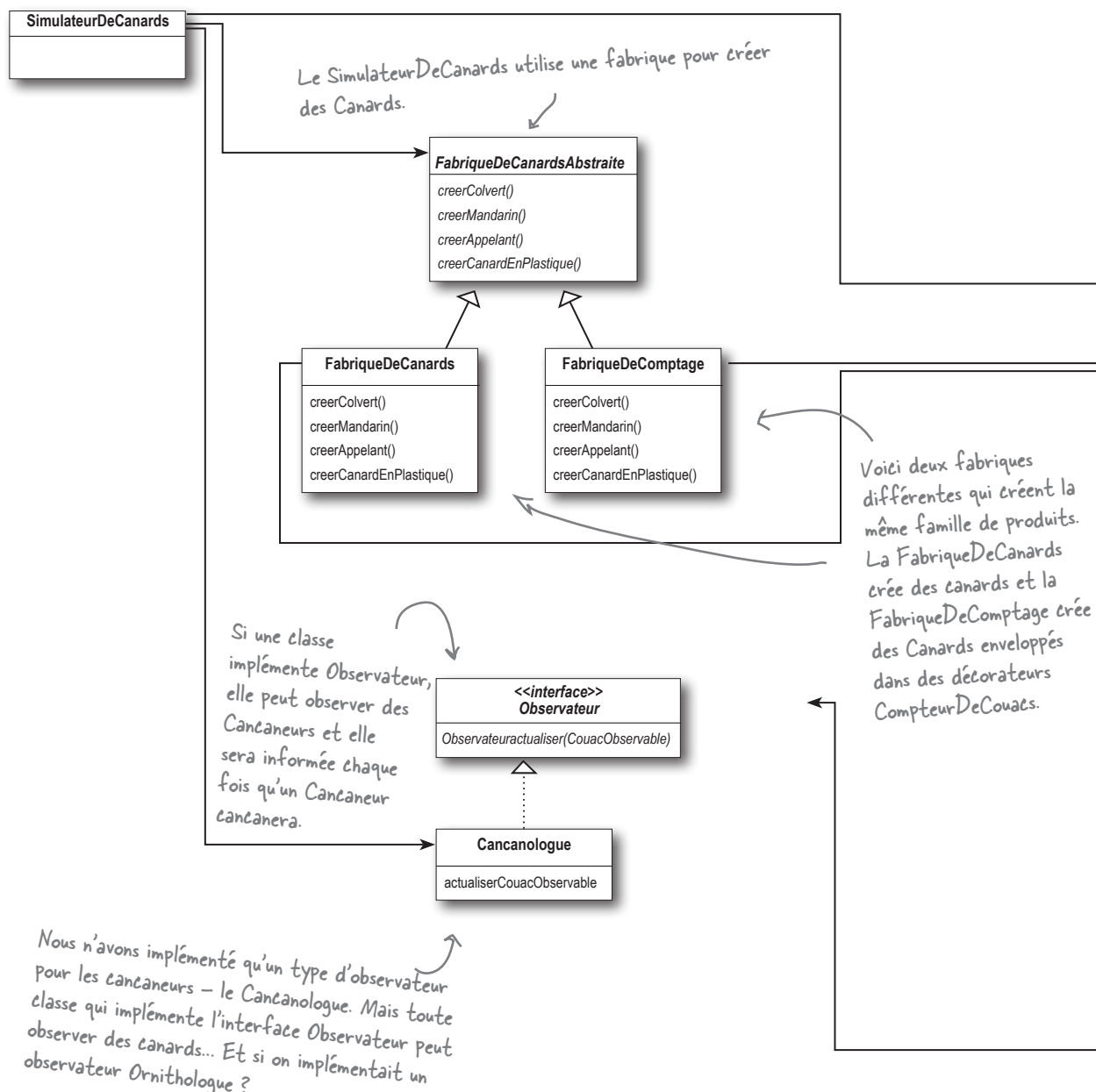
Les Cancanologues voulaient aussi être informés quand un cancaneur quelconque cancanait.

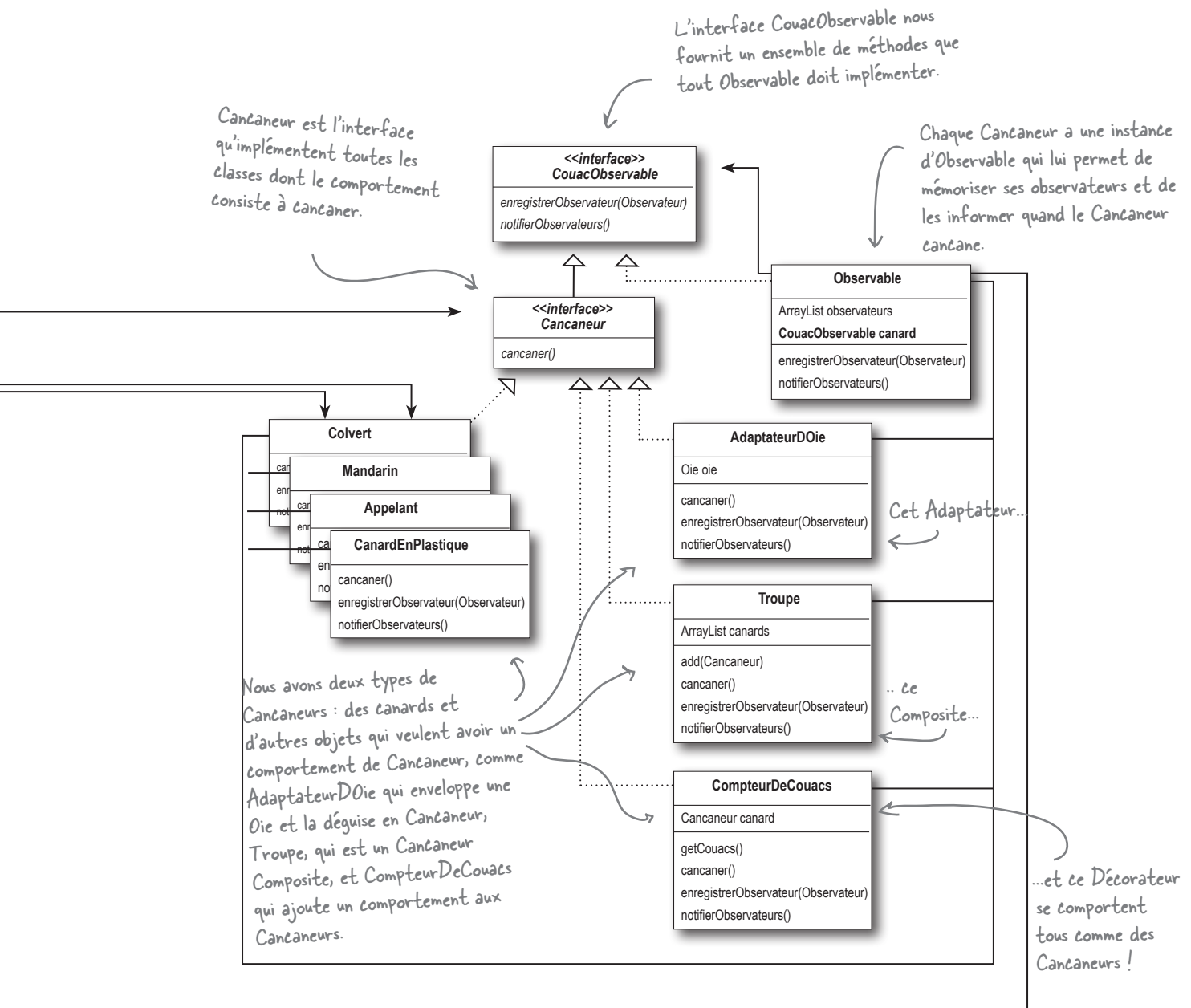
Nous avons alors utilisé le *pattern Observateur* pour que les Cancanologues puissent s'enregistrer comme observateurs de cancaneurs. Maintenant, ils reçoivent une notification chaque fois qu'un Cancaneur cancanne. Nous avons de nouveau utilisé *Iterator* dans cette implémentation. Les cancanologues peuvent même utiliser le *pattern Observateur* avec leurs composites.



Vue plongeante : le diagramme de classes

Nous avons incorporé de nombreux patterns dans un tout petit simulateur de canards ! Voici une vue d'ensemble de ce que nous avons fait :





Le roi des patterns composés

Si Elvis était un pattern composé, il s'appellerait **Modèle-Vue-Contrôleur** et il chanterait une petite chanson comme celle-ci...

Modèle, Vue, Contrôleur

Paroles et musique de James Dempsey.

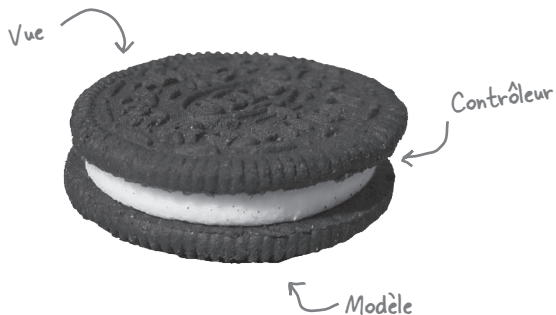
Modèle Vue, Modèle Vue, Modèle Vue Contrôleur

MVC est un paradigme qui factorise ton code

En plusieurs segments fonctionnels pour pas que ta tête explose.

Pour la réutilisabilité il faut des frontières claires

Le Modèle ici, la Vue là, et le Contrôleur au milieu.



Modèle Vue...

Il a trois couches comme les Choco BN

Modèle Vue Moelleux Contrôleur

Le Modèle est la raison d'être de ton application

Des objets qui contiennent données, logique et cetera

Tu crées des classes personnalisées dans le domaine de ton problème

Et tu peux choisir de permuter toutes les vues

Mais les objets du modèle restent les mêmes.

Tu peux modéliser tous les trucs que tu veux

La façon dont marche un bébé d'deux ans

Ou bien une bouteille de bon Chardonnay

Toutes les stupidités que les gens racontent

Ou le bruit que font les œufs en cuisant

La drôle de façon dont Hexley s'dandine

Un, deux, trois, quatre...

Modèle Vue... Tu peux modéliser tous les modèles qui posent pour GQ

Modèle Vue... Contrôleur

La vue a les contrôles qui affichent et éditent

Cocoa en a tout un tas, bien écrits il faut l'dire.

Prends une NSTextView, passe-lui une vieill' chaîne Unicode

L'utilisateur interagit avec, elle gère pratiquement tout

Mais la vue ne connaît pas le Modèle

Cette chaîne peut être un nombre ou bien les œuvres d'Aristote

Garde le couplage lâche

Et tu pourras réutiliser tout ce que tu voudras

Modèle Vue, avec un joli rendu en Aqua blue

Modèle Vue Contrôleur

Tu te demandes maintenant

Tu te demandes comment

Les données passent entre Modèle et Vue

Le Contrôleur est médiateur

Des états changeants entre couches

Pour synchroniser les données des deux
Il tire et pousse toutes les valeurs qui changent

Modèle Vue, grand bravo à l'équipe Smalltalk !
Modèle Vue... Contrôleur

Modèle Vue, ça se prononce Oh Oh pas Ouh Ouh
Modèle Vue... Contrôleur

Mais cette histoire n'est pas finie
Il reste de la route à faire
Personne ne semble tirer de gloire
De l'écriture du contrôleur
Car si l' Modèle est essentiel
Et si la vue est magnifique
J'suis p'têtre feignant, mais parfois c'est fou
Toutes ces lignes de code qui sont que de la colle
Et ce ne serait pas tellement tragique
Mais ce code ne fait vraiment rien de magique
Il ne fait que transmettre des valeurs
Et sans vouloir être méchant
Mais c'est vraiment répétitif
Tous ces trucs que les contrôleurs font
Et j'aimerais avoir dix centimes
Pour les centaines de fois

Que j'ai envoyé textField stringValue.

Modèle Vue

Mais comment se débarrasser de toute cette colle

Modèle Vue... Contrôleur

Les Contrôleurs connaissent... intimement le Modèle et la Vue

Il y a souvent du code en dur, ce qui menace la réutilisabilité

Mais maintenant vous pouvez connecter chaque valeur du modèle
à chaque propriété de la vue

Et une fois que vous commencerez à lier

Vous trouverez je crois moins de code dans votre source

Oui, je suis enthousiasmé par tout ce qu'ils ont automatisé

et tout ce que vous pouvez avoir pour rien

Et cela vaut la peine d'être répété

tout ce code dont tu n'auras pas besoin

quand tu travailleras en IB.

Modèle Vue, il gère même plusieurs sélections ^{avec Swing}

Modèle Vue Contrôleur

Modèle Vue, j'parie que je livre mon appli le premier

Modèle Vue Contrôleur

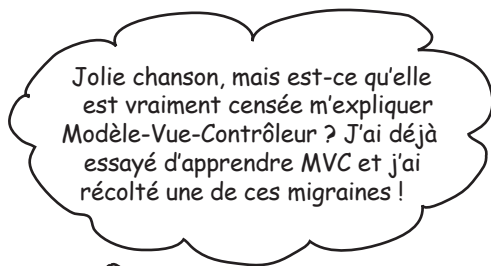


ÉCOUTEZ

Ne vous contentez pas de lire ! Après tout, nous sommes dans la collection Tête la première.
Attrapez votre iPod, tapez cette URL :

<http://www.wickedlysmart.com/headfirstdesignpatterns/media.html>

Faites une pause et écoutez la version originale.



Non. La clé de MVC, ce sont les design patterns.

Nous essayions seulement de vous aiguïser l'appétit. Savez-vous quoi ? Quand vous aurez terminé ce chapitre, écoutez de nouveau la chanson : vous l'apprécierez encore plus.

On dirait que MVC vous a déjà posé quelques problèmes ? C'est le cas de la plupart d'entre nous. Vous avez sans doute entendu d'autres développeurs dire qu'il leur a changé la vie et qu'il pourrait bien apporter la paix dans le monde. C'est assurément un pattern composé très puissant. Nous n'irons pas jusqu'à prétendre qu'il a des pouvoirs pacificateurs, mais il vous permettra d'économiser des heures de codage une fois que vous le connaîtrez bien.

Mais il faut d'abord l'apprendre, d'accord ? Or, il y a cette fois une différence de taille : maintenant vous connaissez les patterns !

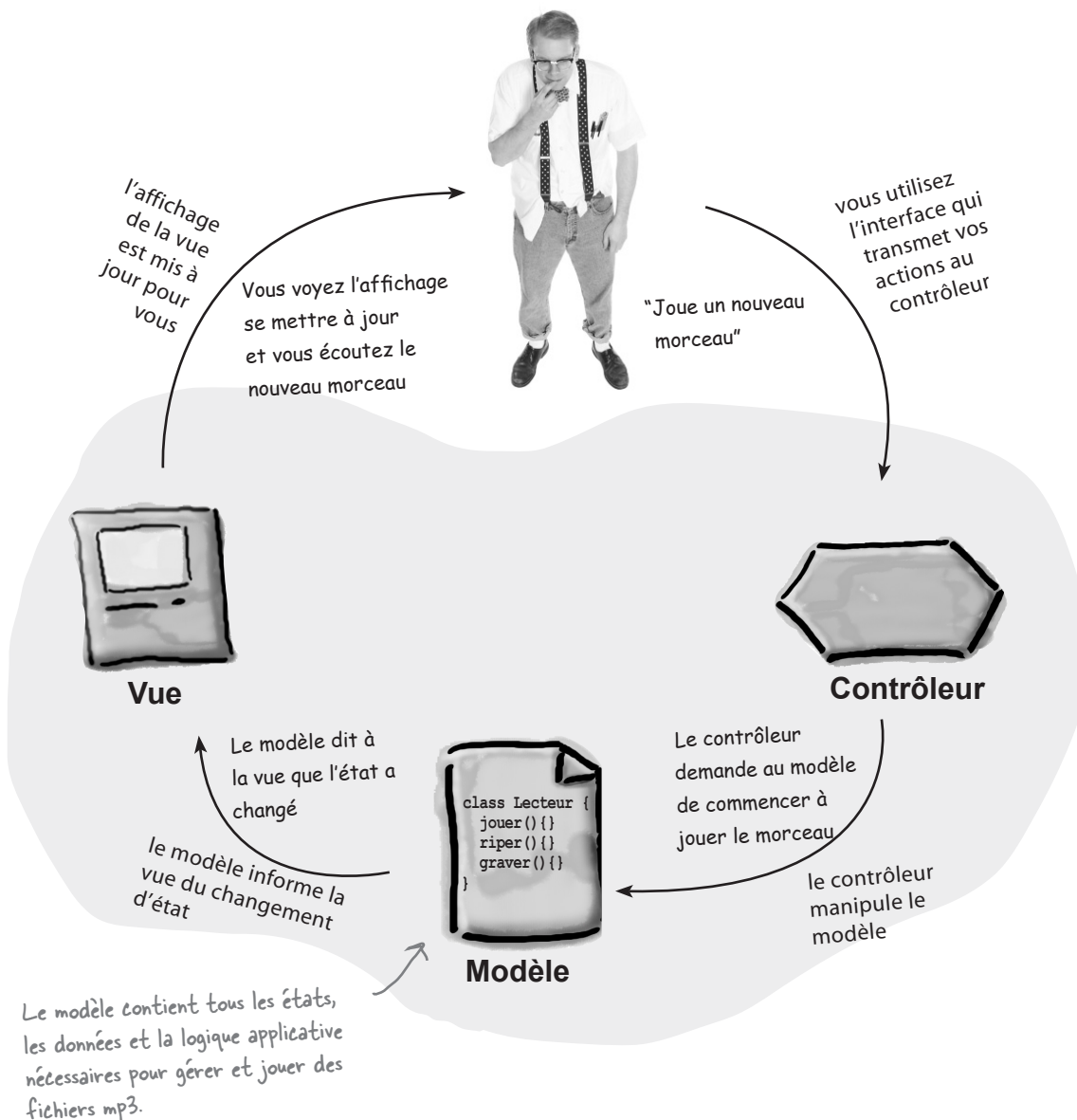
Oui, les patterns sont la clé de MVC. MVC est difficile à apprendre en partant de rien et rares sont les développeurs qui y parviennent. Voici le secret pour comprendre MVC : ce n'est rien d'autre qu'une collection de patterns. Si vous voyez MVC de cette manière, il devient tout à coup complètement logique.

Allons-y. Cette fois, vous n'allez faire qu'une bouchée de MVC !

Faites connaissance avec Modèle-Vue-Contrôleur

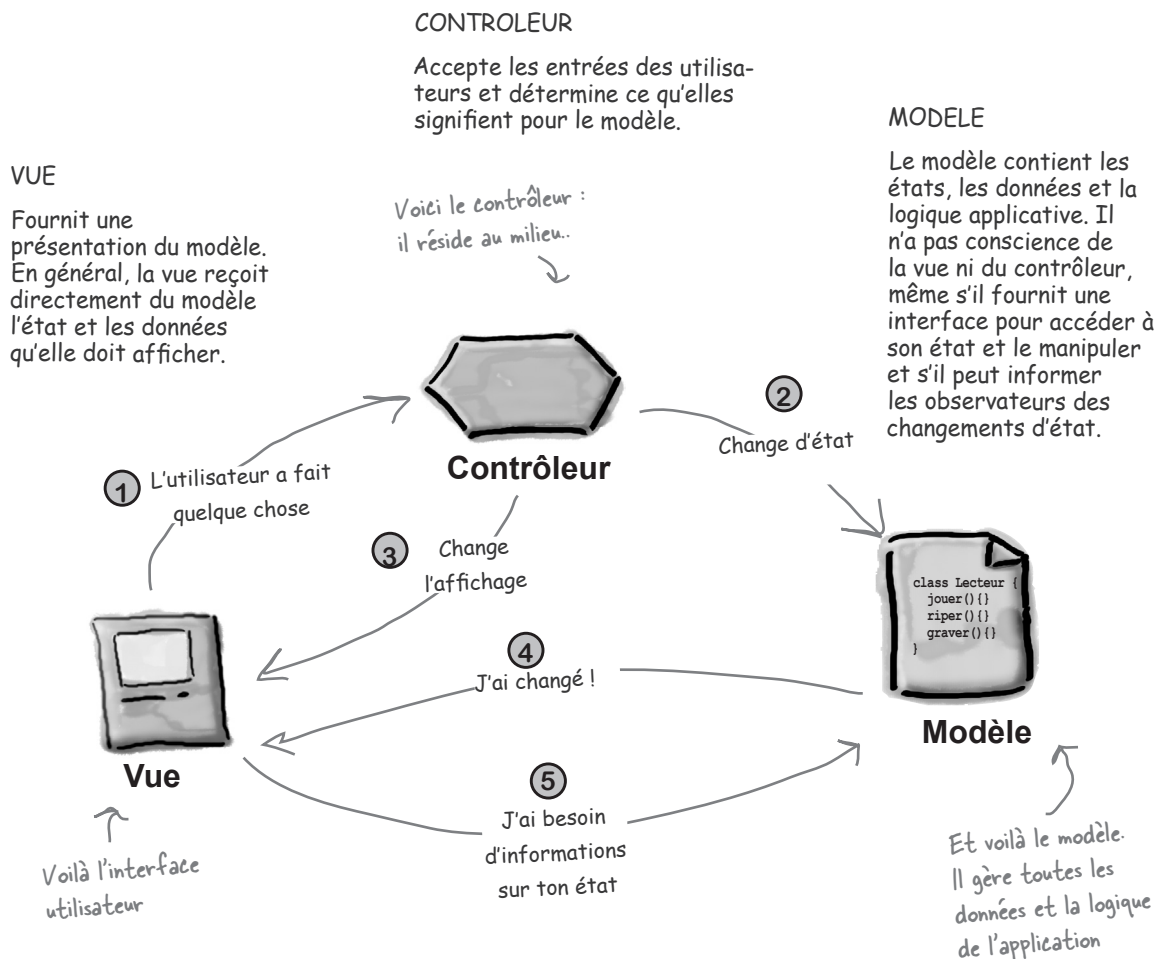
Imaginez votre lecteur MP3 favori, iTunes par exemple. Son interface vous permet d'ajouter de nouveaux morceaux, de gérer des listes et de renommer des pistes. Le lecteur se charge de maintenir une petite base de données qui contient tous vos morceaux, avec les noms et les données qui leur sont associés. Il s'occupe également de jouer les morceaux. Pendant ce temps, l'interface utilisateur est mise à jour en permanence et affiche notamment le titre du morceau courant et sa durée.

Eh bien ce qui gère tout cela, c'est Modèle-Vue-Contrôleur...



Zoom sur MVC...

La description du lecteur MP3 nous fournit une vue d'ensemble de MVC, mais elle ne vous aide pas réellement à comprendre les détails du fonctionnement du pattern, pas plus qu'elle ne vous explique comment en construire un vous-même ou pourquoi il est si intéressant. Commençons par examiner les relations entre le modèle, la vue et le contrôleur, puis nous l'envisagerons de nouveau en termes de design patterns.



- ① **Vous êtes l'utilisateur : vous interagissez avec la vue.**
Pour vous, la vue est une fenêtre ouverte sur le modèle. Lorsque vous faites quelque chose (par exemple cliquer sur le bouton Play), la vue dit au contrôleur ce que vous avez fait. Il appartient au contrôleur de le gérer.
- ② **Le contrôleur demande au modèle de modifier son état.**
Le contrôleur reçoit vos actions et les interprète. Si vous cliquez sur un bouton, c'est au contrôleur de déterminer ce que cela signifie et comment manipuler le modèle en fonction de cette action.
- ③ **Le contrôleur peut également demander à la vue de changer.**
Quand le contrôleur reçoit une action de la vue, il peut avoir besoin de dire à la vue de changer en conséquence. Par exemple, le contrôleur peut activer ou désactiver certains boutons ou certains éléments de menu dans l'interface.
- ④ **Le modèle informe la vue quand son état a changé.**
Lorsque quelque chose change dans le modèle, que ce soit une réponse à l'une de vos actions (comme cliquer sur un bouton) ou une autre modification interne (par exemple le morceau suivant dans la liste a commencé), le modèle notifie à la vue ce changement d'état.
- ⑤ **La vue demande l'état au modèle.**
La vue obtient l'état qu'elle affiche directement du modèle. Par exemple, lorsque le modèle informe la vue qu'un autre morceau a commencé, celle-ci demande le nom du morceau au modèle et l'affiche. Elle peut également demander un état au modèle si le contrôleur a requis un changement dans la vue.

Il n'y a pas de questions stupides

Q: Le contrôleur peut-il devenir un observateur du modèle ?

R: Bien sûr. Dans certaines conceptions, le contrôleur s'enregistre auprès du modèle et reçoit des notifications de changement. Ce peut être le cas quand un élément du modèle affecte directement les commandes de l'interface utilisateur. Par exemple, certains états du modèle peuvent dicter l'activation ou la désactivation de certains éléments de l'interface. À ce moment-là, c'est réellement au contrôleur qu'il appartient de demander à la vue de mettre à jour l'affichage en conséquence.

Q: Si je ne me trompe, le contrôleur se borne à accepter les entrées de l'utilisateur et à les transmettre au modèle. Alors pourquoi s'en encombrer s'il ne fait rien d'autre ? Pourquoi ne pas placer le code dans la vue elle-même ? Dans la plupart des cas, le contrôleur appelle juste une méthode sur le modèle, non ?

R: Le contrôleur ne se contente pas de transmettre les actions : il est chargé de les interpréter et de manipuler le modèle en conséquence. Mais votre vraie question est sans doute : pourquoi ne puis-je pas le faire dans le code de la vue ?

C'est possible, mais ce n'est pas souhaitable pour deux raisons. Premièrement, cela compliquerait le code de la vue parce qu'elle aurait alors deux responsabilités : gérer l'interface utilisateur et s'occuper de la logique de contrôle du modèle. Deuxièmement, la vue et le modèle seraient fortement couplés. Si vous voulez réutiliser la vue avec un autre modèle, c'est râpé. Le contrôleur sépare la logique de contrôle de la vue et découple la vue du modèle. En maintenant un couplage faible entre la vue et le contrôleur, vous construisez une conception plus souple et plus extensible, qui sera plus facile à modifier en aval.

Mettons nos lunettes « spéciales patterns »

Répetons-le : la meilleure façon d'apprendre MVC est de la considérer comme ce qu'il est, un ensemble de patterns qui collaborent dans une même conception.

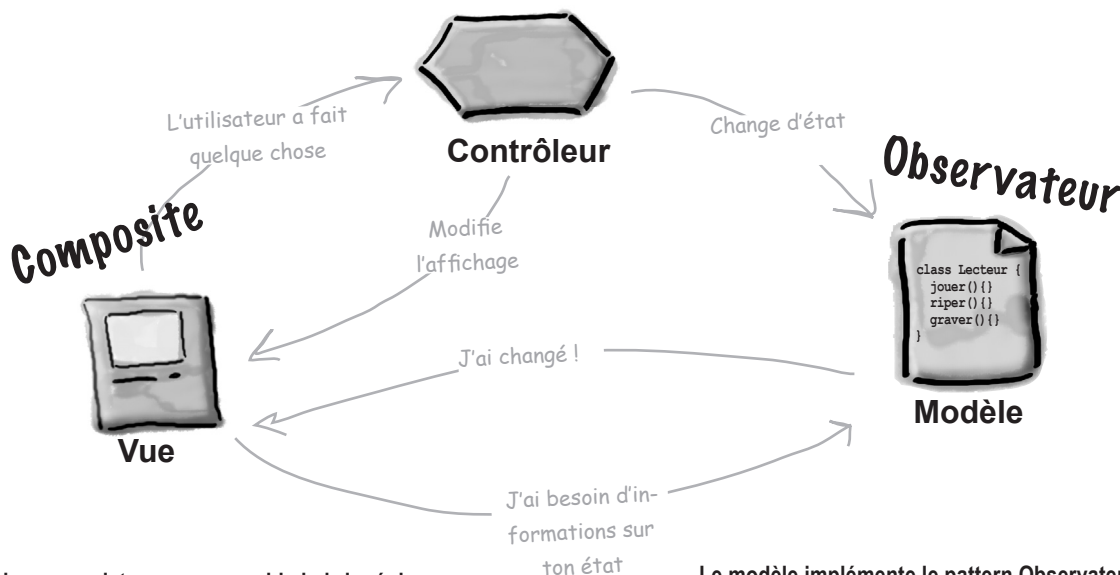
Commençons par le modèle. Comme vous l'avez peut-être deviné, le modèle utilise Observateur pour mettre à jour les vues et les contrôleurs en fonction des changements d'état les plus récents. De leur côté, la vue et le contrôleur mettent en œuvre le pattern Stratégie. Le contrôleur est le comportement de la vue, et il est facile de lui substituer un autre contrôleur si l'on désire un comportement différent. La vue elle-même emploie un pattern en interne pour gérer les fenêtres, les boutons et les autres composants de l'affichage : le pattern Composite.

Voyons cela de plus près :



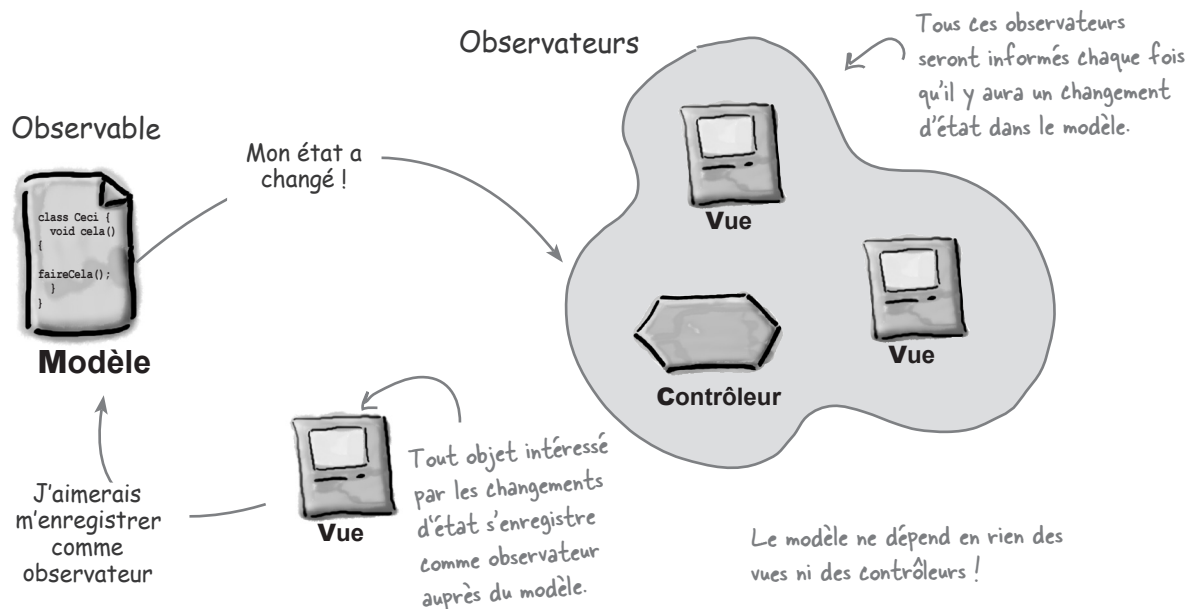
Stratégie

La vue et le contrôleur mettent en œuvre le pattern Stratégie classique : la vue est un objet qui est configuré avec une stratégie. Le contrôleur fournit la stratégie. La vue n'est concernée que par les aspects visuels de l'application et délègue au contrôleur toutes les décisions sur le comportement de l'interface. L'emploi du pattern Stratégie permet également de découpler la vue du modèle parce que c'est le contrôleur qui a la responsabilité d'interagir avec le modèle pour exécuter les requêtes des utilisateurs. La vue ne sait absolument pas comment il procède.

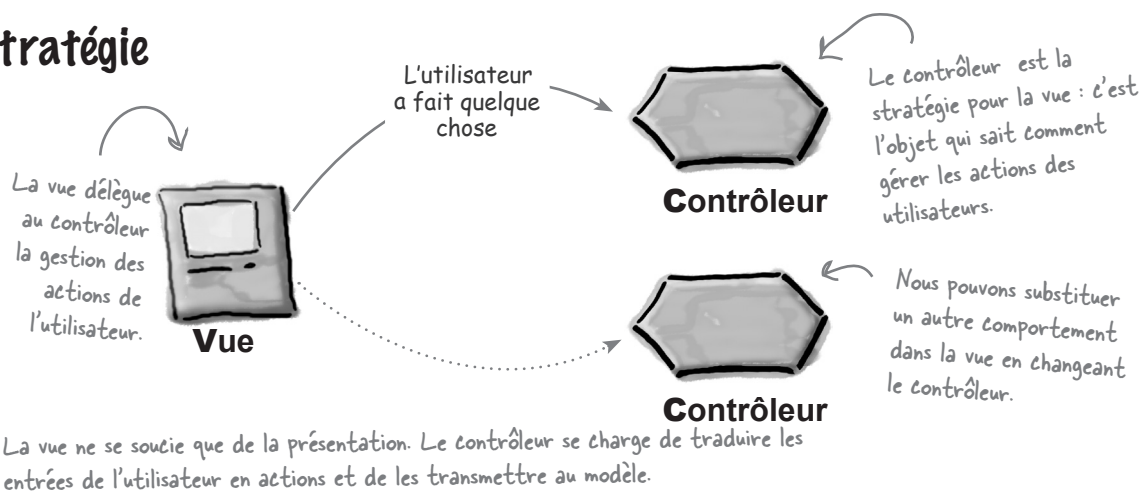


L'affichage consiste en un ensemble imbriqué de fenêtres, de panneaux, de boutons, de champs de texte, etc. Chaque composant de l'affichage est un composite (comme une fenêtre) ou une feuille (comme un bouton). Quand le contrôleur dit à la vue de s'actualiser, il lui suffit de s'adresser au composant principal et le Composite prend soin du reste.

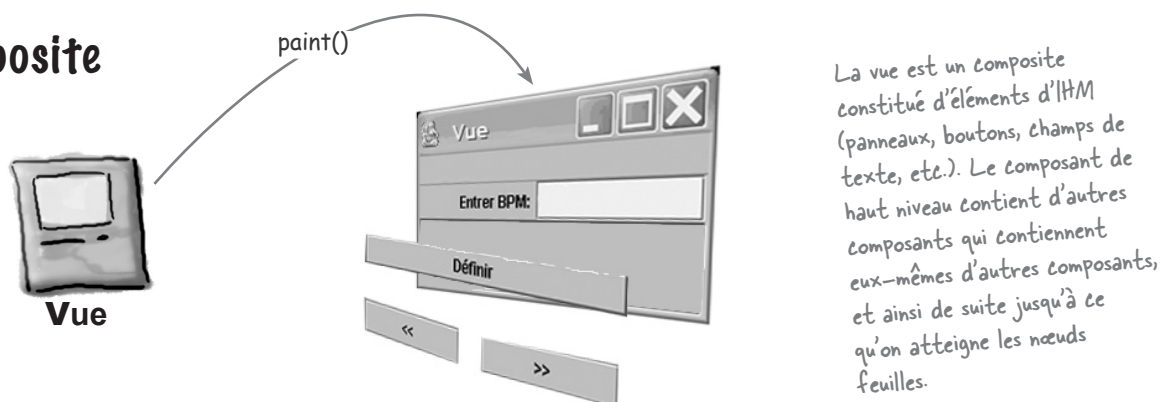
Le modèle implémente le pattern Observateur, afin que les objets intéressés soient mis à jour quand un changement d'état se produit. L'emploi du pattern Observateur garantit que le modèle demeure complètement indépendant des vues et des contrôleurs. Il permet également d'utiliser des vues différentes avec le même modèle, voire d'en utiliser plusieurs en même temps.



Stratégie



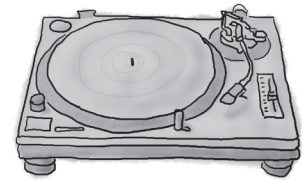
Composite



Contrôler le tempo avec MVC...

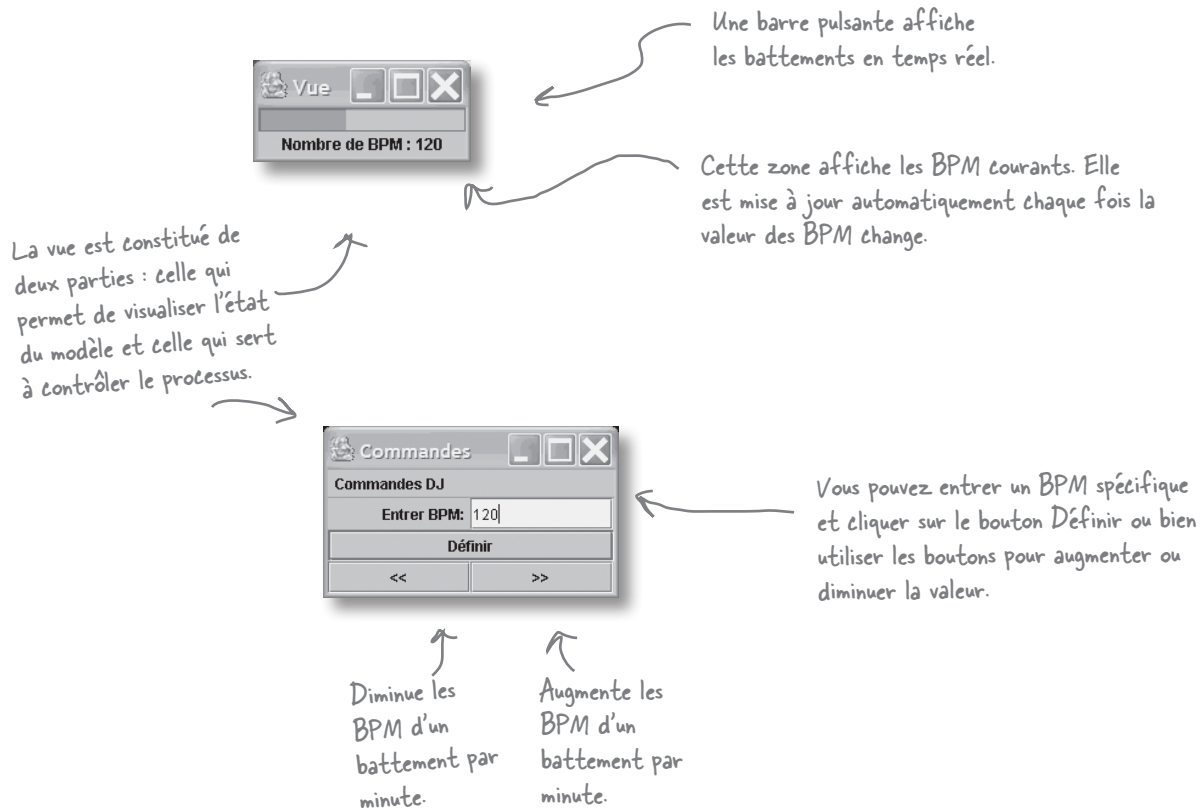
À votre tour d'être DJ. Quand vous êtes DJ, tout est affaire de tempo. Vous pouvez commencer par un *groove* à 95 battements par minute (BPM), puis déclencher la frénésie avec de la trance techno à 140 BPM et enfin terminer par un *ambient mix* très soft à 80 BPM.

Comment allez-vous procéder ? Vous devez contrôler le tempo et vous allez construire l'outil qui va vous permettre d'y parvenir.



Faites connaissance avec la Vue DJ

Commençons par la **vue** de l'outil. C'est la vue qui vous permet de créer et de régler le nombre de battements par minute...



Voici d'autres façons de contrôler la Vue DJ...



Vous générez le son en choisissant Démarrer dans le menu "Commandes DJ".

Utilisez le bouton Arrêter pour stopper la génération du son.



Remarquez que Arrêter est indisponible tant que vous n'avez pas démarré..

Notez que Démarrer est indisponible quand le son a commencé.

Toutes les actions de l'utilisateur sont transmises au contrôleur.

Le contrôleur est au milieu...

Le **contrôleur** réside entre la vue et le modèle. Il lit votre entrée, par exemple la sélection de Démarrer dans le menu Commandes DJ, et la transforme en action pour que le modèle commence la génération du son.

Le contrôleur accepte les entrées de l'utilisateur et détermine comment les traduire en requêtes au modèle.



Contrôleur

N'oublions pas le modèle sous-jacent...

Vous ne pouvez pas voir le **modèle**, mais vous pouvez l'entendre. Le modèle sous-tend tout le reste : il gère le tempo et pilote les enceintes avec du code MIDI.

Le *ModeleTempo* est le cœur de l'application. Il implémente la logique pour lancer et arrêter le son, fixer le nombre de battements par minute (BPM) et générer le son.

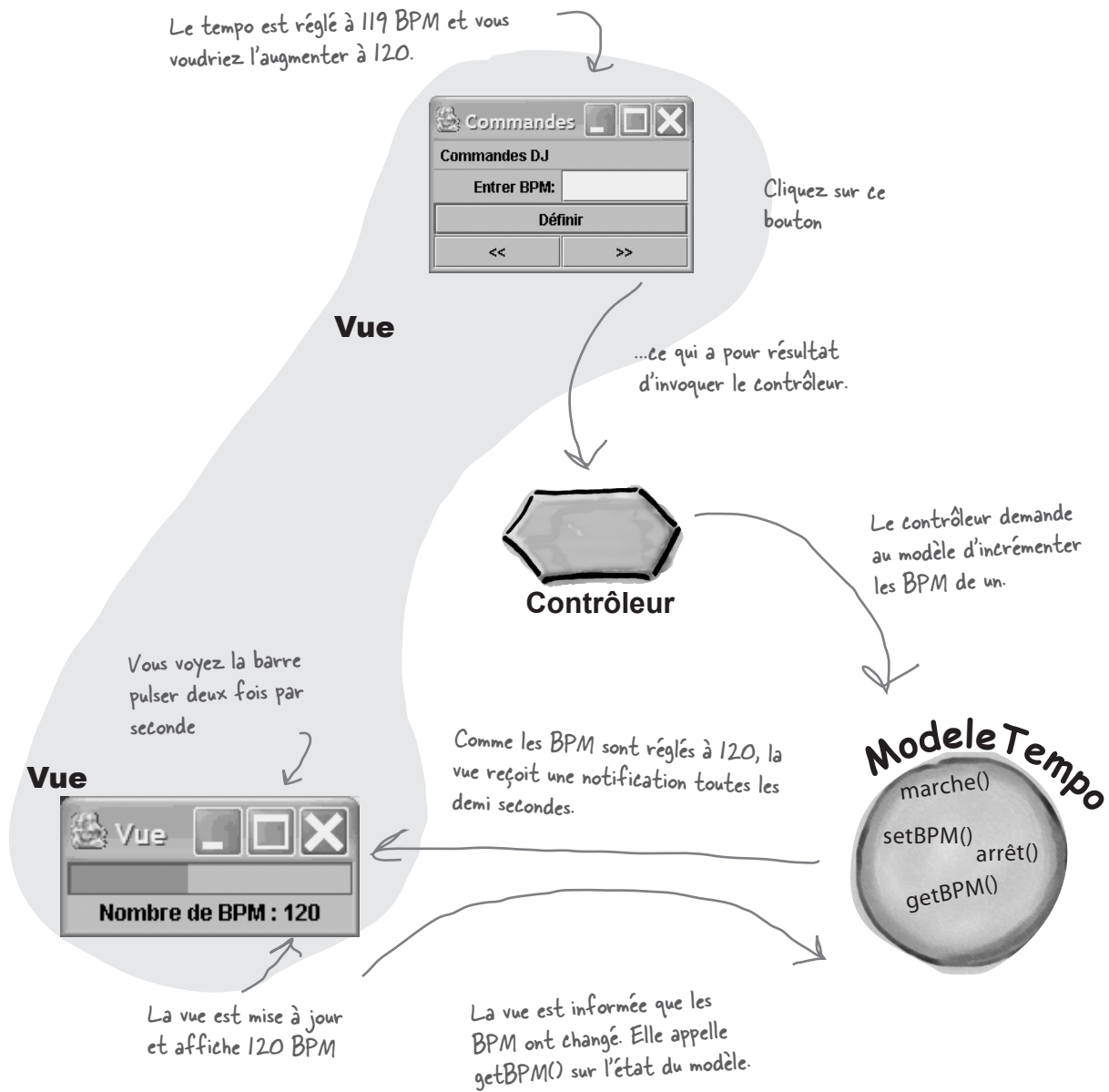
ModeleTempo

marche()
setBPM() arret()
getBPM()

Le modèle nous permet également de connaître son état courant via la méthode getBPM().



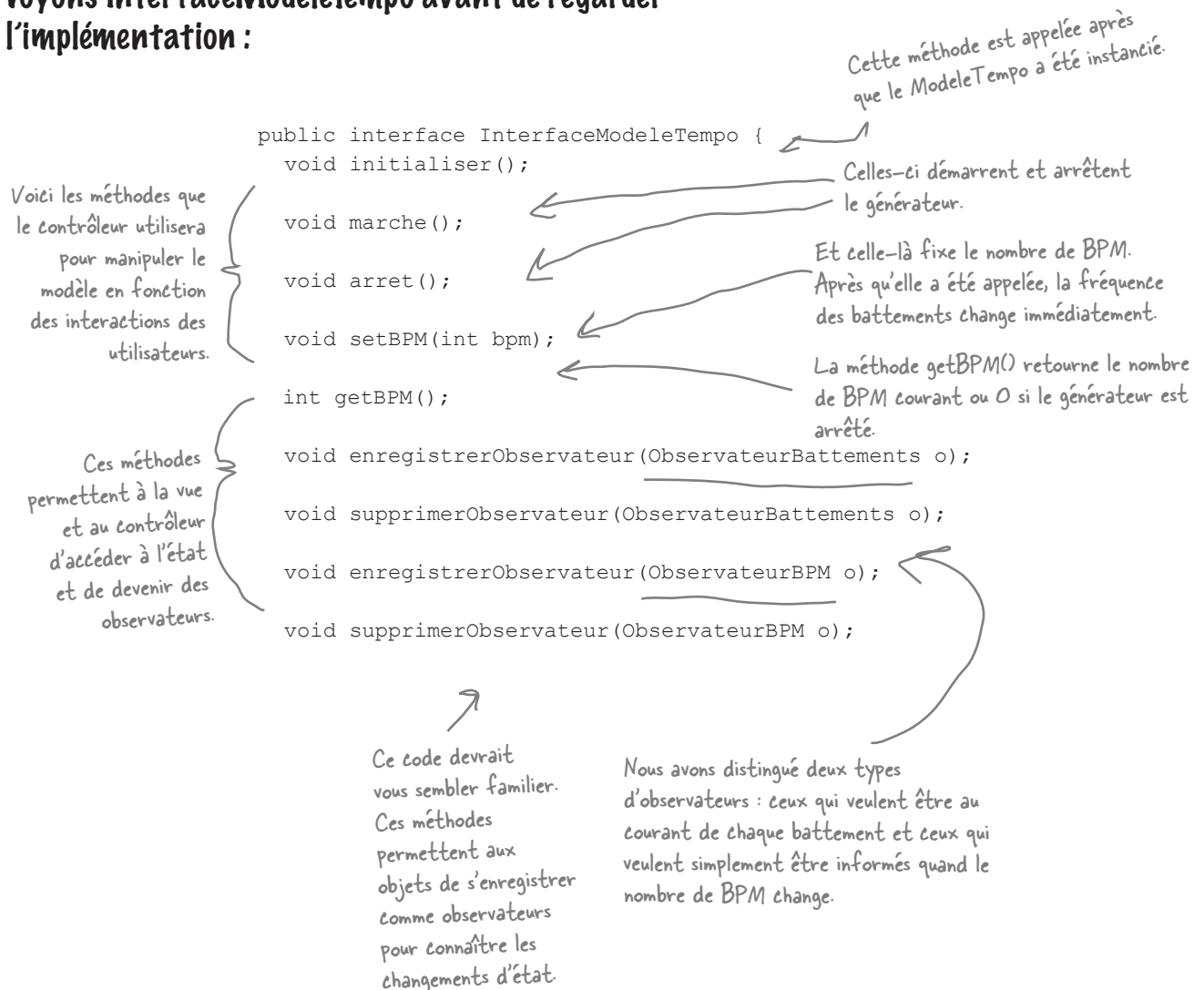
Assembler les morceaux



Construire les morceaux

Bien. Vous savez maintenant que le modèle est responsable de la maintenance de toutes les données, des états et de la logique applicative. Alors, que contient donc `ModeleTempo` ? Sa principale tâche consistant à gérer le tempo, il possède un état qui mémorise le nombre de BPM actuel et des quantités de code qui génère des événements MIDI pour créer le son que nous entendons. Il expose également une interface qui permet au contrôleur de manipuler le son et grâce à laquelle la vue et le contrôleur accèdent à l'état du modèle. De plus, n'oubliez pas que le modèle met en œuvre le pattern Observateur : nous devons donc insérer des méthodes pour que les objets s'enregistrent comme observateurs et pour envoyer des notifications.

Voyons `InterfaceModeleTempo` avant de regarder l'implémentation :



Voyons maintenant la classe concrète `ModeleTempo` :

Nous implémentons `InterfaceModeleTempo`.

Nécessaire pour le code MIDI.

```
public class ModeleTempo implements InterfaceModeleTempo, MetaEventListener {
    Sequencer sequenceur;
    ArrayList observateursBattements = new ArrayList();
    ArrayList observateursBPM = new ArrayList();
    int bpm = 90;
    // autres variables d'instance

    public void initialiser() {
        setUpMidi();
        construirePisteEtDemarrer();
    }

    public void marche() {
        sequenceur.start();
        setBPM(90);
    }

    public void arret() {
        setBPM(0);
        sequenceur.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequenceur.setTempoInBPM(getBPM());
        notifierObservateursBPM();
    }

    public int getBPM() {
        return bpm;
    }

    void evenementBattement() {
        notifierObservateursBattements();
    }

    // Code d'enregistrement et de notification des observateurs

    // Beaucoup de code MIDI pour gérer le son
}
```

Le séquenceur est l'objet qui sait comment générer de vrais sons (que vous pouvez entendre !).

Ces `ArrayList`s contiennent les deux types d'observateur (ceux qui observent les battements et ceux qui observent les BPM).

La variable d'instance `bpm` contient la fréquence des battements – par défaut, 90 BPM.

Cette méthode installe le séquenceur et définit le tempo.

La méthode `marche()` démarre le séquenceur et fixe le tempo par défaut : 90 BPM.

Et `arret()` l'arrête en fixant les BPM à 0 et en stoppant le séquenceur..

La méthode `setBPM()` est le moyen qui permet au contrôleur de manipuler le tempo. Elle fait trois choses :

- (1) Affecter la variable d'instance `bpm`
- (2) Demander au séquenceur de changer la fréquence des BPM.
- (3) Informer tous les ObservateurBPM que le nombre de BPM a changé.

La méthode `getBPM()` se contente de retourner le contenu de la variable d'instance `bpm`, qui indique le nombre de battements par minute actuel.

La méthode `evenementBattement()`, qui n'est pas dans `InterfaceModeleTempo`, est appelée par le code MIDI chaque fois qu'un nouveau battement commence. Cette méthode informe tous les ObservateurBattements qu'il y a eu un nouveau battement.

Beaucoup de code MIDI pour gérer le son



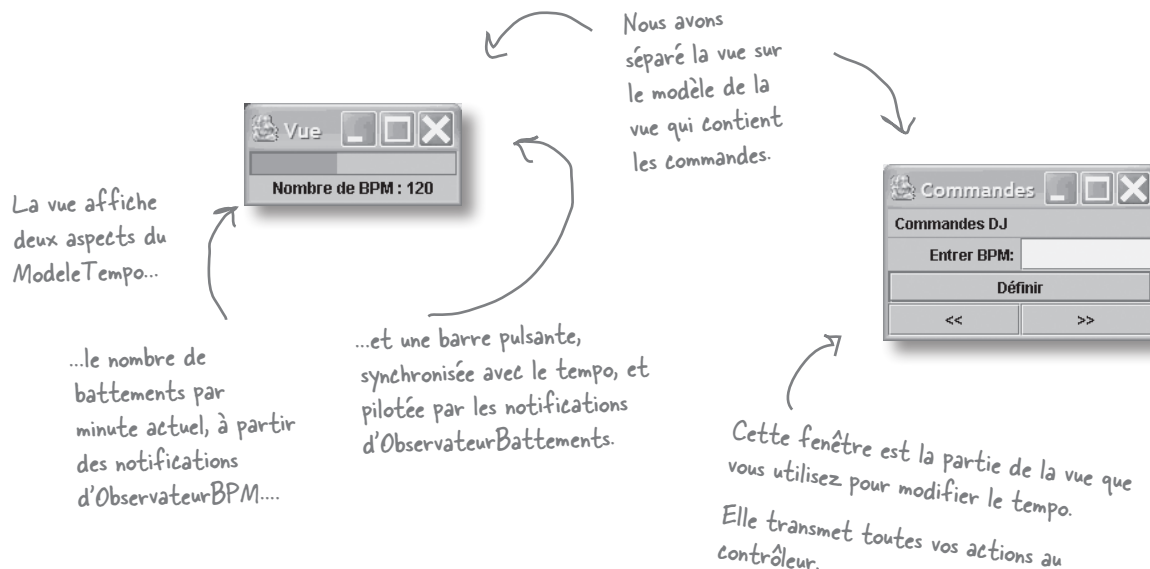
Code prêt à l'emploi

Ce modèle utilise le support MIDI de Java pour générer les sons. Vous trouverez l'implémentation complète de toutes les classes DJ dans les fichiers source téléchargeables sur le site de ce livre, ou dans le code figurant à la fin de ce chapitre.

La Vue

C'est là que cela commence à devenir amusant : nous allons installer une vue et visualiser le ModeleTempo !

Remarquez d'abord que nous avons implémenté la vue de manière à l'afficher dans deux fenêtres séparées. L'une des fenêtres contient la fréquence en BPM courante et la barre pulsante, tandis que l'autre contient les commandes de l'interface. Pourquoi ? Parce que nous voulions mettre l'accent sur la différence entre la partie de l'interface qui contient la vue du modèle et le reste de l'interface qui contient l'ensemble des commandes utilisateur. Regardons de plus près les deux parties de la vue :



MUSCLEZ VOS NEURONES

Notre ModeleTempo ne sait rien de la vue. Le modèle étant mis en œuvre selon le pattern Observateur, il se contente d'informer chaque vue enregistrée en tant qu'observateur quand son état change. La vue utilise l'API du modèle pour accéder à l'état. Nous n'avons implémenté qu'un type de vue. Voyez-vous d'autres vues qui pourraient utiliser les notifications et les états de ModeleTempo ?

Un lightshow basé sur le tempo en temps réel.

Une vue textuelle qui affiche le style de musique en fonction de la fréquence en BPM (ambient, downbeat, techno, etc.).

Implémenter la Vue

Les deux parties de la vue – la vue du modèle et la vue qui contient les commandes utilisateur – sont affichées dans deux fenêtres, mais elles résident toutes les deux dans la même classe Java. Nous allons commencer par le code qui crée la vue du modèle, celui qui affiche le nombre de BPM courant et la barre pulsante. À la page suivante, nous verrons le code qui crée les commandes de `ModeleTempo`, qui affiche les boutons et le champ de texte qui permet de définir le nombre de BPM.



Regardez ! Le code de ces deux pages n'est qu'un squelette !

Nous avons scindé UNE classe en DEUX, une partie de la vue apparaissant sur cette page et l'autre à la page suivante. En réalité, il ne s'agit que d'une seule classe : `VueDJ.java`. Vous trouverez le listing complet à la fin du chapitre.

`VueDJ` observe à la fois les BPM en temps réel ET les changements de BPM.

```
public class VueDJ implements ActionListener, ObservateurBattements, ObservateurBPM {
    InterfaceModeleTempo modele;
    InterfaceControleur controleur;
    JFrame cadreVue;
    JPanel panneauVue;
    BarrePulsante barre;
    JLabel affichageBPM;
```

La vue contient une référence au modèle ET au contrôleur.
Le contrôleur n'est utilisé que par l'interface de contrôle.
Nous allons le voir dans une seconde...

Ici, nous créons quelques composants destinés à l'affichage

```
    public VueDJ(InterfaceControleur controleur, InterfaceModeleTempo modele) {
        this.controleur = controleur;
        this.modele = modele;
        modele.enregistrerObservateur((ObservateurBattements) this);
        modele.enregistrerObservateur((ObservateurBPM) this);
    }

    public void creerVue() {
        // Création de tous les composants Swing
    }

    public void majBPM() {
        int bpm = modele.getBPM();
        if (bpm == 0) {
            affichageBPM.setText("hors ligne");
        } else {
            affichageBPM.setText("Nombre de BPM : " + modele.getBPM());
        }
    }

    public void majTempo() {
        barre.setValue(100);
    }
}
```

Le constructeur reçoit les références qu contrôleur et au modèle, et nous stockons ces références dans les variables d'instance.

Nous enregistrons également un `ObservateurBattements` et un `ObservateurBPM`.

La méthode `majBPM()` est appelée quand un changement d'état survient dans le modèle. À ce moment-là, nous mettons à jour l'affichage avec la valeur courante de BPM. Nous obtenons cette valeur en la demandant directement au modèle.

De même, la méthode `majTempo()` est appelée quand le modèle commence un nouveau battement. À ce moment-là, notre « barre pulsante » doit pulser. Pour ce faire, nous lui affectons sa valeur maximale (100) et nous la laissons gérer l'animation.

Implémenter la Vue, suite...

Voyons maintenant le code de la partie de la vue qui contient les commandes de l'interface utilisateur. Celle-ci vous permet de contrôler le modèle en disant quoi faire au contrôleur qui va à son tour dire quoi faire au modèle. N'oubliez pas : ce code est dans le même fichier .class que celui de l'autre partie de la vue.

```
public class VueDJ implements ActionListener, ObservateurBattements, ObservateurBPM{
    InterfaceModeleTempo modele;
    InterfaceControleur controleur;
    JLabel labelBPM;
    JTextField champTxtBPM;
    JButton definirBPM;
    JButton augmenterBPM;
    JButton diminuerBPM;
    JMenuBar barreMenus;
    JMenu menu;
    JMenuItem choixStart;
    JMenuItem choixStop;
}
```



```
public void creerCommandes() {
    // Création de tous les composants Swing
}
```

Cette méthode crée toutes les commandes et les place dans l'interface. Elle se charge également du menu. Quand l'utilisateur sélectionne Démarrer ou Arrêter, la méthode correspondante est appelée sur le contrôleur.

```
public void activerChoixStop() {
    choixStart.setEnabled(true);
}
```

```
public void desactiverChoixStop() {
    choixStop.setEnabled(false);
}
```

Toutes ces méthodes permettent d'activer et de désactiver Démarrer et Arrêter dans le menu. Nous verrons que le contrôleur les utilise pour modifier l'interface.

```
public void activerChoixStart() {
    choixStop.setEnabled(true);
}
```

Cette méthode est appelée lorsqu'on clique sur un bouton.

```
public void desactiverChoixStart() {
    choixStart.setEnabled(false);
}
```

Si l'on clique sur le bouton Définir, il est transmis au contrôleur avec la nouvelle valeur de bpm.

```
public void actionPerformed(ActionEvent evenement) {
    if (evenement.getSource() == definirBPM) {
        int bpm = Integer.parseInt(champTxtBPM.getText());
        controleur.setBPM(bpm);
    } else if (evenement.getSource() == augmenterBPM) {
        controleur.augmenterBPM();
    } else if (evenement.getSource() == diminuerBPM) {
        controleur.diminuerBPM();
    }
}
```

De même, si les boutons pour augmenter ou diminuer sont cliqués, cette information est transmise au contrôleur.

Et maintenant, le Contrôleur

Il est temps d'écrire le chaînon manquant : le contrôleur. Souvenez-vous : le contrôleur est la stratégie que nous insérons dans la vue pour lui donner des comportements.

Comme nous implémentons le pattern Stratégie, nous devons commencer par une interface qui puisse servir à toute Stratégie qui pourrait être insérée dans la VueDJ. Nous allons l'appeler InterfaceControleur.

```
public interface InterfaceControleur {  
    void start();  
    void stop();  
    void augmenterBPM();  
    void diminuerBPM();  
    void setBPM(int bpm);  
}
```

Voici toutes les méthodes
que la vue peut appeler sur le
contrôleur.

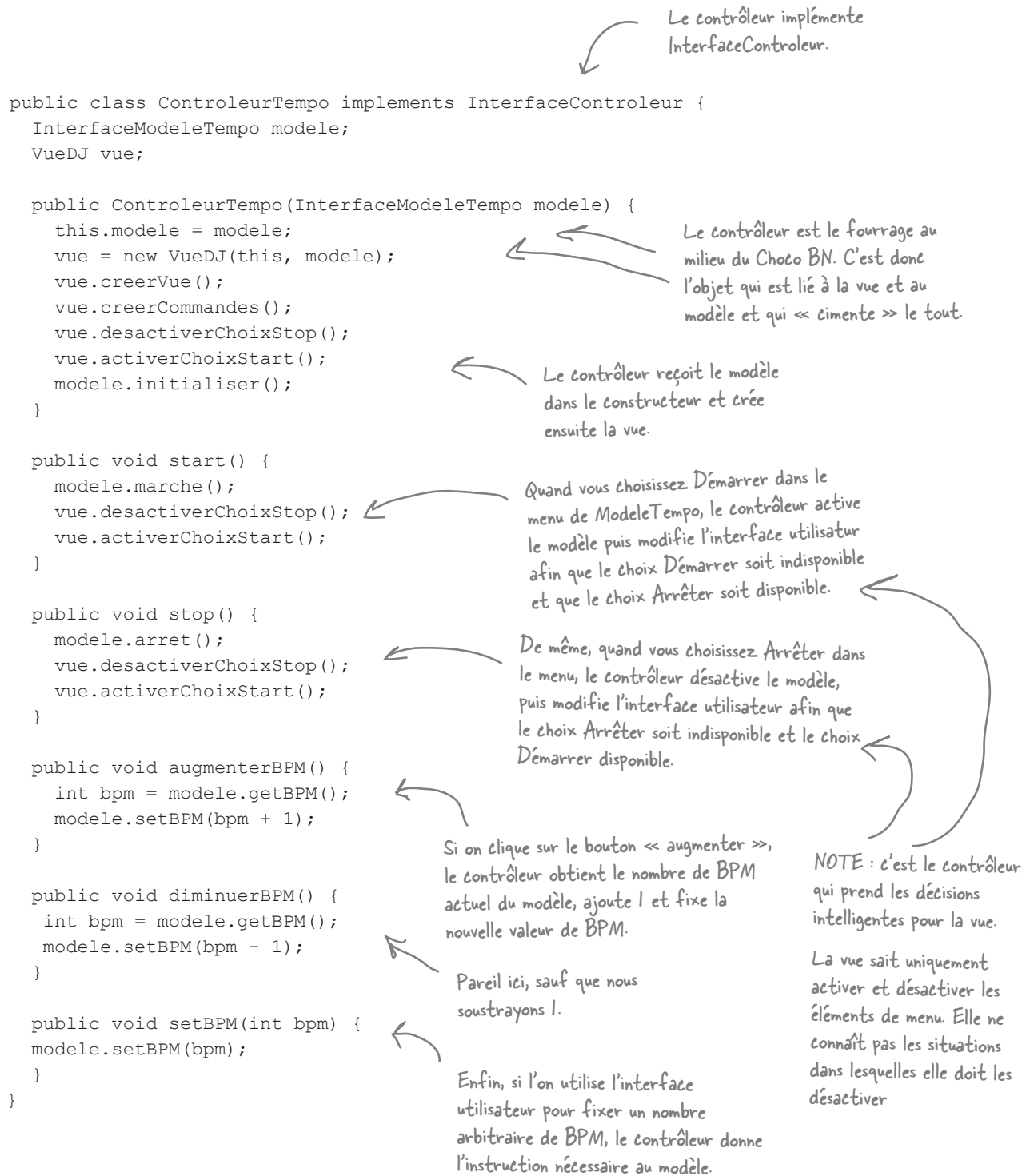
Ces méthodes devraient vous sembler familières après avoir vu l'interface du modèle. Vous pouvez lancer et arrêter la génération des sons et modifier le nombre de BPM. Cette interface est « plus riche » que l'interfaceModeleTempo parce que vous pouvez ajuster les BPM avec les boutons « augmenter » et « diminuer ».



Problème de conception

Vous venez de voir que la vue et le contrôleur emploient tous deux le pattern Stratégie. Pouvez-vous tracer un diagramme de classes qui représente ce pattern ?

Et voici l'implémentation du contrôleur :



Synthèse...

Nous avons tout ce qu'il nous faut : un modèle, une vue et un contrôleur. Il est temps de tout assembler dans un MVC ! Nous allons voir et entendre le superbe résultat de cette collaboration.

Tout ce qui manque, c'est un petit peu de code pour démarrer. Il n'en faut pas beaucoup :

```
public class TestDJ {
    public static void main (String[] args) {
        InterfaceModeleTempo modele = new ModeleTempo();
        InterfaceContrôleur controleur = new ControleurTempo(modele);
    }
}
```

D'abord, créer un modèle...

...puis créer un contrôleur et lui transmettre le modèle. Souvenez-vous : comme le contrôleur crée la vue, nous n'avons pas besoin de le faire.

Et maintenant, testons...

```
% java TestDJ
%
```

Exécutez ceci...

... et vous verrez cela.

Mode d'emploi

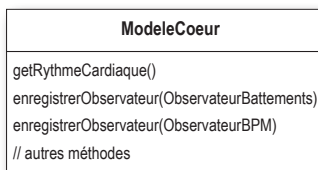
- 1 Lancez la génération du son en sélectionnant Démarrer dans le menu. Remarquez que le contrôleur désactive Démarrer ensuite.
- 2 Utilisez le champ de texte et les deux boutons pour modifier le nombre de BPM. Remarquez que l'affichage reflète les modifications en dépit du fait qu'elle n'a pas de lien logique avec les commandes.
- 3 Remarquez que la barre pulsante est toujours synchronisée avec le tempo, puisqu'elle est un observateur du modèle.
- 4 Mettez votre chanson favorite et voyez si vous pouvez vous synchroniser à son rythme avec les boutons pour augmenter et diminuer.
- 5 Arrêtez le générateur. Remarquez que le contrôleur désactive Arrêter et active Démarrer dans le menu.



Explorer le pattern Stratégie

Approfondissons un peu le pattern Stratégie pour mieux comprendre comment il est utilisé dans MVC. Nous allons aussi voir apparaître un autre pattern sympathique que l'on voit souvent tourner autour du trio MVC : le pattern Adaptateur.

Réfléchissez un instant à ce que fait la Vue DJ : elle affiche un nombre de battements et une pulsation. Es-ce que cela ne vous fait pas penser à autre chose ? Des battements de cœur par exemple ? Eh bien il se trouve que nous avons justement une classe pour un moniteur cardiaque. Voici le diagramme de classes :



← Nous avons une méthode qui lit le rythme cardiaque actuel.

← Et, heureusement, ses développeurs connaissaient les interfaces ObservateurBattements et ObservateurBPM !



MUSCLEZ VOS NEURONES

Ce serait génial de pouvoir réutiliser notre vue actuelle avec ModeleCoeur, mais il nous faut un contrôleur qui fonctionne avec ce modèle. De plus, l'interface de ModeleCoeur ne correspond pas à ce que la vue attend parce qu'elle a une méthode getRythmeCardiaque() au lieu d'une méthode getBPM(). Comment concevriez-vous un ensemble de classes pour permettre à la vue de réutiliser le nouveau modèle ?

Adapter le Modèle

Pour commencer, nous allons devoir adapter le `ModeleCoeur` à un `ModeleTempo`, faute de quoi la vue ne pourra pas fonctionner avec le modèle parce qu'elle a une méthode `getBPM()` et que son équivalent dans `ModeleCoeur` est `getRythmeCardiaque()`. Comment allons-nous procéder ? Nous allons utiliser le pattern Adaptateur, bien sûr ! Il s'avère que c'est une technique courante lorsqu'on travaille avec MVC : employer un adaptateur pour adapter un modèle de sorte qu'il fonctionne avec des vues et des contrôleurs existants.

Voici le code qui permet d'adapter un `ModeleCoeur` à un `ModeleTempo` :

```
public class AdaptateurCoeur implements InterfaceModeleTempo {
    InterfaceModeleCoeur coeur;

    public AdaptateurCoeur(InterfaceModeleCoeur coeur) {
        this.coeur = coeur;
    }

    public void initialiser() {}

    public void marche() {}

    public void arret() {}

    public int getBPM() {
        return coeur.getRythmeCardiaque();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(ObservateurBattements o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(ObservateurBattements o) {
        coeur.removeObserver(o);
    }

    public void registerObserver(ObservateurBPM o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(ObservateurBPM o) {
        coeur.removeObserver(o);
    }
}
```

Nous devons implémenter l'interface cible, en l'occurrence `InterfaceModeleTempo`.

Ici, nous mémorisons une référence à `ModeleCoeur`.

Nous ne savons pas exactement ce que ces méthodes feraient à un cœur, mais nous en frissonnons d'avance. Mieux vaut en faire des « opérations nulles ».

Lors de l'appel de `getBPM()`, nous traduisons simplement ce dernier en appel de `getRythmeCardiaque()` sur `ModeleCoeur`.

Ce ne sont pas des choses à faire à un cœur ! Encore une fois, mieux vaut une « opération nulle ».

Voici nos méthodes liées aux observateurs. Nous les déléguons simplement au `ModeleCoeur` enveloppé.

Maintenant, nous sommes prêts à écrire le `ControleurCoeur`

Avec notre `AdaptateurCoeur` en main, nous devrions être prêts à créer un contrôleur et à faire fonctionner la vue avec le `ModeleCoeur`. Ça, c'est de la réutilisation !

```
public class ControleurCoeur implements InterfaceControleur {
    InterfaceModeleCoeur modele;
    VueDJ vue;

    public ControleurCoeur(InterfaceModeleCoeur modele) {
        this.modele = modele;
        vue = new VueDJ(this, new AdaptateurCoeur(modele));
        vue.creerVue();
        vue.creerCommandes();
        vue.desactiverChoixStop();
        vue.desactiverChoixStart();
    }

    public void start() {}

    public void stop() {}

    public void augmenterBPM() {}

    public void diminuerBPM() {}

    public void setBPM(int bpm) {}
}
```

Le `ControleurCoeur` implémente `InterfaceControleur`, tout comme `ControleurTempo`.

Comme auparavant, le contrôleur crée la vue et cimente le tout.

Il n'y a qu'une modification : nous transmettons un `ModeleCoeur`, pas un `ModeleTempo`...

...et nous devons envelopper ce modèle dans un adaptateur avant de le transmettre à la vue.

Enfin, le `ControleurCoeur` désactive les éléments de menu quand ils ne sont pas nécessaires.

Pas grand chose à faire ici : après tout, nous ne pouvons pas contrôler des cœurs comme on contrôle des tempos.

Et voilà ! Il est temps de coder le test...

```
public class TestCoeur {
    public static void main (String[] args) {
        ModeleCoeur modeleCoeur = new ModeleCoeur();
        InterfaceControleur modele = new ControleurCoeur(modeleCoeur);
    }
}
```

Il suffit de créer le contrôleur et de lui transmettre le modèle.

Exécutons le test...

```
Fichier Édition Fenêtre Aide PrendreLePouls
% java TestCoeur
%
```

← Exécutez ceci...

...et vous verrez cela



Mode d'emploi

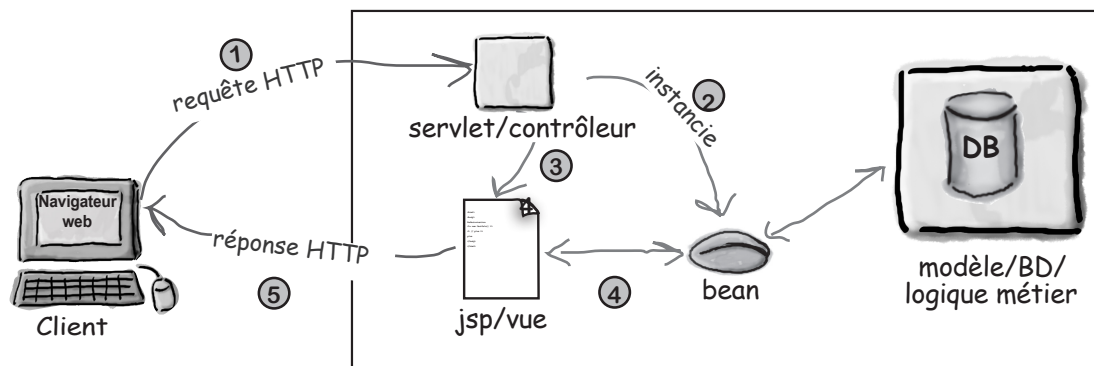
- 1 **Remarquez que l'affichage fonctionne parfaitement avec un cœur ! On dirait vraiment des pulsations. Puisque ModeleCoeur prend également en charge des ObservateurBattements et des ObservateurBPN, nous avons des mises à jour en direct comme dans l'application DJ.**
- 2 **Comme le rythme cardiaque varie naturellement, remarquez que l'affichage est actualisé avec le nouveau nombre de BPM.**
- 3 **Chaque fois que les BPM sont actualisés, l'adaptateur remplit son rôle : il traduit les appels de `getBPM()` en appels de `getRythmeCardiaque()`.**
- 4 **Les choix Démarrer et Arrêter ne sont pas disponibles dans le menu parce que le contrôleur les a désactivés.**
- 5 **Les autres boutons fonctionnent toujours, mais ils n'ont aucun effet parce que le contrôleur implémente des opérations nulles. On pourrait modifier la vue pour les rendre indisponibles.**

↑
Rythme cardiaque
parfaitement sain.

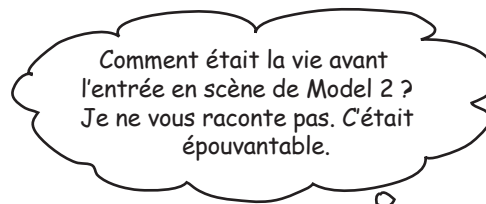
MVC et le Web

Le Web était à peine déployé que les développeurs commençaient à adapter MVC au modèle navigateur/serveur. L'adaptation dominante est simplement connue sous le nom de « Model 2 » et met en œuvre une combinaison de servlets et de JSP pour obtenir la même séparation du modèle, de la vue et du contrôleur que celle que nous voyons dans les IHM conventionnelles.

Voici comment fonctionne Model 2 :



- ① **Vous émettez une requête HTTP, qui est reçue par une servlet.**
Vous effectuez une requête HTTP via votre navigateur web. En général, elle consiste à envoyer les données d'un formulaire, par exemple votre nom d'utilisateur et votre mot de passe. Une servlet reçoit ces données et les analyse.
- ② **La servlet joue le rôle de contrôleur.**
La servlet joue le rôle du contrôleur et traite votre requête, très probablement en interrogeant le modèle (habituellement, une base de données). Le résultat du traitement de la requête est généralement « emballé » sous la forme d'un JavaBean.
- ③ **Le contrôleur transfère le contrôle à la vue.**
La Vue est représentée par une JSP. La seule tâche de celle-ci consiste à générer la page représentant la vue du modèle (④ qu'elle obtient par l'intermédiaire du JavaBean) ainsi que les contrôles nécessaires pour les futures actions.
- ⑤ **La vue retourne une page au navigateur via HTTP.**
Une page est retournée au navigateur dans lequel elle est affichée sous forme de vue. L'utilisateur soumet les requêtes suivantes qui sont traitées de la même façon.



Model 2 va plus loin que la conception.

Les avantages de la séparation de la vue, du modèle et du contrôleur n'ont plus de secrets pour vous. Mais vous ne connaissez pas encore la fin de l'histoire : Model 2 a empêché de nombreuses entreprises en ligne de sombrer dans le chaos.

Comment ? Eh bien, Model 2 ne se limite pas à la conception. Il ne s'agit pas seulement de séparer les composants, mais de séparer les *responsabilités de production*. Sachez-le : par le passé, toute personne capable d'accéder à vos JSP pouvait y écrire tout le code Java qu'elle voulait, même si elle ne savait pas faire la différence entre un fichier jar et une jarre d'huile d'olive. Regardons la réalité en face : la plupart des concepteurs web *connaissent les contenus et HTML, pas la programmation*.

Heureusement, Model 2 vint à la rescousse. Il nous a permis de laisser les tâches de développement aux gars et aux filles qui connaissent leurs servlets tandis que les concepteurs web peuvent exercer leurs talents sur des JSP simples de style Model 2, dans lesquelles ils n'ont accès qu'à du code HTML et à des JavaBeans sommaires.



Ancien fondateur
de startup

Model 2 : DJ sur mobile

Vous ne pensiez quand même pas que nous allions essayer de nous en tirer sans porter ce super ModelTempo sur le Web ? Imaginez seulement : vous pouvez contrôler toute votre session de DJ via une page web sur votre téléphone mobile. Vous pouvez maintenant sortir de votre cabine et vous mêler à la foule. Qu'attendez-vous ? Écrivons ce code !



Le plan

- ① **Adapter le modèle**
Eh bien, en fait, nous n'avons pas à adapter le modèle : il est très bien comme il est !
- ② **Créer une servlet contrôleur**
Il nous faut une servlet simple qui puisse recevoir nos requêtes HTTP et exécuter quelques opérations sur le modèle. Il suffit qu'elle sache démarrer, arrêter et modifier le nombre de battements par minute.
- ③ **Créer une vue HTML**
Nous allons créer une vue simple avec une JSP. Celle-ci va recevoir du contrôleur un ModelTempo qui lui indiquera tout ce qu'elle a besoin d'afficher. Ensuite, elle générera une interface HTML.



Trucs de geeks

Installer votre environnement de servlets

Vous montrer comment installer un environnement de servlets est un peu hors sujet pour un ouvrage sur les Design Patterns, à moins que vous ne vouliez que ce livre pèse trois tonnes !

Ouvrez votre navigateur web et allez directement à l'adresse <http://jakarta.apache.org/tomcat/> où vous pourrez consulter la page principale sur Tomcat, le conteneur de servlets du projet Apache Jakarta. Vous y trouverez tout ce qu'il vous faut pour être opérationnel.

Vous pouvez également lire *Head First Servlets & JSP* de Bryan Basham, Kathy Sierra et Bert Bates.



étape 1 : le modèle

Souvenez-vous : dans MVC, le modèle ne sait rien des vues ni des contrôleurs. Autrement dit, il est totalement découplé. Tout ce qu'il sait, c'est qu'il peut y avoir de nombreux observateurs auxquels il doit envoyer des notifications. C'est toute la beauté du pattern Observateur. Il fournit également une interface que les vues et les contrôleurs peuvent utiliser pour accéder à son état et le modifier.

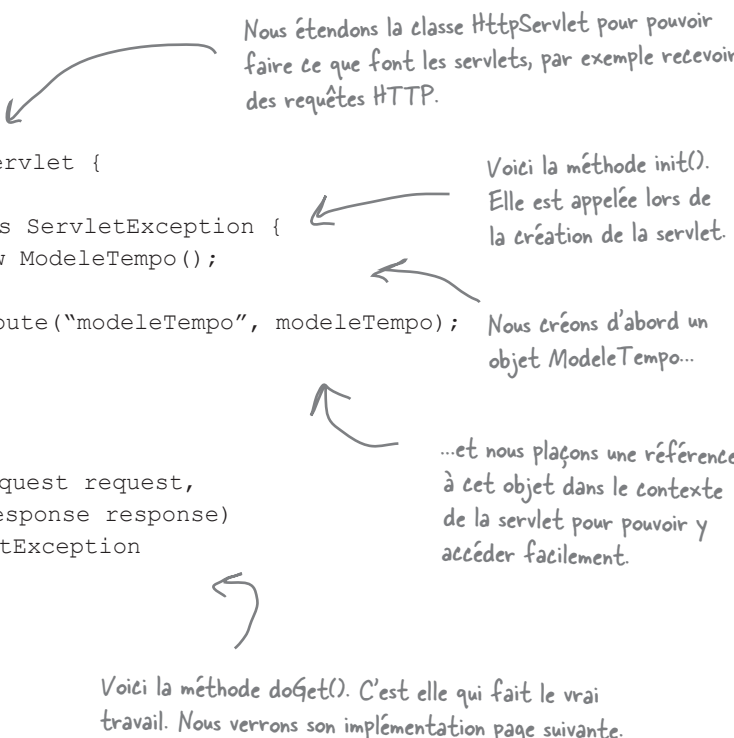
Il suffit donc d'adapter le modèle pour qu'il fonctionne dans un environnement web. Mais comme il ne dépend d'aucune classe extérieure, il n'y a strictement rien à faire. Nous pouvons utiliser notre `ModeleTempo` directement, sans aucune modification. Soyons donc productifs et passons à l'étape 2 !

étape 2 : la servlet contrôleur

Souvenez-vous : la servlet va jouer le rôle de notre contrôleur. Elle va recevoir les entrées du navigateur Web dans une requête HTTP et les transformer en actions pouvant être appliquées au modèle.

Puis, étant donnée la façon dont le Web fonctionne, nous devons retourner une vue au navigateur. Pour ce faire, nous transférons le contrôle à la vue, qui prend la forme d'une JSP. C'est ce que nous verrons à l'étape 3.

Voici le squelette de la servlet ; vous trouverez son implémentation complète page suivante.



```

public class VueDJ extends HttpServlet {

    public void initialiser() throws ServletException {
        ModeleTempo modeleTempo = new ModeleTempo();
        modeleTempo.initialiser();
        getServletContext().setAttribute("modeleTempo", modeleTempo);
    }

    // ici, la méthode doPost

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        // ici l'implémentation
    }
}

```

Nous étendons la classe `HttpServlet` pour pouvoir faire ce que font les servlets, par exemple recevoir des requêtes HTTP.

Voici la méthode `init()`. Elle est appelée lors de la création de la servlet.

Nous créons d'abord un objet `ModeleTempo`...

...et nous plaçons une référence à cet objet dans le contexte de la servlet pour pouvoir y accéder facilement.

Voici la méthode `doGet()`. C'est elle qui fait le vrai travail. Nous verrons son implémentation page suivante.

Voici l'implémentation de la méthode `doGet()` de la page précédente :

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException
{
    ModeleTempo modeleTempo =
        (ModeleTempo) getServletContext().getAttribute("modeleTempo");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = modeleTempo.getBPM() + "";
    }

    String definir = request.getParameter("definir");
    if (definir != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        modeleTempo.setBPM(bpmNumber);
    }

    String diminuer = request.getParameter("diminuer");
    if (diminuer != null) {
        modeleTempo.setBPM(modeleTempo.getBPM() - 1);
    }

    String augmenter = request.getParameter("augmenter");
    if (augmenter != null) {
        modeleTempo.setBPM(modeleTempo.getBPM() + 1);
    }

    String on = request.getParameter("on");
    if (on != null) {
        modeleTempo.start();
    }

    String off = request.getParameter("off");
    if (off != null) {
        modeleTempo.stop();
    }

    request.setAttribute("modeleTempo", modeleTempo);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/VueDJ.jsp");
    dispatcher.forward(request, response);
}
```

Premièrement, nous prenons le modèle dans le contexte de la servlet. En l'absence de référence, nous ne pourrions pas le manipuler.

Puis nous lisons les commandes et paramètres HTTP...

Si nous avons une commande «<definir>», nous prenons sa valeur et nous l'indiquons au modèle.

Pour augmenter ou diminuer, nous lisons le nombre de BPM actuel et nous l'incrémentons ou le décrémente de 1.

Si nous avons une commande on ou off, nous disons au modèle de commencer ou d'arrêter.

Enfin, notre tâche de contrôleur est terminée. Il ne reste plus qu'à demander à la vue de prendre le relais et de créer une vue HTML.

Selon la définition de Model 2, nous transmettons à la JSP un bean qui contient l'état du modèle. En l'occurrence, nous lui transmettons le modèle lui-même, puisqu'il se trouve que c'est un bean.

Maintenant, il nous faut une vue...

Il ne nous manque plus qu'une vue pour que notre générateur de tempo soit opérationnel ! Dans Model 2, la vue est une simple JSP. Tout ce que la JSP sait du bean, elle le reçoit du contrôleur. Dans notre cas, ce bean est le modèle lui-même et la JSP ne va utiliser que sa propriété BPM pour extraire le nombre de battements par minute actuel. Avec cette donnée en main, il crée la vue et aussi les commandes de l'interface utilisateur.

```
<%@ page language="java" errorPage="/jsp/Erreur.jsp" %>

<jsp:useBean id="modeleTempo" scope="request" class="teteptremiere.mix.vuedj.ModeleTempo" />

<html>
<head>
<title>Vue DJ</title>
</head>
<body>

<h1>Vue DJ</h1>
Battements par minute = <jsp:getProperty name="modeleTempo" property="BPM" />
<br />
<hr>
<br />

<form method="post" action="/vuedj/servlet/VueDJ">
BPM: <input type="text" name="bpm"
      value="<jsp:getProperty name="modeleTempo"
      property="BPM" />" />

&nbsp;

<input type="submit" name="definir" value="definir"><br />
<input type="submit" name="diminuer" value="<<">
<input type="submit" name="augmenter" value=">>"><br />
<input type="submit" name="on" value="on">
<input type="submit" name="off" value="off"><br />
</form>

</body>
</html>
```

Voici le bean que le modèle nous a transmis.

Début du code HTML.

Ici, nous extrayons la propriété BPM du bean modèle.

Puis nous générons la vue qui affiche le nombre actuel de battements par minute.

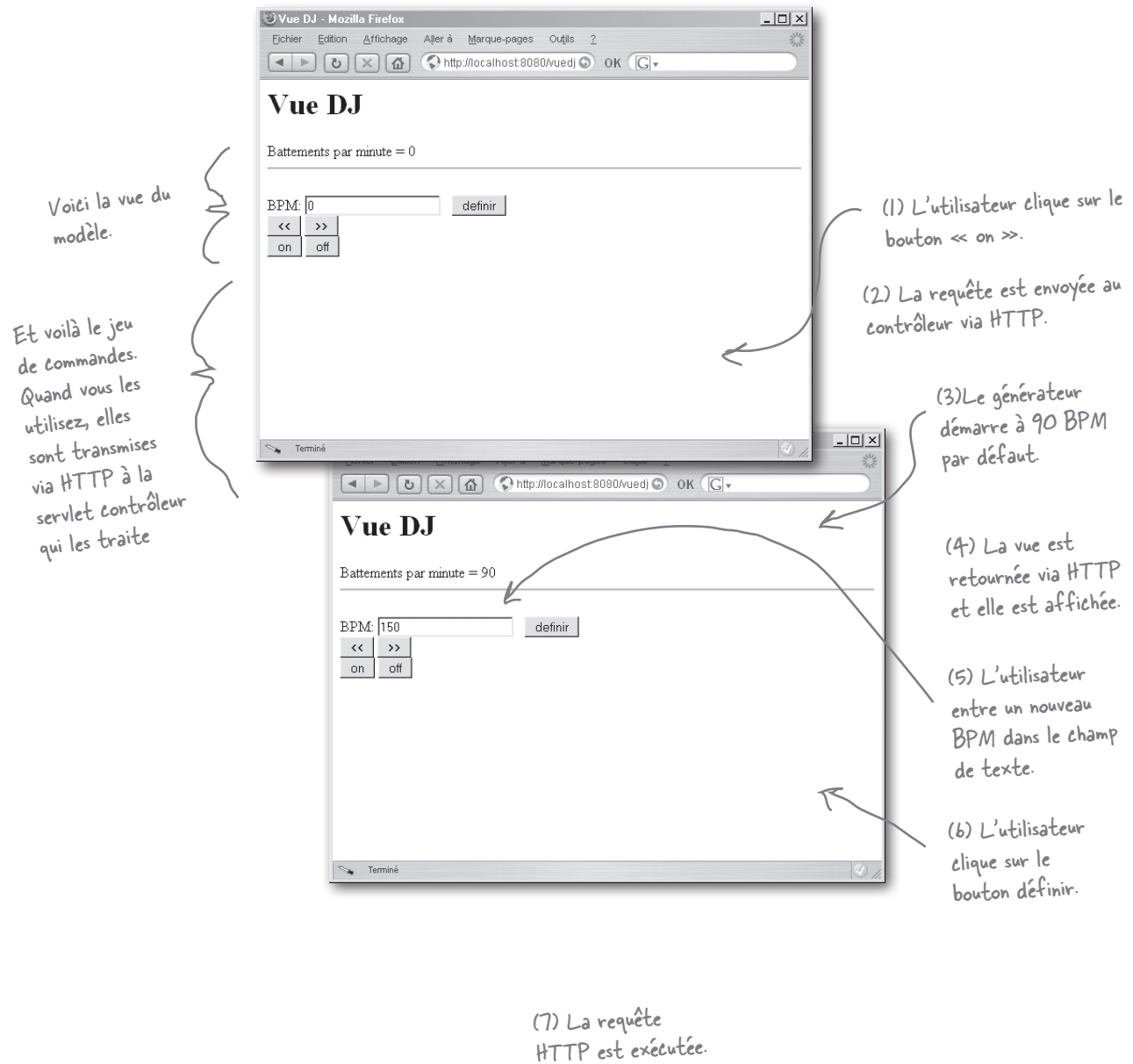
Et voici la partie « commandes » de la vue. Nous avons un champ de texte pour les BPM et des boutons pour démarrer, arrêter, augmenter et diminuer.

Et voici la fin du code HTML.

REMARQUEZ que, exactement comme dans MVC, la vue ne modifie pas le modèle (c'est la tâche du contrôleur) : elle utilise seulement son état !

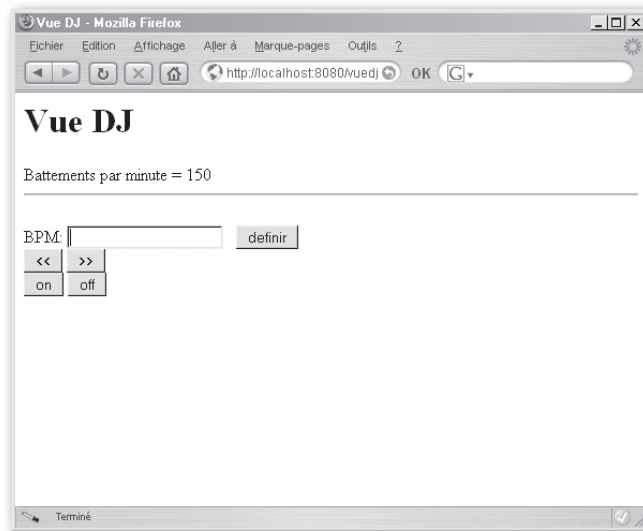
Tester Model 2...

Il est temps de lancer votre navigateur web, d'accéder à la servlet VueDJ et de tester...



(8) Le contrôleur modifie le modèle.

(9) La vue retourne une page HTML reflétant l'état actuel du modèle



Mode d'emploi

- ❶ **Premièrement, atteindre la page web. Vous verrez que les BPM sont à 0. Continuez et cliquez sur le bouton « on ».**
- ❷ **Vous devez maintenant voir la valeur par défaut des BPM : 90. Vous devez également entendre un rythme sur la machine sur laquelle le serveur s'exécute.**
- ❸ **Entrez un tempo spécifique, disons 120, et cliquez sur le bouton Définir. L'écran doit se rafraîchir et afficher la nouvelle valeur (et vous devez entendre les battements s'accélérer).**
- ❹ **Jouez maintenant avec les deux boutons pour accélérer et ralentir le tempo.**
- ❺ **Réfléchissez à la façon dont chaque partie du système fonctionne. L'interface HTML envoie une requête à la servlet (le contrôleur). La servlet l'analyse, envoie une requête au modèle puis transfère le contrôle à la JSP (la vue). Cette dernière crée la vue HTML qui est retournée et affichée.**

Design Patterns et Model 2

Après avoir implémenté les Commandes DJ pour le Web en utilisant Model 2, vous vous demandez peut être où sont passés les patterns. Nous avons une vue créée en HTML à partir d'une JSP, mais qui n'est plus un observateur du modèle. Nous avons un contrôleur, une servlet qui reçoit des requêtes HTTP, mais utilisons-nous toujours le pattern Stratégie ? Et qu'en est-il de Composite ? Nous avons une vue générée en HTML qui est affichée dans un navigateur web. S'agit-il toujours du pattern Composite ?

Model 2 est une adaptation de MVC pour le Web

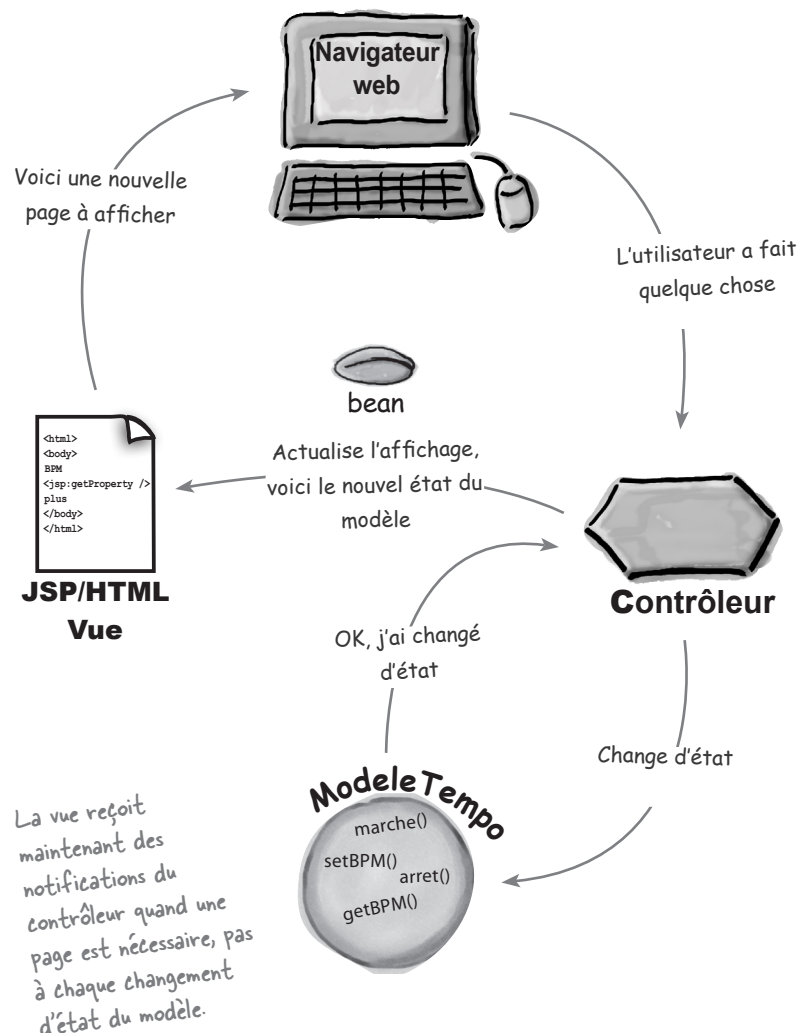
Même si Model 2 ne ressemble pas vraiment au MVC des manuels, tous ses constituants sont toujours là : ils ont simplement été adaptés pour refléter les idiosyncrasies du modèle d'un navigateur web. Regardons de nouveau...

Observateur

La vue n'est plus un observateur du modèle au sens classique du terme ; autrement dit, elle ne s'enregistre pas auprès du modèle pour être informée de ses changements d'état.

En revanche, la vue reçoit bien l'équivalent de notifications, mais elle les reçoit indirectement, car c'est le contrôleur qui l'informe que le modèle a changé. Le contrôleur lui transmet même un bean qui lui permet d'extraire l'état du modèle.

Si vous réfléchissez au modèle du navigateur, vous constaterez que la vue n'a besoin d'être informée des changements d'état que lorsqu'une réponse HTTP est retournée au navigateur ; à tout autre moment, une notification serait inutile. Ce n'est que lorsqu'une page est créée et retournée que la génération de la vue et l'incorporation de l'état du modèle ont un sens.

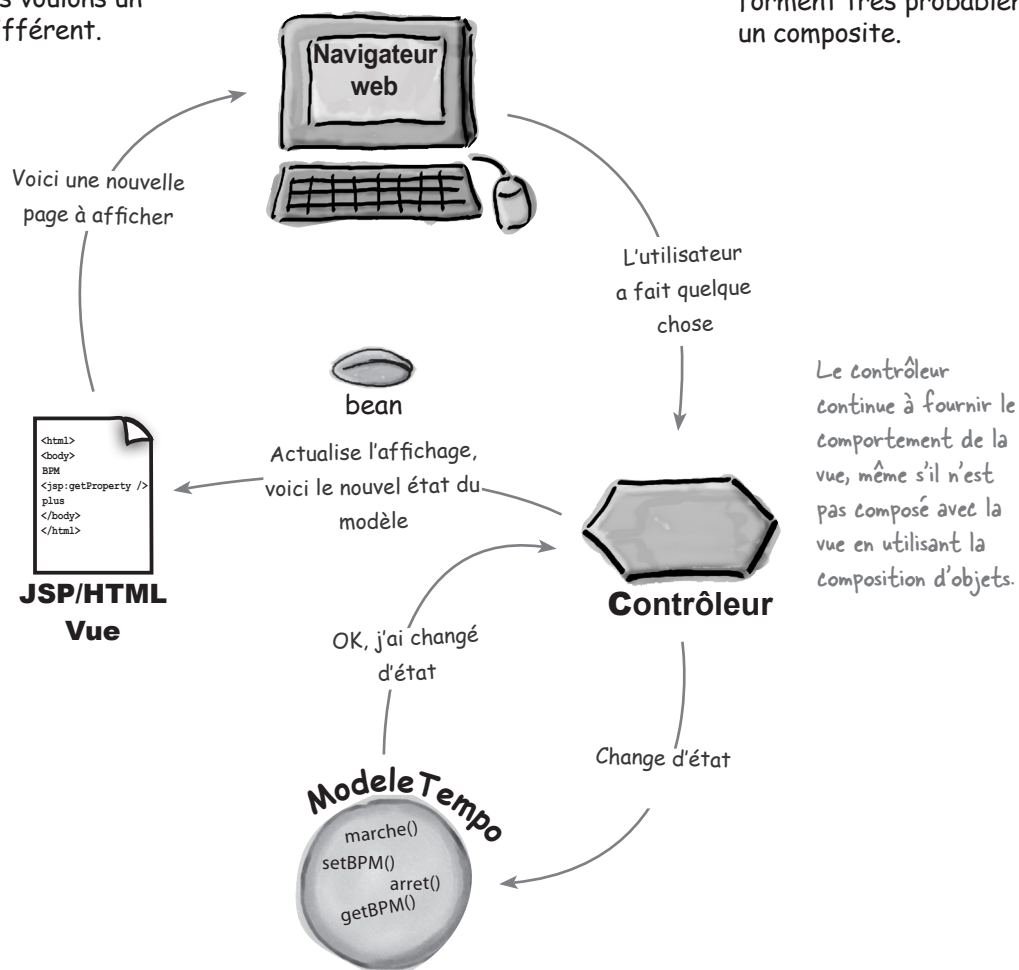


Stratégie

Dans Model 2, l'objet Stratégie est toujours la servlet contrôleur, mais elle n'est pas composée directement avec la vue de la façon classique. Cela dit, c'est un objet qui implémente un comportement pour la vue et nous pouvons le remplacer par un autre contrôleur si nous voulons un comportement différent.

Composite

Comme dans notre IHM Swing, la vue est finalement constituée d'un ensemble de composants graphiques. En l'occurrence, ils sont rendus par un navigateur web à partir d'une description HTML, mais ce qui sous-tend le tout, c'est un système d'objets qui forment très probablement un composite.



Il n'y a pas de questions stupides

Q: On dirait que vous insistez lourdement sur le fait que le pattern Composite est vraiment dans MVC. Est-il réellement là ?

R: Oui, il y a vraiment un pattern Composite dans MVC. Mais c'est une très bonne question. Aujourd'hui, les packages comme Swing sont devenus si sophistiqués que nous sommes à peine conscients de leur structure interne et de l'emploi de composites dans la construction et la mise à jour de l'affichage. C'est encore plus difficile à percevoir quand on est en présence d'un navigateur web capable de convertir en interface utilisateur un code écrit dans un langage à balises. À l'époque où MVC a été découvert, la création d'IHM nécessitait beaucoup plus d'interventions manuelles et l'appartenance du pattern à MVC était plus évidente.

Q: Arrive-t-il que le contrôleur s'occupe de la logique applicative ?

R: Non. Le contrôleur implémente un comportement pour la vue. C'est le petit malin qui traduit les actions provenant de la vue en actions sur le modèle. Le modèle reçoit ces actions et c'est lui qui implémente la logique applicative pour décider quoi faire en réponse à ces actions. Le contrôleur pourrait participer en déterminant quelles méthodes appeler sur le modèle, mais on ne peut pas parler de « logique applicative ». La véritable logique applicative, c'est le code qui gère et manipule vos données, et il réside dans le modèle.

Q: J'ai toujours eu un peu de mal à saisir le terme de modèle. Maintenant, je me rends compte que c'est le cœur de l'application, mais pourquoi avoir employé un mot aussi vague et aussi difficile à comprendre pour décrire cet aspect de MVC ?

R: Quand MVC a été baptisé, on a eu besoin d'un mot commençant par « M », sinon on n'aurait pas pu l'appeler MVC. Mais, sérieusement, nous sommes d'accord avec vous. Tout le monde se gratte la tête et se demande ce qu'est un modèle. Et puis tout le monde parvient à la conclusion qu'il ne voit pas de meilleur terme.

Q: Vous avez beaucoup parlé de l'état du modèle. Cela veut-il dire qu'il implémente le pattern État ?

R: Non, nous parlons de la notion générale d'état. Mais il existe certainement des modèles qui utilisent le pattern État pour gérer leurs états internes.

Q: J'ai lu des descriptions de MVC qui affirment que le contrôleur est un médiateur entre la vue et le modèle. Est-ce que le contrôleur implémente le pattern Médiateur ?

R: Comme nous n'avons pas abordé le pattern Médiateur (même si vous en trouverez une description dans l'annexe), nous n'entrerons pas dans les détails. Toutefois, le médiateur a pour fonction d'encapsuler la façon dont les objets interagissent et d'encourager le couplage faible en empêchant les références mutuelles explicites entre deux objets. Ainsi, jusqu'à un certain point, on peut voir le contrôleur comme un médiateur puisque la vue ne modifie jamais directement l'état du modèle mais qu'elle passe par le contrôleur. Mais n'oubliez pas que la vue a besoin d'une référence au modèle pour accéder à son état. Si le contrôleur était un vrai médiateur, la vue devrait également passer par le contrôleur pour accéder à l'état du modèle.

Q: Est-ce que la vue est toujours obligée de demander son état au modèle ? Ne pourrait-on pas appliquer le modèle « push » et envoyer l'état avec la notification de changement ?

R: Oui, le modèle pourrait sans doute envoyer son état avec la notification. En fait, si vous regardez de nouveau la vue JSP/HTML, c'est exactement ce que nous faisons. Nous transmettons la totalité du modèle dans un bean, et la vue accède à l'état dont elle a besoin en utilisant les propriétés du bean. Nous pourrions faire quelque chose de similaire avec le *ModeloTempo* en nous contentant d'envoyer l'état qui intéresse la vue. Mais si vous vous souvenez du chapitre sur le pattern Observateur, vous vous souvenez également que cette pratique présente quelques inconvénients si vous n'y regardez pas à deux fois.

Q: Si j'ai plusieurs vues, est-ce que j'ai toujours besoin de plusieurs contrôleurs ?

R: En général, vous avez besoin d'un contrôleur par vue au moment de l'exécution. Mais la même classe contrôleur peut facilement gérer plusieurs vues.

Q: La vue n'est pas censée manipuler le modèle. Pourtant, j'ai remarqué dans votre implémentation que la vue accédait à toutes les méthodes qui modifiaient l'état du modèle. N'est-ce pas dangereux ?

R: Vous avez raison ; nous avons permis à la vue d'accéder à toutes les méthodes du modèle. Nous l'avons fait pour simplifier, mais il peut y avoir des circonstances dans lesquelles vous voudrez limiter l'accès. Il y a un grand design pattern qui permet d'adapter une interface pour ne fournir qu'un sous-ensemble des méthodes. Savez-vous lequel ?



Votre boîte à outils de concepteur

Vous pourriez impressionner n'importe qui avec votre boîte à outils. Regardez tous ces principes, ces patterns... et maintenant des patterns composés !

Principes 00

Encapsulez ce qui varie.
Préférez la composition à l'héritage.
Programmez des interfaces, non des implémentations.
Efforcez-vous de coupler faiblement les objets qui interagissent.
Les classes doivent être ouvertes à l'extension mais fermées à la modification.
Dépendez des abstractions. Ne dépendez pas des classes concrètes.
Ne parlez qu'à vos amis.
Ne nous appelez pas, nous vous appellerons.
Une classe ne doit avoir qu'une seule raison de changer.

Bases de l'OO

Abstraction
Encapsulation
Polymorphisme
Héritage

Patterns 00

Proxy - fournit un remplaçant à un autre objet, pour en contrôler l'accès.

Patterns composés

Un Pattern Composé combine deux ou plusieurs patterns pour résoudre un problème général ou récurrent.

Nous avons une nouvelle catégorie ! MVC et Model 2 sont des patterns composés.



POINTS D'IMPACT

- Le pattern Modèle Vue Contrôleur (MVC) est un pattern composé constitué des patterns Observateur, Stratégie et Composite.
- Le modèle utilise le pattern Observateur pour pouvoir tenir les observateurs à jour tout en restant découplé d'eux.
- Le contrôleur est la stratégie pour la vue. La vue peut utiliser différentes implémentations du contrôleur pour avoir des comportements différents.
- La vue utilise le pattern Composite pour implémenter l'interface utilisateur. Celle-ci est généralement constituée de composants imbriqués, comme des fenêtres, des cadres et des boutons.
- Ces patterns collaborent pour découpler les trois acteurs du modèle MVC, ce qui préserve la clarté et la souplesse de la conception.
- On peut utiliser le pattern Adaptateur pour adapter un nouveau modèle à une vue et à un contrôleur existants.
- Model 2 est une adaptation de MVC aux applications web.
- Dans Model 2, la contrôleur est implémenté sous forme de servlet et la vue sous forme de JSP et de code HTML.



Solutions des exercices

À vos crayons

Le CompteurDeCouacs est aussi un Cancaneur. Quand nous modifions Cancaneur pour étendre CouacObservable, nous devons modifier toutes les classes qui implémentent Cancaneur, y compris CompteurDeCouacs :

CompteurDeCouacs étant un Cancaneur, il est maintenant aussi un CouacObservable.

```
public class CompteurDeCouacs implements Cancaneur {
    Cancaneur canard;
    static int nombreDeCouacs;
```

```
    public CompteurDeCouacs(Cancaneur canard) {
        this.canard = canard;
    }
```

```
    public void cancaner() {
        canard.cancaner();
        nombreDeCouacs++;
    }
```

```
    public static int getCouacs() {
        return nombreDeCouacs;
    }
```

Voici le canard que le CompteurDeCouacs décore. C'est ce canard qui doit réellement gérer les méthodes de CouacObservable.

Toute cette partie du code est identique à la précédente version de CompteurDeCouacs

```
    public void enregisterObservateur(Observateur observateur) {
        canard.enregistrerObservateur (observateur);
    }
```

```
    public void notifierObservateurs() {
        canard.notifierObservateurs();
    }
```

Voici les méthodes de CouacObservable. Remarquez que nous délégons simplement les deux appels au canard que nous décorons.

À vos crayons

Et si un Cancanologue veut observer une troupe entière ? Et d'abord, qu'est-ce que cela veut dire ? Vous pouvez voir les choses ainsi : si nous observons un composite, nous observons tout ce qu'il contient. Si vous vous enregistrez auprès d'une troupe, le composite (la troupe) s'assure que vous êtes enregistré auprès de tous ses enfants (pardon, tous ses petits cancaneurs), ce qui peut comprendre d'autres troupes.

Comme Troupe est Cancaneur, il est aussi maintenant un CouacObservable.

```
public class Troupe implements Cancaneur {
    ArrayList canards = new ArrayList();
```

Voici les Cancaneurs qui sont dans la Troupe.

```
    public void add(Cancaneur canard) {
        canards.add(canard);
    }
```

```
    public void cancaner() {
        Iterator itérateur = canards.itérateur();
        while (itérateur.hasNext()) {
            Cancaneur canard = (Cancaneur)itérateur.next();
            canard.cancaner();
        }
    }
```

Quand vous enregistrez un observateur auprès de la Troupe, vous êtes en réalité enregistré auprès de tout ce qui APPARTIENT à la troupe, autrement dit de chaque Cancaneur, que ce soit un canard ou une autre Troupe.

```
    public void enregistrerObservateur(Observateur observateur) {
        Iterator itérateur = canards.itérateur();
        while (itérateur.hasNext()) {
            Cancaneur canard = (Cancaneur)itérateur.next();
            canard.enregistrerObservateur(observateur);
        }
    }
```

Nous parcourons tous les Cancaneurs de la Troupe et déléguons l'appel à chaque Cancaneur. Si le Cancaneur est une autre Troupe, il fera de même.

```
    public void notifierObservateurs() { }
}
```

Puisque chaque Cancaneur émet ses propres notifications, la Troupe n'a pas besoin de s'en occuper. C'est ce qui se passe quand la Troupe délègue cancaner() à chaque Cancaneur de la Troupe.

À vos crayons



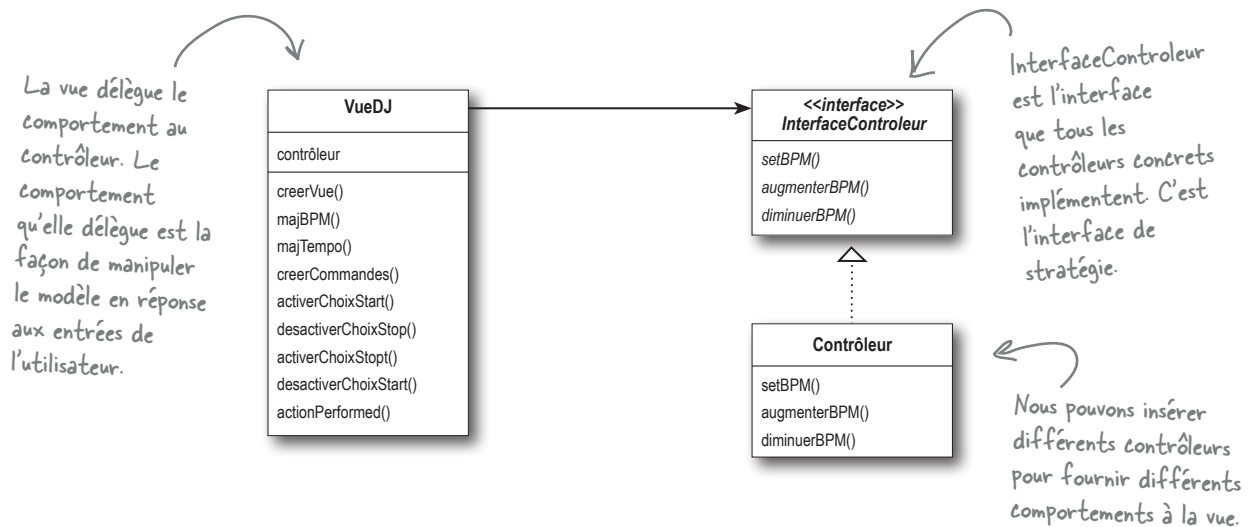
Nous continuons à instancier directement les Oies en nous appuyant sur des classes concrètes. Sauriez-vous écrire une Fabrique abstraite pour les Oies ? Comment générerait-elle la création de canards qui seraient des oies ?

Vous pourriez ajouter une méthode `creerCanardOie()` aux fabriques de canards existantes. Ou bien vous pourriez créer une fabrique totalement séparée pour créer des familles d'oies.



Problème de conception

Vous venez de voir que la vue et le contrôleur emploient tous deux le pattern Stratégie. Pouvez-vous tracer un diagramme de classes qui représente ce pattern ?





Code prêt
à l'emploi

Voici l'implémentation complète de l'application DJ. Elle montre tout le code MIDI pour générer le son et tous les composants Swing pour créer la vue. Vous pouvez également télécharger ce code en anglais sur <http://www.wickedlysmart.com> ou en français à l'adresse <http://www.oreilly.fr/catalogue/2841773507.html>. Amusez-vous bien !

```
package tetepremiere.mix.vuedj;

public class TestDJ {
    public static void main (String[] args) {
        InterfaceModeleTempo modele = new ModeleTempo();
        InterfaceControleur controleur = new ControleurTempo(modele);
    }
}
```

Le Modèle de Tempo

```
package tetepremiere.mix.vuedj;

public interface InterfaceModeleTempo {
    void initialiser();

    void marche();

    void arret();

    void setBPM(int bpm);

    int getBPM();

    void enregistrerObservateur(ObservateurBattements o);

    void supprimerObservateur(ObservateurBattements o);

    void enregistrerObservateur(ObservateurBPM o);

    void supprimerObservateur(ObservateurBPM o);
}
```



```

package tetepremiere.mix.vuedj;

import javax.sound.midi.*;
import java.util.*;

public class ModeleTempo implements InterfaceModeleTempo, MetaEventListener {
    Sequencer sequenceur;
    ArrayList observateursBattements = new ArrayList();
    ArrayList observateursBPM = new ArrayList();
    int bpm = 90;
    // autres variables d'instance
    Sequence sequence;
    Track piste;

    public void initialiser() {
        setUpMidi();
        construirePisteEtDemarrer();
    }

    public void marche() {
        sequenceur.start();
        setBPM(90);
    }

    public void arret() {
        setBPM(0);
        sequenceur.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequenceur.setTempoInBPM(getBPM());
        notifierObservateursBPM();
    }

    public int getBPM() {
        return bpm;
    }

    void evenementBattement() {
        notifierObservateursBattements();
    }

    public void registerObserver(ObservateurBattements o) {
        observateursBattements.add(o);
    }

    public void notifierObservateursBattements() {
        for(int i = 0; i < observateursBattements.size(); i++) {

```




Code prêt à l'emploi

```
        ObservateurBattements observateur = (ObservateurBattements)observateursBattements.
get(i);
        observateur.majTempo();
    }
}

public void registerObserver(ObservateurBPM o) {
    observateursBPM.add(o);
}

public void notifierObservateursBPM() {
    for(int i = 0; i < observateursBPM.size(); i++) {
        ObservateurBPM observateur = (ObservateurBPM) observateursBPM.get(i);
        observateur.majBPM();
    }
}

public void removeObserver(ObservateurBattements o) {
    int i = observateursBattements.indexOf(o);
    if (i >= 0) {
        observateursBattements.remove(i);
    }
}

public void removeObserver(ObservateurBPM o) {
    int i = observateursBPM.indexOf(o);
    if (i >= 0) {
        observateursBPM.remove(i);
    }
}

public void meta(MetaMessage message) {
    if (message.getType() == 47) {
        evenementBattement();
        sequenceur.start();
        setBPM(getBPM());
    }
}

public void setUpMidi() {
    try {
        sequenceur = MidiSystem.getSequencer();
    }
```



```

        sequenceur.open();
        sequenceur.addMetaEventListener(this);
        sequence = new Sequence(Sequence.PPQ, 4);
        piste = sequence.createTrack();
        sequenceur.setTempoInBPM(getBPM());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void construirePisteEtDemarrer() {
    int[] listePistes = {35, 0, 46, 0};

    sequence.deleteTrack(null);
    piste = sequence.createTrack();

    makeTracks(listePistes);
    piste.add(makeEvent(192, 9, 1, 0, 4));

    try {
        sequenceur.setSequence(sequence);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void makeTracks(int[] list) {
    for (int i = 0; i < list.length; i++) {
        int key = list[i];

        if (key != 0) {
            piste.add(makeEvent(144, 9, key, 100, i));
            piste.add(makeEvent(128, 9, key, 100, i+1));
        }
    }
}

public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent evenement = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        evenement = new MidiEvent(a, tick);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return evenement;
}
}

```


La Vue

Code prêt à l'emploi



```
package tetepremiere.mix.vuedj;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class VueDJ implements ActionListener, ObservateurBattements, ObservateurBPM {
    InterfaceModeleTempo modele;
    InterfaceControleur controleur;
    JFrame cadreVue;
    JPanel panneauVue;
    BarrePulsante barre;
    JLabel affichageBPM;
    JFrame cadreControle;
    JPanel panneauControle;
    JLabel labelBPM;
    JTextField champTxtBPM;
    JButton definirBPM;
    JButton augmenterBPM;
    JButton diminuerBPM;
    JMenuBar barreMenus;
    JMenu menu;
    JMenuItem choixStart;
    JMenuItem choixStop;

    public VueDJ(InterfaceControleur controleur, InterfaceModeleTempo modele) {
        this.controleur = controleur;
        this.modele = modele;
        modele.registerObserver((ObservateurBattements)this);
        modele.registerObserver((ObservateurBPM)this);
    }

    public void creerVue() {
        // Création de tous les composants Swing
        panneauVue = new JPanel(new GridLayout(1, 2));
        cadreVue = new JFrame("Vue");
        cadreVue.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadreVue.setSize(new Dimension(100, 80));
        affichageBPM = new JLabel("hors ligne", SwingConstants.CENTER);
        barre = new BarrePulsante();
        barre.setValue(0);
        JPanel panneauBMP = new JPanel(new GridLayout(2, 1));
        panneauBMP.add(barre);
        panneauBMP.add(affichageBPM);
        panneauVue.add(panneauBMP);
        cadreVue.getContentPane().add(panneauVue, BorderLayout.CENTER);
        cadreVue.pack();
        cadreVue.setVisible(true);
    }
}
```



```

public void creerCommandes() {
// Création de tous les composants Swing
JFrame.setDefaultLookAndFeelDecorated(true);
cadreControle = new JFrame("Commandes");
cadreControle.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
cadreControle.setSize(new Dimension(100, 80));

panneauControle = new JPanel(new GridLayout(1, 2));

barreMenus = new JMenuBar();
menu = new JMenu("Commandes DJ");
choixStart = new JMenuItem("Démarrer");
menu.add(choixStart);
choixStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evenement) {
        controleur.start();
    }
});
choixStop = new JMenuItem("Arrêter");
menu.add(choixStop);
choixStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evenement) {
        controleur.stop();
        //affichageBPM.setText("hors ligne");
    }
});
JMenuItem exit = new JMenuItem("Quitter");
exit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evenement) {
        System.exit(0);
    }
});

menu.add(exit);
barreMenus.add(menu);
cadreControle.setJMenuBar(barreMenus);

champTxtBPM = new JTextField(2);
labelBPM = new JLabel("Entrer BPM:", SwingConstants.RIGHT);
definirBPM = new JButton("Définir");
definirBPM.setSize(new Dimension(10,40));
augmenterBPM = new JButton(">>");
diminuerBPM = new JButton("<<");
definirBPM.addActionListener(this);
augmenterBPM.addActionListener(this);
diminuerBPM.addActionListener(this);

```


Code prêt à l'emploi



```
JPanel panneauBoutons = new JPanel(new GridLayout(1, 2));

panneauBoutons.add(diminuerBPM);
panneauBoutons.add(augmenterBPM);

JPanel panneauSaisie = new JPanel(new GridLayout(1, 2));

panneauSaisie.add(labelBPM);
panneauSaisie.add(champTxtBPM);

JPanel panneauControleInterne = new JPanel(new GridLayout(3, 1));
panneauControleInterne.add(panneauSaisie);
panneauControleInterne.add(definirBPM);
panneauControleInterne.add(panneauBoutons);
panneauControle.add(panneauControleInterne);

labelBPM.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
affichageBPM.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));

cadreControle.getRootPane().setDefaultButton(definirBPM);
cadreControle.getContentPane().add(panneauControle, BorderLayout.CENTER);

cadreControle.pack();
cadreControle.setVisible(true);
}

public void activerChoixStop() {
    choixStop.setEnabled(true);
}

public void desactiverChoixStop() {
    choixStop.setEnabled(false);
}

public void activerChoixStart() {
    choixStart.setEnabled(true);
}

public void desactiverChoixStart() {
    choixStart.setEnabled(false);
}
```



```

public void actionPerformed(ActionEvent evenement) {
    if (evenement.getSource() == definirBPM) {
        int bpm = Integer.parseInt(champTxtBPM.getText());
        controleur.setBPM(bpm);
    } else if (evenement.getSource() == augmenterBPM) {
        controleur.augmenterBPM();
    } else if (evenement.getSource() == diminuerBPM) {
        controleur.diminuerBPM();
    }
}

public void majBPM() {
    int bpm = modele.getBPM();
    if (bpm == 0) {
        affichageBPM.setText("hors ligne");
    } else {
        affichageBPM.setText("Nombre de BPM : " + modele.getBPM());
    }
}

public void majTempo() {
    barre.setValue(100);
}
}

```

Le Contrôleur

```

package tetepremiere.mix.vuedj;

public interface InterfaceContrôleur {
    void start();
    void stop();
    void augmenterBPM();
    void diminuerBPM();
    void setBPM(int bpm);
}

```


Code prêt à l'emploi



```
package tetepremiere.mix.vuedj;

public class ControleurTempo implements InterfaceControleur {
    InterfaceModeleTempo modele;
    VueDJ vue;

    public ControleurTempo(InterfaceModeleTempo modele) {
        this.modele = modele;
        vue = new VueDJ(this, modele);
        vue.creerVue();
        vue.creerCommandes();
        vue.desactiverChoixStop();
        vue.activerChoixStart();
        modele.initialiser();
    }

    public void start() {
        modele.marche();
        vue.desactiverChoixStop();
        vue.activerChoixStart();
    }

    public void stop() {
        modele.arret();
        vue.desactiverChoixStop();
        vue.activerChoixStart();
    }

    public void augmenterBPM() {
        int bpm = modele.getBPM();
        modele.setBPM(bpm + 1);
    }

    public void diminuerBPM() {
        int bpm = modele.getBPM();
        modele.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        modele.setBPM(bpm);
    }
}
```


Le modèle du Cœur

```

package tetepremiere.mix.vuedj;

public class TestCoeur {
    public static void main (String[] args) {
        ModeleCoeur modeleCoeur = new ModeleCoeur();
        InterfaceControleur modele = new ControleurCoeur(modeleCoeur);
    }
}

package tetepremiere.mix.vuedj;

public interface InterfaceModeleCoeur {
    int getRythmeCardiaque();
    void registerObserver(ObservateurBattements o);
    void removeObserver(ObservateurBattements o);
    void registerObserver(ObservateurBPM o);
    void removeObserver(ObservateurBPM o);
}

package tetepremiere.mix.vuedj;
import java.util.*;

public class ModeleCoeur implements InterfaceModeleCoeur, Runnable {
    ArrayList observateursBattements = new ArrayList();
    ArrayList observateursBPM = new ArrayList();
    int temps = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public ModeleCoeur() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int dernierRythme = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }

            int rythme = 60000/(temps + change);
            if (rythme < 120 && rythme > 50) {
                temps += change;
                notifierObservateursBattements();
            }
        }
    }
}

```




```

        if (rythme != dernierRythme) {
            dernierRythme = rythme;
            notifierObservateursBPM();
        }
    }
    try {
        Thread.sleep(temps);
    } catch (Exception e) {}
}

public int getRythmeCardiaque() {
    return 60000/temps;
}

public void registerObserver(ObservateurBattements o) {
    observateursBattements.add(o);
}

public void removeObserver(ObservateurBattements o) {
    int i = observateursBattements.indexOf(o);
    if (i >= 0) {
        observateursBattements.remove(i);
    }
}

public void notifierObservateursBattements() {
    for(int i = 0; i < observateursBattements.size(); i++) {
        ObservateurBattements observateur = (ObservateurBattements)observateursBattements.get(i);
        observateur.majTempo();
    }
}

public void registerObserver(ObservateurBPM o) {
    observateursBPM.add(o);
}

public void removeObserver(ObservateurBPM o) {
    int i = observateursBPM.indexOf(o);
    if (i >= 0) {
        observateursBPM.remove(i);
    }
}

public void notifierObservateursBPM() {
    for(int i = 0; i < observateursBPM.size(); i++) {
        ObservateurBPM observateur = (ObservateurBPM)observateursBPM.get(i);
        observateur.majBPM();
    }
}
}

```


L'Adaptateur du Cœur

```
package tetepremiere.mix.vuedj;

public class AdaptateurCoeur implements InterfaceModeleTempo {
    InterfaceModeleCoeur coeur;

    public AdaptateurCoeur(InterfaceModeleCoeur coeur) {
        this.coeur = coeur;
    }

    public void initialiser() {}

    public void marche() {}

    public void arret() {}

    public int getBPM() {
        return coeur.getRythmeCardiaque();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(ObservateurBattements o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(ObservateurBattements o) {
        coeur.removeObserver(o);
    }

    public void registerObserver(ObservateurBPM o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(ObservateurBPM o) {
        coeur.removeObserver(o);
    }
}
```


Code prêt à l'emploi



Le Contrôleur

```
package tetepremiere.mix.vuedj;

public class ControleurCoeur implements InterfaceControleur {
    InterfaceModeleCoeur modele;
    VueDJ vue;

    public ControleurCoeur(InterfaceModeleCoeur modele) {
        this.modele = modele;
        vue = new VueDJ(this, new AdaptateurCoeur(modele));
        vue.creerVue();
        vue.creerCommandes();
        vue.desactiverChoixStop();
        vue.desactiverChoixStart();
    }

    public void start() {}

    public void stop() {}

    public void augmenterBPM() {}

    public void diminuerBPM() {}

    public void setBPM(int bpm) {}
}
```


13 mieux vivre avec les patterns

✧ ***Les patterns dans le monde réel*** ✧ ✧ ✧



Maintenant, vous êtes prêt à vivre dans un monde nouveau, un monde peuplé de Design Patterns. Mais avant de vous laisser ouvrir toutes ces nouvelles portes, nous devons aborder quelques détails que vous rencontrerez dans le monde réel. Eh oui, les choses y sont un peu plus complexes qu'elles ne le sont ici, à Objectville. Suivez-nous ! Vous trouverez à la page suivante un guide extra qui vous aidera à effectuer la transition...

Le guide d'Objectville

Mieux vivre avec les Design Patterns



Merci d'avoir choisi notre guide. Il contient de nombreux trucs et astuces pour vivre avec les patterns dans le monde réel. Dans ce guide, vous allez :

- Apprendre quelles sont les idées fausses les plus courantes sur la définition d'un « design pattern ».
- Découvrir qu'il existe de superbes catalogues de patterns et comprendre pourquoi vous devez absolument vous en procurer un.
- Eviter la honte associée à l'emploi inopportun d'un design pattern.
- Apprendre à reconnaître à quelles catégories les patterns appartiennent.
- Voir que la découverte de patterns n'est pas réservée aux gourous ; lisez notre référence rapide et devenez vous aussi un auteur de patterns.
- Etre présent quand la vraie identité de la mystérieuse Bande des quatre sera révélée.
- Apprendre à ne pas vous laisser distancer – les livres de chevet que tout utilisateur de patterns se doit de posséder.
- Apprendre à exercer votre esprit comme un maître zen.
- Découvrir comment vous faire des amis et impressionner vos collègues développeurs en améliorant votre vocabulaire.

Design Pattern : définition

Après avoir lu cet ouvrage, vous avez très certainement une assez bonne idée de ce qu'est un pattern. Pourtant, nous n'avons jamais vraiment fourni de définition d'un design pattern. Eh bien, celle que l'on donne couramment risque de vous surprendre légèrement :

Un pattern est une solution à un problème dans un contexte.

Plutôt sibyllin comme définition, non ? Ne vous inquiétez pas, nous allons décortiquer chacun de ses composants : contexte, problème et solution :

Le **contexte** est la situation dans laquelle le pattern s'applique. Celle-ci doit être récurrente.

Le **problème** désigne le but que vous essayez d'atteindre dans ce contexte, ainsi que toutes les contraintes qui peuvent s'y présenter.

La **solution** est ce que vous recherchez : une conception générique que tout le monde peut appliquer et qui permet d'atteindre l'objectif en respectant l'ensemble des contraintes.

Exemple : Vous avez une collection d'objets.

Vous devez parcourir ces objets sans exposer l'implémentation de la collection.

Encapsulez l'itération dans une classe séparée.

C'est l'une de ces définitions qu'il faut un peu de temps pour assimiler, mais vous pouvez la décomposer en étapes. Voici un petit procédé mnémotechnique que vous pouvez vous répéter pour vous en souvenir :

« Si vous vous trouvez dans un contexte avec un problème qui implique un but qui est affecté par un ensemble de contraintes, vous pouvez appliquer une conception qui permet d'atteindre le but, de résoudre les contraintes et de déboucher sur une solution. »

Vous pensez sans doute que cela représente beaucoup de travail uniquement pour parvenir à comprendre ce qu'est un design pattern. Après tout, vous savez déjà qu'un pattern fournit une solution à un problème de conception récurrent. Où donc tout ce formalisme nous mène-t-il ? Eh bien, vous allez voir que le fait de disposer d'une façon formelle de décrire les patterns nous permet de créer des catalogues de patterns, ce qui présente toutes sortes d'avantages.



Vous avez peut-être raison. Réfléchissons un peu... Il nous faut un *problème*, un *contexte* et une *solution* :

Problème : Comment faire pour arriver à l'heure au bureau ?

Contexte : J'ai claqué la porte avec mes clés dans la voiture.

Solution : Casser la vitre, monter dans la voiture, démarrer et se rendre au bureau.

Nous avons tous les composants de la définition : nous avons un problème, un but – aller travailler – et des contraintes de temps et de distance probablement assorties d'autres facteurs. Nous avons également un contexte, dans lequel les clés du véhicule sont inaccessibles. Enfin, nous avons une solution qui nous permet de récupérer les clés et de respecter les contraintes de temps et de distance. Nous devons donc être en présence d'un pattern ! Qu'en pensez-vous ?



MUSCLEZ VOS NEURONES

Nous avons suivi notre définition et décrit un problème, un contexte et une solution (qui fonctionne !). S'agit-il pour autant d'un pattern ? Sinon, pourquoi ? Pourrions-nous commettre la même erreur en définissant un design pattern OO ?

Regardons la définition de plus près

La prochaine fois que quelqu'un vous dit qu'un pattern est une solution à un problème dans un contexte, hochez la tête et souriez. Vous savez ce qu'il veut dire, même si ce n'est pas une définition suffisante pour expliquer ce qu'est réellement un Design Pattern.

Notre exemple a tout l'air de correspondre à la définition d'un Design Pattern, mais ce n'est pas un vrai pattern. Pourquoi ? Pour commencer, nous savons qu'un pattern doit pouvoir s'appliquer à un problème récurrent. S'il est possible à une personne distraite d'oublier fréquemment ses clés dans sa voiture, briser la vitre ne constitue pas une solution applicable de nombreuses fois (du moins si nous tenons compte d'une contrainte supplémentaire : le coût).

Il comprend également deux autres erreurs. Premièrement, il n'est pas facile de prendre cette description et de la fournir à quelqu'un qui s'en servira pour résoudre son propre problème. Deuxièmement, nous avons enfreint un principe important mais simple : nous ne l'avons même pas nommé ! En l'absence de nom, le pattern ne nous permet pas de partager un vocabulaire commun avec d'autres développeurs.

Heureusement, les patterns ne sont pas simplement définis et documentés sous la forme « problème-contexte-solution » et nous disposons de bien meilleures façons de les décrire et de les rassembler dans des *catalogues de patterns*.



Q: Vais-je rencontrer des descriptions de patterns exprimées sous la forme problème-contexte-solution ?

R: Ces descriptions, que vous trouvez généralement dans les catalogues de patterns, sont d'ordinaire un peu plus explicites. Nous allons aborder en détail ces catalogues de patterns dans une minute, mais sachez qu'ils mentionnent l'intention et la motivation du pattern, les circonstances dans lesquelles il peut s'appliquer ainsi que la conception de la solution et les conséquences (positives ou non) de son emploi.

Q: Ai-je le droit de modifier légèrement la structure d'un pattern pour l'adapter à ma conception ? Ou bien suis-je obligé d'appliquer strictement la définition ?

R: Bien sûr, vous pouvez la modifier. À l'instar des principes de conception, les patterns ne sont pas censés être des lois ou des règles. Ce sont des lignes directrices que vous pouvez adapter en fonction de vos besoins. Comme vous l'avez constaté, de nombreux exemples du monde réel s'accommodent mal des définitions classiques.

Mais quand vous adaptez des patterns, cela ne fait jamais de mal d'expliquer en quoi votre pattern diffère de la conception classique – cela permet aux autres développeurs de reconnaître rapidement les patterns que vous utilisez, ainsi que les différences qu'ils présentent avec les patterns canoniques.

Q: Où puis-je me procurer un catalogue de patterns ?

R: Le premier catalogue de patterns, qui fait autorité, est *Design Patterns : Catalogue de modèles de conception réutilisables*, de Gamma, Helm, Johnson & Vlissides (Vuibert informatique). Cet ouvrage décrit vingt-trois patterns fondamentaux, et nous en parlerons plus en détail dans quelques pages.

De nombreux catalogues de patterns commencent à paraître dans différents domaines, notamment les logiciels d'entreprise, les systèmes concurrents et les systèmes de gestion.



Trucs de geeks

Que la force soit avec vous

La définition du Design Pattern nous dit que le problème est constitué d'un but et d'un ensemble de contraintes. Les gourous des patterns ont un terme pour désigner ces composants : ils parlent de forces. Pourquoi ? Eh bien ils ont certainement leurs propres raisons, mais si vous vous souvenez du film, la force façonne et contrôle l'Univers.

De même, dans la définition du pattern, les forces façonnent et contrôlent la solution. Ce n'est que lorsque les deux aspects de la force (le côté lumineux, votre but, et le côté sombre, les contraintes) sont équilibrés que vous avez un pattern utile.

Ce terme de « force » peut être déroutant la première fois que vous le rencontrez dans la présentation d'un pattern, mais il suffit de se souvenir que la force présente deux aspects (les buts et les contraintes) et qu'il faut les équilibrer ou les résoudre pour créer une solution. Ne vous laissez pas impressionner par le jargon, et que la force soit avec vous !



Paul : Mets-nous au courant, Yann. J'ai seulement lu quelques articles sur les patterns ici et là.

Yann : Bien sûr. Chaque catalogue aborde un ensemble de patterns, décrit chacun d'eux en détail et explique leurs relations avec les autres patterns.

Guy : Es-tu en train de dire qu'il existe plusieurs catalogues de patterns ?

Yann : Naturellement. Il y a des catalogues pour les patterns fondamentaux, et d'autres pour des patterns spécifiques à un domaine, par exemple les EJB.

Paul : Et lequel regardes-tu ?

Yann : C'est le catalogue GoF classique ; il contient vingt-trois design patterns fondamentaux.

Paul : GoF?

Yann : Oui, les initiales de Gang of Four, bande des quatre en français. La bande des quatre, ce sont les auteurs qui ont assemblé le premier catalogue de patterns.

Guy : Et qu'est-ce qu'il y a dans le catalogue ?

Yann : Un ensemble de patterns apparentés. La description de chaque pattern observe toujours la même structure et fournit de nombreux détails. Par exemple, chaque pattern porte *un nom*.

Paul : Diable ! C'est la révélation de l'année – un nom ! Voyez-vous ça.

Yann : Pas si vite, Paul. En fait, le nom est réellement important. Quand un pattern est nommé, nous disposons d'un moyen d'en parler, tu sais, toute cette histoire de vocabulaire partagé...

Paul : D'accord, d'accord, je plaisantais. Continue. Qu'y a-t-il d'autre ?

Yann : Eh bien, comme je le disais, la description de chaque pattern observe la même structure. Pour chacun, nous avons un nom et un certain nombre de sections qui l'expliquent en détail. Par exemple, il y a une section Intention qui expose ce qu'est le pattern, comme une sorte de définition. Puis il y a les sections Motivation et Indications d'utilisation qui décrivent quand et où le pattern peut être employé.

Guy : Et la conception elle-même ?

Yann : Il y a plusieurs sections qui décrivent la conception, avec toutes les classes qui la constituent et le rôle qu'elles jouent. Il y a aussi une section qui explique comment implémenter le pattern, et souvent des exemples de code.

Paul : On dirait qu'ils ont pensé à tout.

Yann : Ce n'est pas tout. Il y a aussi des exemples d'utilisation des patterns dans des systèmes réels, et une section que je considère comme l'une des plus utiles : celle qui explique les rapports du pattern avec *d'autres patterns*.

Paul : Oh, tu veux dire qu'ils disent par exemple en quoi *État* et *Stratégie* diffèrent.

Yann : Exactement !

Guy : Alors, Yann, comment utilises-tu le catalogue ? Quand tu as un problème, tu vas à la pêche pour y trouver une solution ?

Yann : J'essaie d'abord de me familiariser avec tous les patterns et leurs relations. Ensuite, quand j'ai besoin d'un pattern, j'ai déjà une bonne idée de celui qu'il me faut et je lis les sections Motivation et Indications d'utilisation pour vérifier que c'est le bon. Il y a aussi une section très importante, nommée Conséquences. Je la consulte pour être sûr qu'il n'y aura pas d'effets de bord accidentels sur ma conception.

Paul : C'est logique. Donc, une fois que tu sais que tu as le bon pattern comment procèdes-tu pour l'incorporer à ta conception et l'implémenter ?

Yann : C'est là que les diagrammes de classes entrent en jeu. Je commence par lire la section Structure pour avoir une vue d'ensemble avec le diagramme, puis la section Participants pour vérifier que j'ai bien compris le rôle de chaque classe. À partir de là, j'incorpore le pattern à ma conception, en apportant toutes les modifications nécessaires pour l'adapter. Enfin, je lis les sections Implémentation et Exemples de code pour être certain de connaître toutes les bonnes techniques d'implémentation et les pièges que je pourrais rencontrer.

Guy : Je vois comment un catalogue pourrait m'aider à me mettre plus rapidement aux patterns !

Paul : Absolument. Yann, pourrais-tu nous faire faire une visite commentée ?

Dans un catalogue, tous les patterns commencent par un nom. Le nom est une composante vitale du pattern – en l'absence de nom bien choisi, il ne peut pas faire partie du vocabulaire que vous partagez avec les autres développeurs.

La section Motivation présente un scénario concret qui décrit le problème et la façon dont la solution résout celui-ci.

La section Indications d'utilisation décrit les situations dans lesquelles le pattern peut être appliqué.

Les participants sont les classes et les objets de la conception. Cette section décrit leurs responsabilités et le rôle qu'ils jouent dans le pattern.

Les conséquences décrivent les effets possibles – positifs ou négatifs – de l'utilisation de ce pattern.

La section Implémentation fournit les techniques que vous devez employer pour implémenter ce pattern et signale les problèmes auxquels vous devez être attentif.

La section Utilisations remarquables décrit des exemples de ce pattern rencontrés dans des systèmes réels.

SINGLETON

Objet Créateur

Intention

Et aliquat, velasus ent lare feus acillao rperci tat, quat nonsequam il ea at nim nos do enim qui cratio ex ea faci tet, sequi diu utat, volare magni. Ral modore du larect augiam iril el dipis dionsequi dignibz cumty nibh esequat. Dois malpitem ipisim escece conullut wisi.

Motivation

Et aliquat, velasus ent lare feus acillao rperci tat, quat nonsequam il ea at nim nos do enim qui cratio ex ea faci tet, sequi diu utat, volare magni. Ral modore du larect augiam iril el dipis dionsequi dignibz cumty nibh esequat. Dois malpitem ipisim escece conullut wisi.

Indications d'utilisation

Dois malpitem ipisim escece conullut wisi. Exem ad magna aliquid blamet, conullandere dolore magna feus nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Structure

Singleton
static Singleton uniqueInstance;
/* Autres variables d'instance */
static Singleton()
/* Autres méthodes */

Participants

Dois malpitem ipisim escece conullut wisi. Exem ad magna aliquid blamet, conullandere dolore magna feus nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Collaborations

• Feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore.

Conséquences

Dois malpitem ipisim escece conullut wisi. Exem ad magna aliquid blamet, conullandere dolore magna feus nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Implémentation/Exemples de code

```

public class Singleton {
    private static Singleton uniqueInstance;

    // autres variables d'instance

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // autres méthodes
}

```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Utilisations remarquables

Dois malpitem ipisim escece conullut wisi. Exem ad magna aliquid blamet, conullandere dolore magna feus nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Patterns apparentés

Elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er, alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ca feupit ing ent laore magnibz entat wissaece et, suscilla ad mininci blam dolorep reilit iri, conse dolore dolore et, verci entis entit ip elecequid ut ad escecem ing ea con eros autem diam nomullu ptatis isomodignibz er.

Voici la classification ou catégorie du pattern. Nous en parlerons dans quelques pages.

L'intention décrit brièvement la fonction du pattern. Vous pouvez également la considérer comme sa définition (c'est ainsi que nous l'avons nommée dans ce livre).

La section Structure contient un diagramme qui illustre les relations entre les classes qui participent au pattern.

La section Collaborations nous indique comment les participants collaborent dans le pattern.

Les exemples de code sont des fragments de programme qui peuvent vous aider dans votre implémentation.

La section Patterns apparentés décrit les relations que ce pattern entretient avec d'autres.

Il n'y a pas de questions stupides

Q: Est-il possible de créer ses propres design patterns ou bien faut-il être un « gourou » pour pouvoir le faire ?

R: Tout d'abord, souvenez-vous que les patterns sont découverts, pas créés. En conséquence, n'importe qui peut découvrir un design pattern puis écrire sa description. Mais ce n'est pas facile, et ce n'est ni rapide ni fréquent. Être un auteur de patterns demande beaucoup d'investissement.

Demandez-vous d'abord pourquoi vous en avez envie : la plupart des informaticiens n'écrivent pas de patterns, ils se contentent de les utiliser. Mais vous travaillez peut-être dans un domaine spécialisé dans lequel vous pensez que de nouveaux patterns seraient utiles, ou peut-être êtes vous tombé par hasard sur une solution à ce que vous considérez comme un problème récurrent. Il se peut aussi que vous vouliez simplement vous impliquer dans la communauté des patterns et contribuer à étoffer le corpus existant.

Q: Je suis partant. Comment commencer ?

R: Comme dans toute discipline, plus vous en saurez mieux ce sera. L'étude des patterns existants, de ce qu'ils font et de leurs relations avec d'autres patterns, est cruciale. Non seulement elle vous permet de vous familiariser avec la façon dont les patterns sont forgés, mais elle vous évite de réinventer la roue. De là, vous voudrez sans doute commencer à jeter vos propres patterns sur le papier afin de pouvoir en faire part à d'autres développeurs ; nous verrons comment procéder pour communiquer vos patterns dans un instant. Si vous êtes réellement intéressé, lisez la section qui suit ces « questions-réponses ».

Q: Comment saurai-je que j'ai réellement découvert un pattern ?

R: C'est une très bonne question : vous n'avez pas de pattern tant que d'autres développeurs ne l'ont pas utilisé et constaté qu'il fonctionnait. En général, un pattern n'en est pas un tant qu'il n'a pas réussi le test de la « règle des trois ». Cette règle énonce qu'on ne peut qualifier un pattern de pattern que s'il a été appliqué au moins trois fois dans une situation du monde réel.

Donc vous voulez être une star des patterns ?

Écoutez d'abord ce que je vous dis.

Achetez un catalogue de patterns,

Passez-y du temps, étudiez-le bien.

Et quand votre description s'ra au point,

Et qu'trois développeurs seront d'votre avis,

Alors vous saurez que c'est un pattern.

↑
Sur l'air de « So you wanna be a Rock'n Roll Star ».

Donc, vous voulez être auteur de patterns...

Étudiez. Vous devez être versé dans les patterns existants avant de pouvoir en créer un nouveau. La plupart des patterns qui semblent nouveaux ne sont en réalité que des variantes de ceux qui existent déjà. L'étude des patterns vous permet de les reconnaître de mieux en mieux et d'apprendre à les mettre en relation avec d'autres patterns.

Prenez le temps de réfléchir et d'évaluer. C'est de votre expérience – les problèmes que vous avez rencontrés et les solutions que vous avez apportées – que naissent les idées de patterns. Prenez donc le temps de réfléchir à ces expériences et d'y rechercher des solutions innovantes et récurrentes. N'oubliez pas que la plupart des conceptions sont des variations sur des patterns existants, non de nouveaux patterns. Et si vous trouvez quelque chose de nouveau, son champ d'application peut être trop étroit pour qu'on puisse vraiment le qualifier de pattern.

Couchez vos idées sur le papier de façon à être compris par les autres. Identifier de nouveaux patterns ne sert pas à grand-chose si les autres ne peuvent pas exploiter votre découverte : vous devez documenter vos candidats au statut de pattern de telle sorte que les autres puissent les lire, les comprendre et les appliquer à leur propre solution puis vous fournir un retour. Heureusement, vous n'aurez pas à inventer votre propre méthode pour documenter vos patterns. Comme vous l'avez constaté avec le format « GoF », beaucoup de réflexion a déjà été investi dans la façon de décrire les patterns et leurs caractéristiques.

Demandez aux autres d'essayer vos patterns ; puis affinez et affinez encore. Ne vous attendez pas à ce que votre pattern soit correct du premier coup. Considérez-le comme un projet en cours qui s'améliorera au fil du temps. Demandez à d'autres développeurs d'étudier votre pattern, de le tester et de vous donner leur avis. Tenez-en compte dans votre description et réessayez. Cette description ne sera jamais parfaite, mais viendra un moment où elle sera suffisamment solide pour que d'autres développeurs puissent la lire et la comprendre.

N'oubliez pas la règle des trois. Souvenez-vous : tant que votre pattern n'a pas été appliqué avec succès dans trois solutions du monde réel, il n'a pas droit au titre de pattern. Voilà une autre bonne raison de communiquer votre pattern aux autres pour qu'ils puissent les essayer, vous faire part de ce qu'ils en pensent et vous permettre de parvenir à un pattern opérationnel.

Utilisez l'un des formats existants pour définir votre pattern. Ces formats ont été soigneusement pensés et les autres utilisateurs de patterns les reconnaîtront.



Qui fait quoi ?

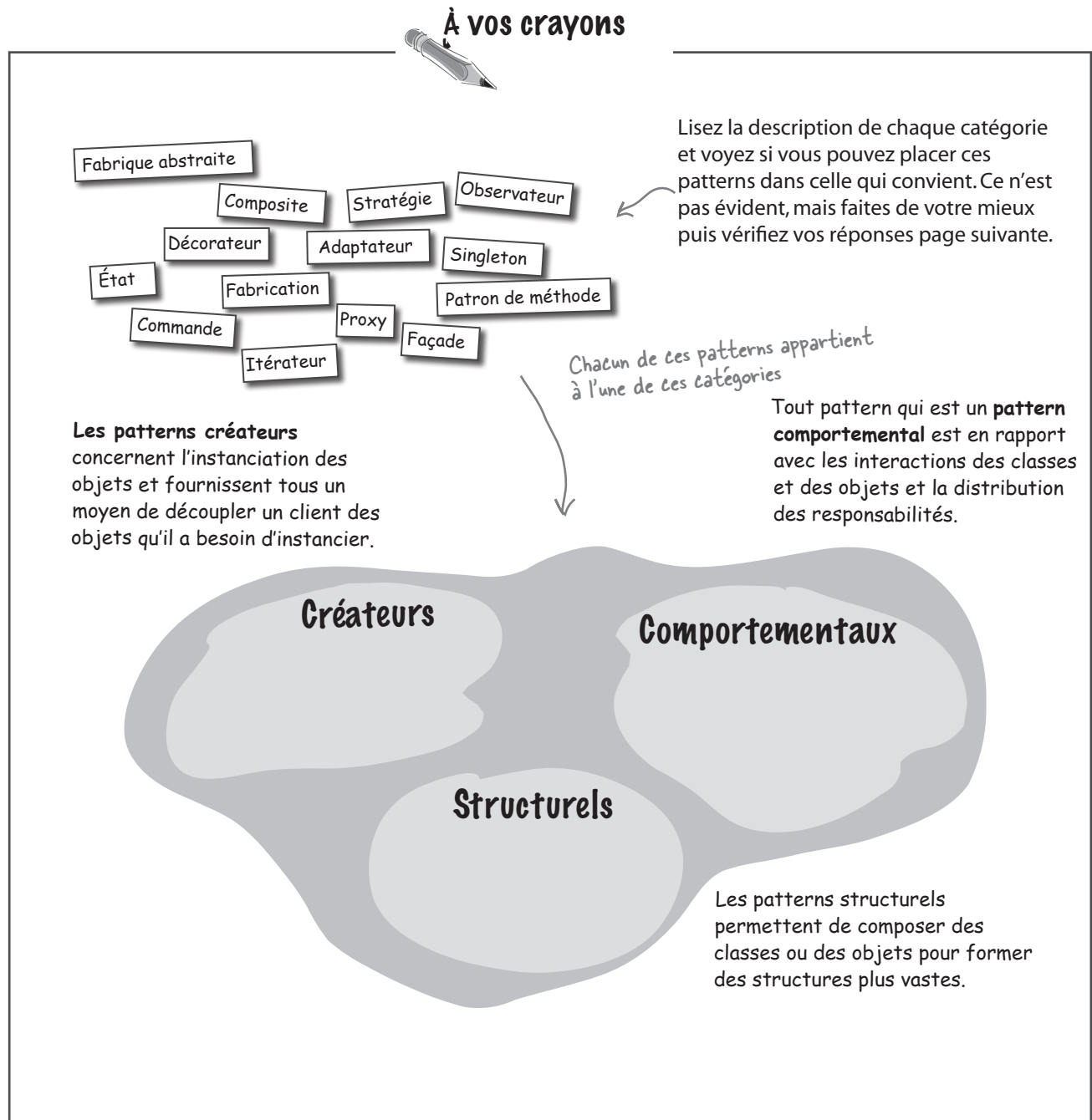
Faites correspondre chaque pattern à sa description :

Pattern	Description
Décorateur	Enveloppe un objet et fournit une interface différente pour y accéder
État	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
Itérateur	Les sous-classes décident quelles sont les classes concrètes à créer
Façade	Garantit qu'un objet et un seul est créé
Stratégie	Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
Proxy	Les clients traitent les collections d'objets et les objets individuels de la même manière
Fabrication	Encapsule des comportements basés sur des états et utilise la délégation pour permuter ces comportements
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer son implémentation
Observateur	Simplifie l'interface d'un ensemble de classes
Patron de méthode	Enveloppe un objet pour fournir un nouveau comportement
Composite	Permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes.
Singleton	Permet de notifier des changements d'état à des objets
Fabrique Abstraite	Enveloppe un objet et en contrôle l'accès
Commande	Encapsule une requête sous forme d'objet

Organiser les design patterns

À mesure que le nombre de design patterns augmente, il devient judicieux de définir des classifications pour pouvoir les organiser, restreindre nos recherches à un sous-ensemble de tous les patterns et effectuer des comparaisons au sein d'un groupe de patterns.

Dans la plupart des catalogues, vous trouverez les patterns groupés selon un ou plusieurs schémas de classification. Le plus connu de ces schémas est celui du premier catalogue de patterns et partitionne ces derniers en trois catégories distinctes en fonction de leur finalité : Créateur, Comportemental et Structurel.



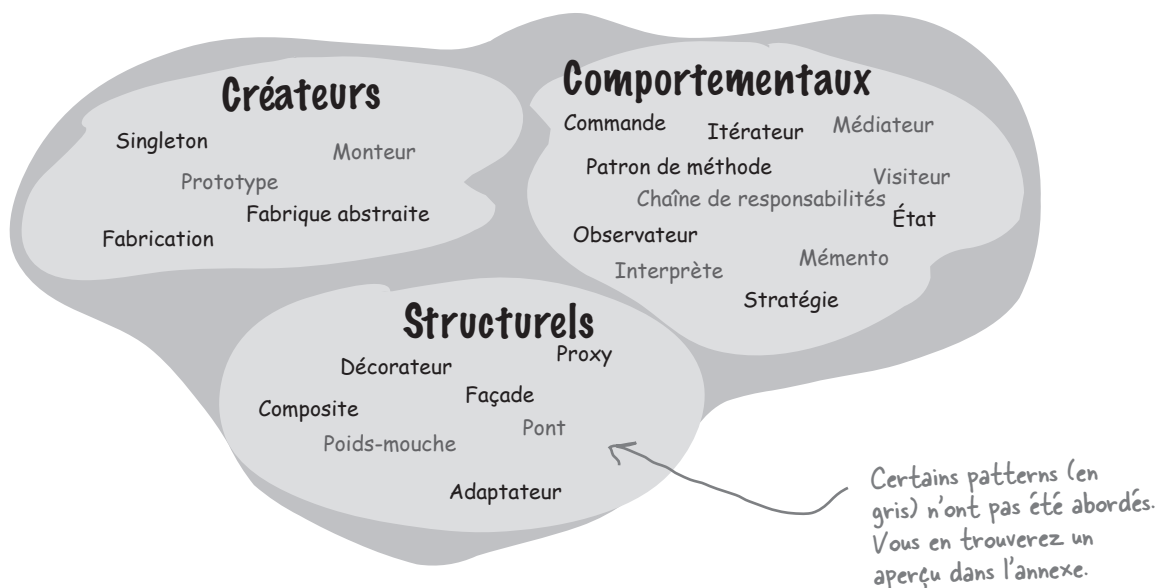
Solution : les catégories de patterns

Voici comment les patterns sont regroupés en catégories. Vous avez sans doute trouvé l'exercice difficile, parce que nombre d'entre eux ont l'air de vouloir entrer dans plusieurs catégories. Ne vous inquiétez pas : tout le monde a du mal à déterminer quelle est la bonne catégorie pour un pattern.

Les Patterns Créateurs

concernent l'instanciation des objets et fournissent tous un moyen de découpler un client des objets qu'il a besoin d'instancier.

Tout pattern qui est un **Pattern Comportemental** est en rapport avec les interactions des classes et des objets et la distribution des responsabilités.

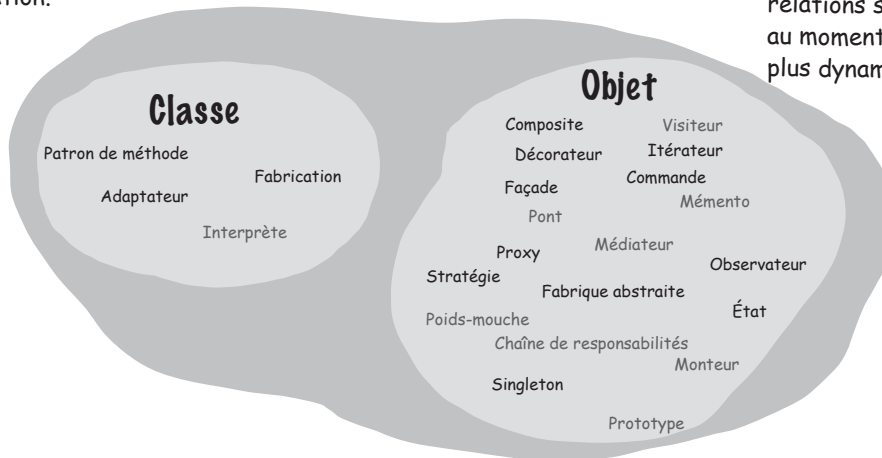


Les Patterns Structurels permettent de composer des classes ou des objets pour former des structures plus vastes.

On applique souvent un autre critère de classification : un pattern donné s'applique-t-il à des classes ou à des objets ?

Les patterns de classes définissent des relations entre classes et sont définis via l'héritage. Dans ces patterns, les relations sont établies au moment de la compilation.

Les patterns d'objets décrivent des relations entre objets et sont principalement définis par composition. Dans ces patterns, les relations sont généralement définies au moment de l'exécution et sont plus dynamiques et plus souples.



Notez que les patterns d'objets sont beaucoup plus nombreux que les patterns de classes

il n'y a pas de questions stupides

Q: Est-ce que ce sont les seuls schémas de classification ?

R: Non, d'autres schémas ont été proposés. Certains commencent par les trois catégories puis ajoutent des sous-catégories, telles que « patterns de découplage ». Vous devez connaître les schémas d'organisation les plus courants, mais n'hésitez pas à créer les vôtres si cela vous aide à mieux comprendre les patterns.

Q: Organiser les patterns en catégories aide-t-il réellement à les mémoriser ?

R: En tout cas, cela vous fournit une structure qui permet de les comparer. Mais ces catégories – créateur, structurel et comportemental – déroutent de nombreuses personnes parce que les patterns semblent souvent pouvoir appartenir à plus d'une catégorie. Le plus important est de connaître les patterns et les relations qu'ils entretiennent. Si les catégories vous aident, utilisez-les !

Q: Pourquoi le pattern Décorateur est-il dans la catégorie Structurel ? J'aurais cru que c'était un pattern comportemental : après tout, il permet d'ajouter des comportements !

R: Oui, c'est l'opinion de nombreux développeurs ! Mais voici l'idée qui sous-tend la classification de la Bande des quatre : les patterns structurels décrivent la façon dont les objets sont composés pour créer de nouvelles structures ou de nouvelles fonctionnalités. Le pattern Décorateur permet de composer des objets en enveloppant un objet dans un autre afin de fournir une nouvelle fonctionnalité. L'accent est donc mis sur la composition dynamique des objets pour obtenir une fonctionnalité plutôt que sur la communication et l'interconnexion entre objets, qui sont du ressort des patterns comportementaux. C'est une distinction subtile, surtout quand vous considérez les similitudes structurelles entre Décorateur (un pattern structurel) et Proxy (un pattern comportemental). Mais n'oubliez pas que l'intention de ces patterns est différente, et c'est souvent la clé qui permet de comprendre pourquoi un pattern appartient à telle ou telle catégorie.



Maître et disciple...

Maître : Tu as l'air troublé, Petit Scarabée.

Disciple : Oui, je viens d'étudier la classification des patterns et mes idées sont confuses.

Maître : Continue, Scarabée...

Disciple : Après avoir autant étudié les patterns, je viens d'apprendre que chacun d'eux appartenait à une catégorie : Structurel, Comportemental ou Créateur. Pourquoi avons-nous besoin de ces classifications ?

Maître : Scarabée, chaque fois que nous avons une importante collection d'entités quelconques, nous trouvons naturellement des catégories dans lesquelles les faire entrer. Cela nous permet de les penser à un niveau plus abstrait.

Disciple : Maître... Pouvez-vous me donner un exemple ?

Maître : Bien sûr. Prends les automobiles. Il existe de nombreux modèles d'automobiles différents et nous les classons naturellement dans des catégories, comme les voitures économiques, les voitures de sport, les véhicules utilitaires, les camionnettes et les voitures de luxe.

Disciple : ...

Maître : Tu as l'air choqué, Scarabée. Trouverais-tu cela illogique ?

Disciple : C'est très logique, Maître, mais votre connaissance des voitures me stupéfie !

Maître : Scarabée, je ne peux pas **tout** comparer à des fleurs de lotus ou à des bols de riz. Puis-je continuer ?

Disciple : Oui, oui, je suis désolé. Continuez, Maître, je vous en prie.

Maître : Une fois qu'on dispose de classifications ou de catégories, il est plus facile de parler des différents groupes : « Si vous vous rendez de Sophia Antipolis à Val-d'Isère par les routes touristiques, une voiture sportive avec une bonne tenue de route est la meilleure option ». Ou encore « Avec la dégradation de la situation pétrolière, mieux vaut acheter une voiture économique peu gourmande en carburant ».

Disciple : Les catégories nous permettent donc de parler d'un ensemble de patterns en tant que groupe. Nous savons peut-être qu'il nous faut un pattern comportemental sans savoir exactement lequel, mais nous pouvons toujours parler de patterns comportementaux.

Maître : Oui, et elles nous offrent également un moyen de comparer un membre au reste de la catégorie, par exemple, « la Mini est vraiment la voiture compacte la plus élégante qui soit » ou de restreindre notre recherche « J'ai besoin d'une voiture qui ne consomme pas trop ».

Disciple : Je vois... Je pourrais donc également dire que le pattern Adaptateur est le pattern structurel qui convient le mieux pour changer l'interface d'un objet.

Maître : Oui. Nous pouvons également employer des catégories pour une autre raison : pour nous lancer sur un nouveau marché. Par exemple : « Notre objectif est de proposer une voiture ayant les performances d'une Ferrari pour le prix d'une Logan ».

Disciple : Un piège mortel, somme toute.

Maître : Désolé, je n'ai pas entendu, Scarabée.

Disciple : Euh... J'ai dit « Oui, Maître, j'écoute ».

Maître : Mmmm...

Disciple : Et ainsi, les catégories nous fournissent un moyen de voir quels sont les rapports entre les groupes de patterns et comment les patterns au sein d'un même groupe sont reliés. Elles nous permettent également d'extrapoler à de nouveaux patterns. Mais pourquoi y a-t-il trois catégories, et pas quatre ou cinq ?

Maître : Ah, telles les étoiles au firmament, il y a autant de catégories que tu veux en voir. Trois est un nombre pratique et de nombreuses personnes l'ont trouvé commode pour grouper les patterns. Mais d'autres en ont suggéré quatre, ou cinq, voire plus.



Penser en termes de patterns

Contextes, contraintes, forces, catalogues, classifications... Dites donc... cela commence à sonner prodigieusement scolaire. D'accord, toutes ces notions sont importantes et savoir c'est pouvoir. Mais regardons les choses en face : si vous comprenez les aspects théoriques sans avoir l'expérience et la pratique de l'utilisation des patterns, cela ne fera pas grande différence dans votre vie.

Voici une référence rapide qui vous aidera à commencer à *penser en termes de patterns*. Qu'entendons-nous par là ? Eh bien nous entendons par là être capable d'observer une conception et de voir si les patterns cadrent naturellement avec celle-ci ou non.



Cerveau dopé aux patterns

Restez simple

En premier lieu, efforcez-vous de trouver la solution la plus naturelle possible. Vous devez viser la simplicité, non vous demander comment vous pourriez bien appliquer un pattern à un problème donné. N'allez pas imaginer que vous n'êtes pas un développeur raffiné si vous n'utilisez pas un pattern pour résoudre un problème. Vos pairs apprécieront et ils admireront la simplicité de votre conception. Cela dit, il arrive que la meilleure façon de conserver simplicité et souplesse consiste à employer un pattern.

Les design patterns ne sont pas une potion magique ; en fait, ils ne sont même pas une potion du tout !

Comme vous le savez, les patterns sont des solutions générales à des problèmes récurrents. Ils présentent également l'avantage d'avoir été testés par des myriades de développeurs. Quand vous constatez qu'il vous en faut un, vous pouvez donc dormir sur vos deux oreilles en sachant que de nombreux informaticiens vous ont précédé et ont résolu le problème en appliquant des techniques similaires.

Mais les patterns ne sont pas une potion magique. Vous ne pouvez pas en insérer un, compiler et aller à la pêche. Pour bien utiliser les patterns, vous devez également réfléchir aux conséquences qu'ils auront sur le reste de votre conception.

Vous savez que vous avez besoin d'un pattern quand...

Ah... voilà la question la plus importante : quand utiliser un pattern ? Pour commencer, n'introduisez de pattern que si vous êtes sûr qu'il résout réellement un problème dans votre conception. Si une solution plus simple est susceptible de fonctionner, envisagez-la avant de vous engager dans l'emploi d'un pattern.

Comment savoir quand un pattern s'applique ? C'est là que votre savoir et votre expérience entrent en jeu. Une fois que vous êtes certain qu'une solution simple ne suffira pas à vos besoins, considérez le problème et l'ensemble des contraintes que la solution devra respecter – cela vous aidera à apparier votre problème à un pattern. Si vous connaissez bien les patterns, vous saurez peut-être lequel correspond à vos besoins. Sinon, étudiez ceux qui semblent pouvoir résoudre le problème. Pour ce faire, les sections Intention et Indications d'utilisation du catalogue de patterns sont particulièrement utiles. Une fois que vous avez trouvé un pattern qui semble convenir, vérifiez que ses conséquences sont supportables et étudiez son effet sur le reste de votre conception. Si tout a l'air correct, allez-y !

Une seule situation impose de préférer un pattern à une solution plus simple, même si vous savez qu'elle peut fonctionner : celle où vous vous attendez à ce que certains aspects de votre système varient. Comme nous l'avons vu, l'identification de points de variation est généralement un bon indicateur de la nécessité d'un pattern. Vérifiez simplement que vous ajoutez des patterns pour traiter des changements probables qui sont susceptibles de se produire, non des changements hypothétiques qui pourraient éventuellement survenir.

L'introduction de patterns n'est pas réservée à la phase de conception : ils sont également utiles au moment de la refactorisation.

Le temps de la refactorisation est le temps des patterns !

La refactorisation est le processus qui consiste à apporter des modifications à votre code pour améliorer la façon dont il est organisé. Le but est de parfaire sa structure, non de changer son comportement. C'est le moment rêvé pour réexaminer votre conception et voir si des patterns permettraient de mieux la structurer. Par exemple, un code qui pullule d'instructions conditionnelles peut indiquer la nécessité d'appliquer le pattern État. Ou bien il peut être temps de se débarrasser de dépendances concrètes grâce à une Fabrique. Des ouvrages entiers ont été consacrés à la refactorisation, et vous devrez approfondir ce domaine à mesure que vous acquerez de l'expérience.

Éliminez ce qui n'est pas vraiment nécessaire. N'ayez pas peur de supprimer un design pattern de votre conception.

Personne ne dit jamais quand supprimer un pattern, comme si c'était un blasphème ! Nous sommes entre adultes ici, non ? Nous pouvons assumer.

Alors, quand supprimer un pattern ? Quand votre système est devenu complexe et que la souplesse que vous aviez voulu préserver n'est pas nécessaire. Autrement dit, quand une solution plus simple serait préférable au pattern.

En l'absence de nécessité immédiate, abstenez-vous.

Les design patterns sont puissants et il est aisé de voir toutes sortes d'occasions de les employer dans vos conceptions actuelles. Les développeurs ont une passion naturelle pour la création de superbes architectures capables de faire face à tous les changements possibles.

Résistez à la tentation. S'il existe aujourd'hui une nécessité pratique de prendre en charge un changement, allez-y : utilisez un pattern pour le gérer. Mais si le changement est seulement hypothétique, renoncez au pattern : il ne ferait qu'ajouter de la complexité à votre système et vous n'en aurez peut-être même jamais besoin !

Concentrez-vous sur la conception, pas sur les patterns. Employez des patterns quand ils correspondent à un besoin naturel. Si une solution plus simple peut fonctionner, adoptez-la.





Maître et Disciple...

Maître : Ta formation initiale est presque terminée, Petit scarabée. Quels sont tes projets ?

Disciple : D'abord, je vais aller à Disneyland ! Et puis je vais écrire des tas de programmes avec des patterns !

Maître : Hé, attends une minute. Inutile de sortir l'artillerie lourde si tu n'en as pas besoin.

Disciple : Que voulez vous dire, Maître ? Maintenant que j'ai appris les design patterns, est-ce que je ne dois pas les utiliser pour obtenir le maximum de puissance, de souplesse et de maintenabilité ?

Maître : Non. Les patterns sont un outil, et on ne doit utiliser un outil que lorsqu'on en a besoin. Tu as aussi passé beaucoup de temps à apprendre les principes de conception. Pars toujours de ces principes, et écris toujours le code le plus simple possible pourvu qu'il remplisse sa fonction. Mais si la nécessité d'un pattern se fait jour, emploie-le.

Disciple : Alors, je ne dois pas construire mes conceptions sur des patterns ?

Maître : Cela ne doit pas être ton objectif quand tu entames une conception. Laisse les patterns émerger naturellement à mesure que celle-ci progresse.

Disciple : Si les patterns sont tellement géniaux, pourquoi tant de circonspection ?

Maître : Les patterns peuvent introduire de la complexité et nous préférons toujours éviter la complexité lorsqu'elle est inutile. Mais les patterns sont puissants quand on les utilise à bon escient. Comme tu le sais, les patterns sont des solutions éprouvées issues de l'expérience et l'on peut les utiliser pour éviter des erreurs courantes. Ils constituent également un vocabulaire partagé qui nous permet de communiquer notre conception à d'autres développeurs.

Disciple : Mais comment sait-on qu'il est judicieux d'introduire un pattern ?

Maître : N'introduis un pattern que lorsque tu es sûr qu'il est nécessaire pour résoudre un problème dans ta conception ou pour faire face à un changement futur dans les exigences de ton application.

Disciple : Je crois que mon apprentissage va se poursuivre, même si je comprends déjà beaucoup de patterns.

Maître : Oui, Scarabée. Apprendre à gérer la complexité et le changement dans le domaine logiciel est une poursuite sans fin. Mais maintenant que tu connais un bel assortiment de patterns, le temps est venu de les appliquer et de continuer à en apprendre d'autres.

Disciple : Attendez une minute, vous voulez dire que je ne les connais pas TOUS ?

Maître : Scarabée... Tu as appris les patterns fondamentaux, mais tu vas découvrir qu'il en existe beaucoup plus, notamment des patterns qui s'appliquent uniquement à des domaines spécialisés tels que les systèmes concurrents et les systèmes d'entreprise. Mais maintenant que tu connais les bases, tu es prêt à les apprendre !

Votre esprit et les patterns



ESPRIT NEUF

Le débutant voit des patterns partout. C'est bien : il pratique les patterns et accumule de l'expérience. Le débutant pense également : « Plus ma conception contiendra de patterns, meilleure elle sera ». Il apprendra bientôt qu'il n'en est pas ainsi et que toute conception doit être aussi simple que possible. La complexité et les patterns ne sont utiles que lorsqu'il existe un réel problème d'extensibilité.

<< J'ai besoin d'un pattern pour Hello World. >>

À mesure que son apprentissage progresse, le développeur commence à voir quand un pattern est nécessaire ou non. Il continue à essayer de faire entrer trop de patterns carrés dans des trous ronds, mais il commence également à se rendre compte que l'on peut adapter les patterns pour gérer des situations dans lesquelles les patterns canoniques ne conviennent pas.



STADE INTERMÉDIAIRE

<< Peut-être qu'un Singleton conviendrait. >>



ESPRIT ZEN

L'esprit Zen sait quand un pattern s'applique naturellement. Il n'est pas obsédé par l'emploi des patterns mais recherche des solutions simples qui résolvent au mieux le problème. Il pense en termes de principes objets et pèse leurs avantages et leurs inconvénients. Quand un pattern semble s'imposer naturellement, l'esprit Zen l'applique en sachant qu'il devra peut-être l'adapter. Il voit également les relations qu'il entretient avec des patterns similaires et comprend les subtilités des différences d'intention entre patterns apparentés. *L'esprit Zen est aussi un esprit neuf* – il ne laisse pas tout ce savoir sur les patterns exercer une influence exagérée sur ses décisions de conception.

<< Voilà une place naturelle pour Décorateur. >>

ATTENTION : L'abus de design patterns peut aboutir à un code exagérément élaboré. Recherchez toujours la solution la plus simple et n'introduisez de patterns que lorsque le besoin s'en fait ressentir.

Attendez une minute !
J'ai lu tout ce livre sur les
patterns et maintenant vous
me dites de ne PAS les
utiliser ?



Bien sûr que nous voulons que vous utilisiez des design patterns !

Mais nous voulons encore plus que vous soyez un bon concepteur OO.

Lorsqu'un problème de conception demande un pattern, vous tirez parti d'une solution qui a été éprouvée par quantité de développeurs. De plus, cette solution est bien documentée et sera reconnue par vos pairs (vous savez toute cette histoire de vocabulaire partagé).

Toutefois, l'application de design patterns peut présenter un inconvénient. Les patterns introduisent souvent des classes et des objets supplémentaires, ce qui peut rendre vos conceptions plus complexes. Ils peuvent également obliger à créer des couches additionnelles, ce qui n'augmente pas la complexité mais entraîne des problèmes d'efficacité.

De plus, l'emploi d'un design pattern peut parfois relever du surarmement. Vous pouvez souvent revenir à vos principes de conception et trouver une solution plus simple au même problème. Si cela se produit, ne résistez pas : choisissez la solution simple.

Mais nous ne voulons pas vous décourager. Si un design pattern s'avère être le bon outil, les avantages seront légion.

N'oubliez pas le pouvoir du vocabulaire partagé

Nous avons consacré tellement de temps à évoquer les détails techniques de la conception objet qu'il serait facile d'oublier l'aspect humain des design patterns – non contents de vous fournir des solutions, ils mettent à votre disposition un vocabulaire que les autres développeurs partagent. Ne sous-estimez pas l'importance de ce vocabulaire partagé : c'est l'un des *principaux avantages* des design patterns.

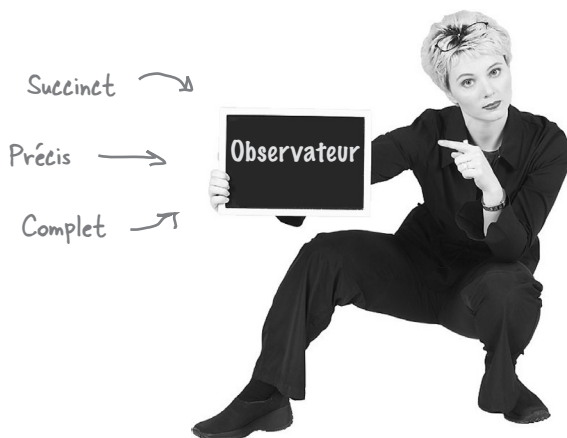
Réfléchissons : quelque chose a changé depuis la dernière fois que nous avons parlé de vocabulaires partagés et vous avez commencé à construire tout un vocabulaire qui vous est propre ! Sans mentionner le fait que vous avez également appris tout un ensemble de principes de conception OO qui vous permettent de comprendre la motivation et le fonctionnement de tout nouveau pattern que vous rencontrerez.

Maintenant que vous avez assimilé les bases des design patterns, il est temps pour vous de faire passer le mot. Pourquoi ? Parce que si vos collègues développeurs connaissent les patterns et emploient ce vocabulaire partagé, les conceptions s'améliorent, la communication aussi, et, avantage non négligeable, vous disposez de plus de temps à consacrer à des activités plus intéressantes.



Les cinq meilleures façons de partager votre vocabulaire

- 1 Dans les réunions de conception :** Quand vous vous réunissez avec votre équipe pour étudier une conception logicielle, utilisez les design patterns pour pouvoir rester « dans la conception » plus longtemps. Analyser une conception du point de vue des design patterns et des principes OO empêche votre équipe de s'enliser dans les détails de l'implémentation et évite de nombreux malentendus.
- 2 Avec les autres développeurs :** Utilisez les patterns dans vos discussions avec les autres développeurs. Cela les aide à apprendre de nouveaux patterns et construire une communauté. Quand vous partagez ce que vous avez appris, la meilleure partie est cette super sensation que quelqu'un d'autre « a compris » !
- 3 Dans la documentation de l'architecture :** Quand vous rédigez une documentation d'architecture, l'emploi de patterns réduit le volume de celle-ci et permet au lecteur de se faire une image plus claire de la conception.
- 4 Dans les commentaires du code et les conventions de nommage :** Quand vous écrivez du code, identifiez clairement les patterns que vous utilisez dans des commentaires. Choisissez également des noms de classe et de méthode qui évoquent les patterns sous-jacents. Les autres développeurs qui doivent lire votre code vous remercieront de leur permettre de comprendre rapidement votre implémentation.
- 5 Dans les groupes de développeurs intéressés :** Partagez votre savoir. Beaucoup de développeurs ont entendu parler des patterns, mais ne comprennent pas bien de quoi il s'agit. Organisez un « déjeuner patterns » ou donnez une conférence à votre groupe de patterns local.



Tour d'Objectville avec la Bande des quatre

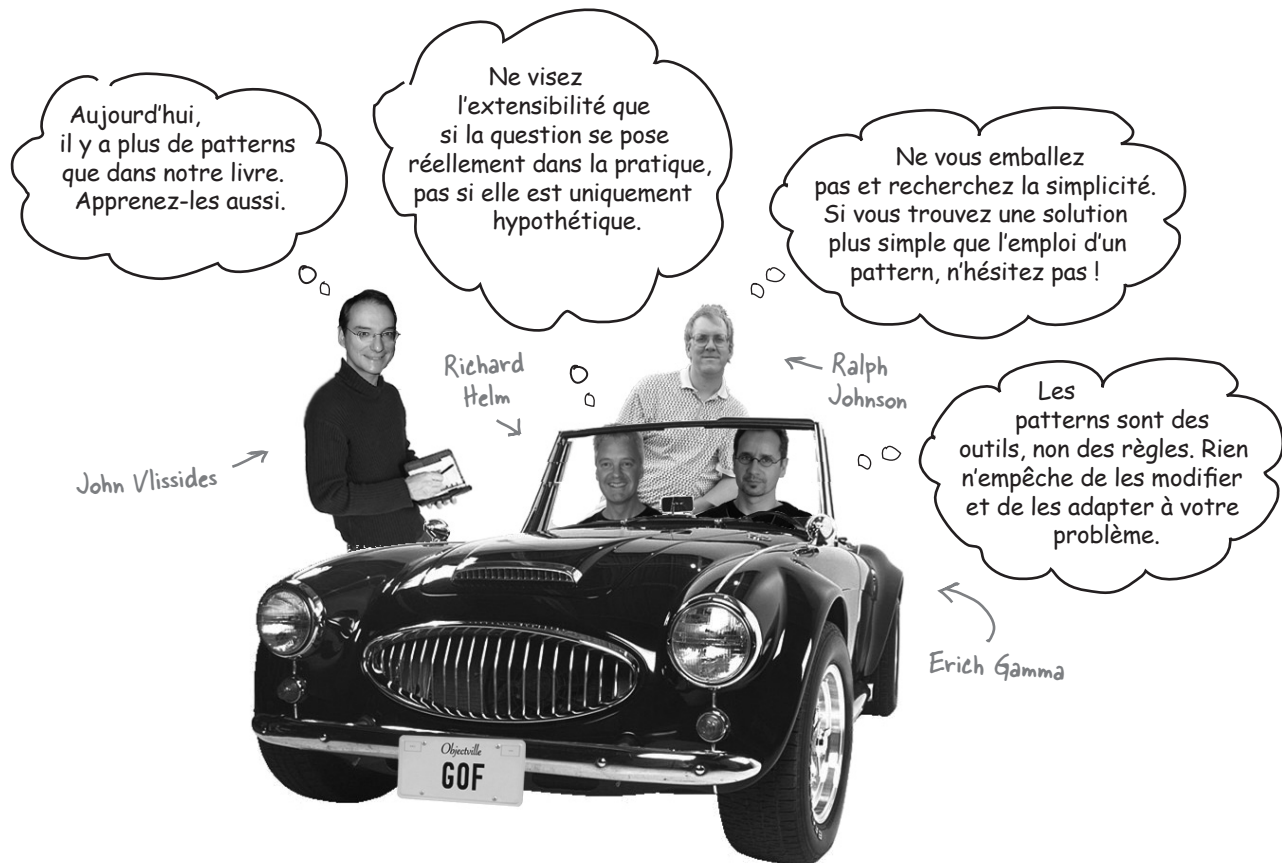
Vous ne trouverez ni Jets ni Sharks traînant dans les rues d'Objectville, mais vous y trouverez la Bande des quatre. Vous l'avez sans doute constaté, on ne va pas loin dans le Monde des patterns sans tomber sur eux.

Mais quelle est donc cette mystérieuse bande ?

Il s'agit d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, le groupe qui a constitué le premier catalogue de patterns, et, ce faisant, lancé tout un mouvement dans le domaine du développement logiciel !

D'où ce nom leur vient-il ? Personne ne le sait exactement, c'est juste un nom qui leur est resté. Mais réfléchissez : si nous devons avoir une bande qui zone dans Objectville, pouvez-vous imaginer meilleure compagnie ? En fait, ils se sont même rendus à notre invitation...

La Bande des quatre a lancé le mouvement des patterns logiciels, mais de nombreuses autres personnes ont apporté une contribution significative, notamment Ward Cunningham, Kent Beck, Yann Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad et Doug Schmidt, pour n'en citer que quelques-unes.



Votre voyage ne fait que commencer...

Maintenant que vous vous débrouillez avec les Design Patterns et que vous êtes prêt à aller plus loin, voici trois ouvrages indispensables que votre bibliothèque se doit de contenir...

Le livre décisif sur les Design Patterns

C'est le livre qui a donné le coup d'envoi à tout le mouvement des *Design Patterns* lors de sa sortie en 1995. Vous y trouverez tous les patterns fondamentaux. En fait, c'est la base de l'ensemble de patterns que nous avons décrit dans *Design patterns* tête la première.

Mais ce n'est pas le mot de la fin sur les *Design Patterns*, car ce domaine s'est substantiellement développé depuis sa publication. Mais c'est le premier et c'est celui qui fait le plus autorité. Se procurer un exemplaire de *Design Patterns* est une excellente façon de commencer à les explorer après Tête la première.



Les auteurs de *Design Patterns* ont reçu le surnom affectueux de « Bande des quatre » (« Gang of Four » en anglais, ou GoF pour abréger, d'où l'expression de patterns GoF)...

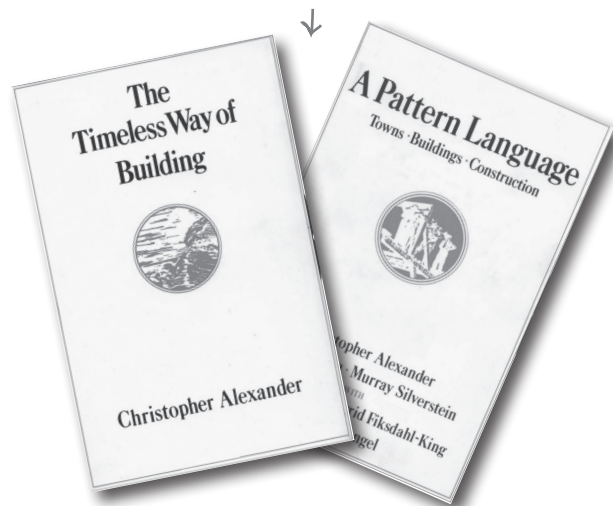
Les textes de base sur les Patterns en général

Les patterns n'ont pas commencé avec la Bande des quatre, mais avec Christopher Alexander, professeur d'architecture à l'université de Berkeley. Oui, Alexander est un *architecte*, non un informaticien, et il a inventé les patterns pour construire des architectures vivantes (telles des maisons, des villes et des cités).

La prochaine fois que vous êtes d'humeur à vous plonger dans une lecture ardue, mais captivante, procurez-vous *The Timeless Way of Building* et *A Pattern Language*. Vous assisterez aux vrais débuts des Design Patterns et vous reconnaîtrez les analogies directes entre la création d'une « architecture vivante » et celle de logiciels souples et extensibles.

Alors allez vous chercher une tasse de café Starbuzz, installez-vous confortablement et appréciez...

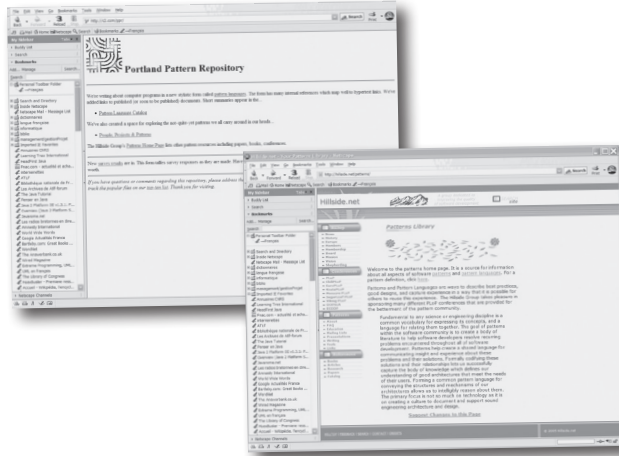
C'est Christopher Alexander qui a inventé la notion de patterns, qui a inspiré l'application de solutions similaires au développement logiciel.



Autres ressources sur les design patterns

Vous trouverez là une communauté d'utilisateurs et d'auteurs de patterns conviviale et passionnée, qui sera heureuse de vous voir la rejoindre.

Voici quelques ressources pour vous aider à démarrer..



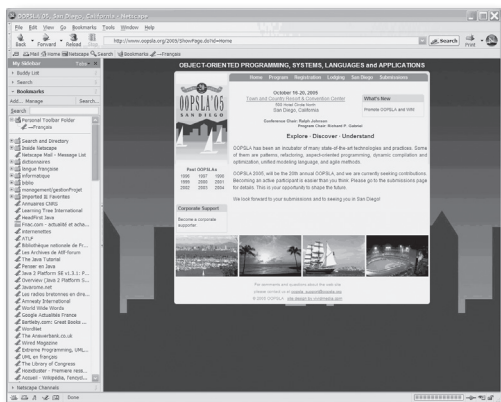
Sites web

The Portland Patterns Repository, maintenu par Ward Cunningham, est un WIKI dédié à tout ce qui est lié aux patterns. Tout le monde peut y participer. Vous y trouverez des fils de discussion sur tous les sujets imaginables en rapport avec les patterns et les systèmes OO.

<http://c2.com/ppr/>

Le **Hillside Group** encourage les pratiques de programmation et de conception communes et centralise des ressources sur les patterns. Le site contient de nombreuses informations sur ces ressources, notamment les livres, les articles, les listes de diffusion et les outils.

<http://hillside.net/patterns>



Conférences et ateliers

Et si vous voulez rencontrer en direct la communauté des utilisateurs de patterns, il existe quantité de conférences et d'ateliers. Le site de Hillside en maintient une liste complète. Consultez au moins la page de l'OOPSLA, la conférence de l'ACM sur les systèmes, les langages et les applications orientés objet.

Le Zoo des patterns

Nous venons de le voir, les patterns n'ont pas vu le jour dans le contexte du développement logiciel, mais dans celui de l'architecture des bâtiments et des villes. En réalité, le concept de pattern peut s'appliquer à de nombreux domaines très différents. Allons faire un tour au Zoo des patterns pour observer quelques spécimens...



Les Patterns d'architecture servent à créer l'architecture vivante des bâtiments, des villes et des cités. C'est de là que les patterns sont originaires.

Habitat : rencontré dans les constructions que vous aimez visiter, regarder et habiter.

Habitat : vue dans le voisinage des architectures à trois couches, des systèmes client-serveur et des applications web.

Les Patterns d'application permettent de créer des architectures de système. Nombre d'architecture en couches entrent dans cette catégorie...



Note de terrain : MVC est connu pour passer pour un pattern d'application.



Les Patterns de domaine concernent des problèmes spécifiques à un domaine, comme les systèmes concurrents ou les systèmes temps-réel.

Aidez à leur trouver un habitat

J2EE

Les Patterns de processus métier décrivent les interactions entre les entreprises, les clients et les données, et sont applicables à des problèmes tels que « comment prendre et communiquer efficacement des décisions ».



Observé aux environs des réunions de conseils d'administration de gestion de projets.

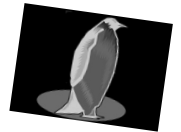
Aidez à leur trouver un habitat

Équipes de développement

Équipes de services clients

Les Patterns d'organisation décrivent les structures et les pratiques des organisations humaines.

Jusqu'ici, les efforts se sont concentrés sur les organisations qui produisent du logiciel ou du support.



Les Patterns de conception d'interfaces utilisateur traitent des problèmes de la conception de logiciels interactifs.



Habitat : rencontré au voisinage des concepteurs de jeux vidéo et d'IHM diverses.

Notes de terrain : ajoutez vos propres observations ici :

Annihiler le mal avec les Anti-Patterns



L'Univers serait tout bonnement incomplet si nous avions des patterns et pas d'anti-patterns, n'est-ce pas ?

Si un Design Pattern vous fournit une solution générale à un problème récurrent dans un contexte donné, que vous donnera un anti-pattern ?

Un **Anti-Pattern** vous dit comment partir d'un problème et parvenir à une MAUVAISE solution.

Vous vous demandez probablement : « Pourquoi diable quelqu'un voudrait-il gaspiller son temps à documenter de mauvaises solutions ? ».

Vous pouvez voir les choses ainsi : s'il existe une mauvaise solution récurrente à un problème courant, le fait de la documenter peut empêcher d'autres développeurs de commettre la même erreur. Après tout, éviter de mauvaises solutions peut-être tout aussi productif que d'en trouver de bonnes !

Voyons quels sont les éléments d'un anti-pattern :

Un anti-pattern vous dit pourquoi une mauvaise solution est attrayante. Regardons les choses en face, personne ne choisirait une mauvaise solution si elle ne présentait pas d'abord un aspect séduisant. L'une des fonctions les plus importantes de l'anti-pattern consiste à vous mettre en garde contre cet aspect séduisant.

Un anti-pattern vous dit pourquoi cette solution est mauvaise à long terme. Pour savoir pourquoi c'est un anti-pattern, vous devez d'abord comprendre en quoi il aura un effet négatif en aval. L'anti-pattern décrit les ennuis que vous allez vous attirer en adoptant cette solution.

Un anti-pattern suggère d'autres patterns applicables pouvant fournir de meilleures solutions. Pour être vraiment utile, un anti-pattern doit vous montrer la bonne direction. Il doit indiquer d'autres possibilités qui peuvent permettre de déboucher sur de bonnes solutions.

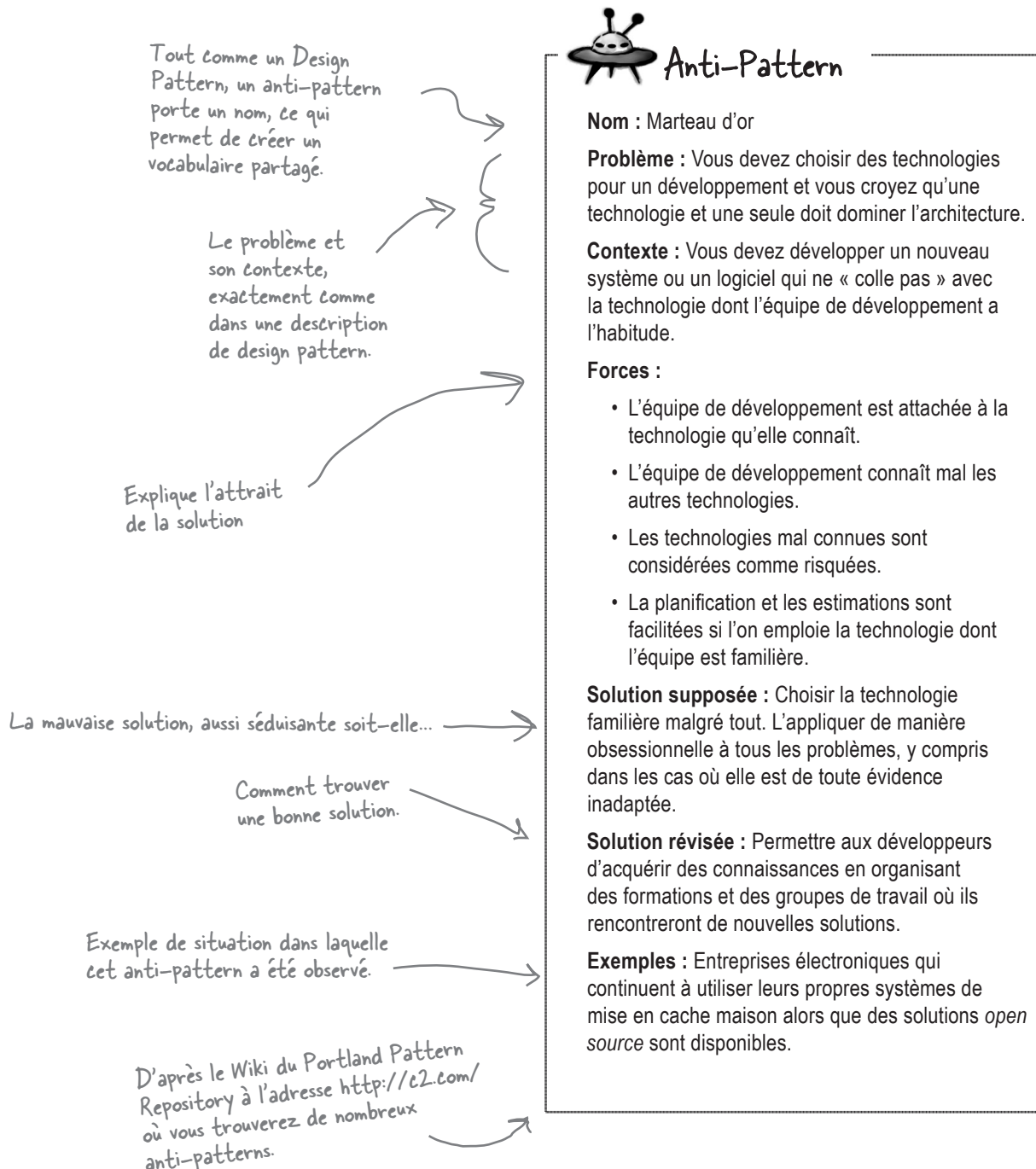
Jetons un coup d'œil sur un anti-pattern.

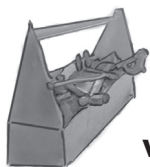
Un anti-pattern ressemble toujours à une bonne solution, ladite solution s'avérant mauvaise lorsqu'elle est appliquée.

En documentant les anti-patterns, nous aidons les autres développeurs à reconnaître les mauvaises solutions avant de commencer à les mettre en œuvre.

À l'instar des patterns, il existe de nombreux types d'anti-patterns : anti-patterns de développement, OO, organisationnels, spécifiques à un domaine, etc.

Voici un exemple d'anti-pattern de développement logiciel.





Votre boîte à outils de concepteur

Vos connaissances font qu'il est désormais temps de parcourir le monde et d'explorer les patterns par vous-même...



Le moment est venu pour vous de découvrir d'autres patterns par vous-même. Il y aura de nombreux patterns spécifiques à un domaine que nous n'avons pas mentionnés et des patterns fondamentaux que nous n'avons pas abordés.

Vous avez également vos propres patterns à créer.

Consultez l'annexe, Vous y trouverez un aperçu d'autres patterns fondamentaux qui vous intéresseront certainement.



POINTS D'IMPACT

- Laissez les Design Patterns émerger dans vos conceptions, ne les forcez pas pour le simple plaisir d'utiliser un pattern.
- Les Design Patterns ne sont pas gravés dans la pierre : adaptez-les selon vos besoins.
- Choisissez toujours la solution la plus simple qui correspond à vos besoins, même si elle ne fait appel à aucun pattern.
- Étudiez les catalogues de Design Patterns pour vous familiariser avec les patterns et les relations qui existent entre eux.
- Les classifications (ou catégories) permettent de grouper les patterns. Quand elles sont utiles, utilisez-les.
- Vous devez vous impliquer pour être un auteur de patterns : cela demande du temps et de la patience, et beaucoup de minutie.
- Souvenez-vous que la plupart des patterns que vous rencontrez seront des adaptations de patterns existants, non de nouveaux patterns.
- Construisez un vocabulaire commun avec votre équipe. C'est l'un des principaux avantages de l'utilisation des patterns.
- Comme toute communauté, celle des patterns possède son propre jargon. Que cela ne vous rebute pas. Ayant lu cet ouvrage, vous en connaissez maintenant la plus grande partie.

Quitter Objectville...



Ce fut un plaisir de vous accueillir à Objectville.

Vous allez nous manquer, c'est certain. Mais ne vous inquiétez pas – avant d'avoir eu le temps de dire « ouf » le prochain *Tête la première* sera sorti et vous pourrez revenir nous voir. Quel sera le prochain livre, dites-vous ? Mmmm, bonne question ! Pourquoi ne pas nous aider à décider ? Envoyez un message (en anglais) à booksuggestions@wickedlysmart.com.

Qui fait quoi ?

Faites correspondre chaque pattern à sa description :

Pattern	Description
Décorateur	Enveloppe un objet et fournit une interface différente pour y accéder
État	Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
Itérateur	Les sous-classes décident quelles sont les classes concrètes à créer
Façade	Garantit qu'un objet et un seul est créé
Stratégie	Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
Proxy	Les clients traitent les collections d'objets et les objets individuels de la même manière
Fabrication	Encapsule des comportements basés sur des états et utilise la délégation pour permuter ces comportements
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer son implémentation
Observateur	Simplifie l'interface d'un ensemble de classes
Patron de méthode	Enveloppe un objet pour fournir un nouveau comportement
Composite	Permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes.
Singleton	Permet de notifier des changements d'état à des objets
Fabrique Abstraite	Enveloppe un objet et en contrôle l'accès
Commande	Encapsule une requête sous forme d'objet

Annexe : Les patterns restants



La popularité n'est pas donnée à tout le monde. Beaucoup de choses ont changé ces dix dernières années. Depuis la parution de *Design Patterns : Catalogue de modèles de conception réutilisables*, les développeurs ont appliqué ces patterns des milliers de fois. Ceux que nous avons résumés dans cette annexe sont des patterns GoF à part entière. Bien qu'ils soient tout aussi officiels que ceux que nous avons étudiés jusqu'à maintenant, leur emploi est moins fréquent. Mais ils sont tout aussi remarquables, et, si la situation le demande, vous pouvez les utiliser la tête haute. Notre objectif est de vous donner une vue d'ensemble de ces patterns.

Pont

Utilisez le pattern Pont pour faire varier non seulement vos implémentations, mais aussi vos abstractions.

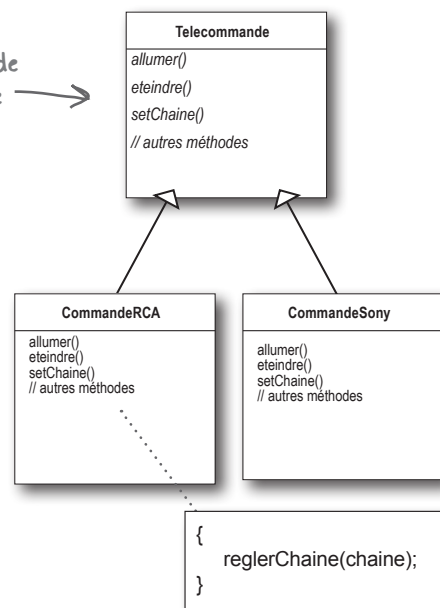
Un scénario

Imaginez que vous allez révolutionner l'« extrême cocooning ». Vous êtes en train d'écrire le code d'une nouvelle télécommande de téléviseur ergonomique et conviviale. Vous savez déjà que vous allez devoir appliquer de bonnes techniques OO, car si la commande est basée sur la même *abstraction*, il y aura de nombreuses *implémentations* – une pour chaque modèle de poste.

Voici l'abstraction. Ce pourrait être une interface ou une classe abstraite.

Chaque télécommande s'appuie sur la même abstraction.

De nombreuses implémentations, une par modèle de télécommande.



Votre dilemme

Vous savez que l'interface utilisateur de la télécommande ne sera pas parfaite du premier coup. En fait, vous vous attendez à devoir affiner le produit plusieurs fois, à mesure que vous recueillerez des données sur sa facilité d'utilisation.

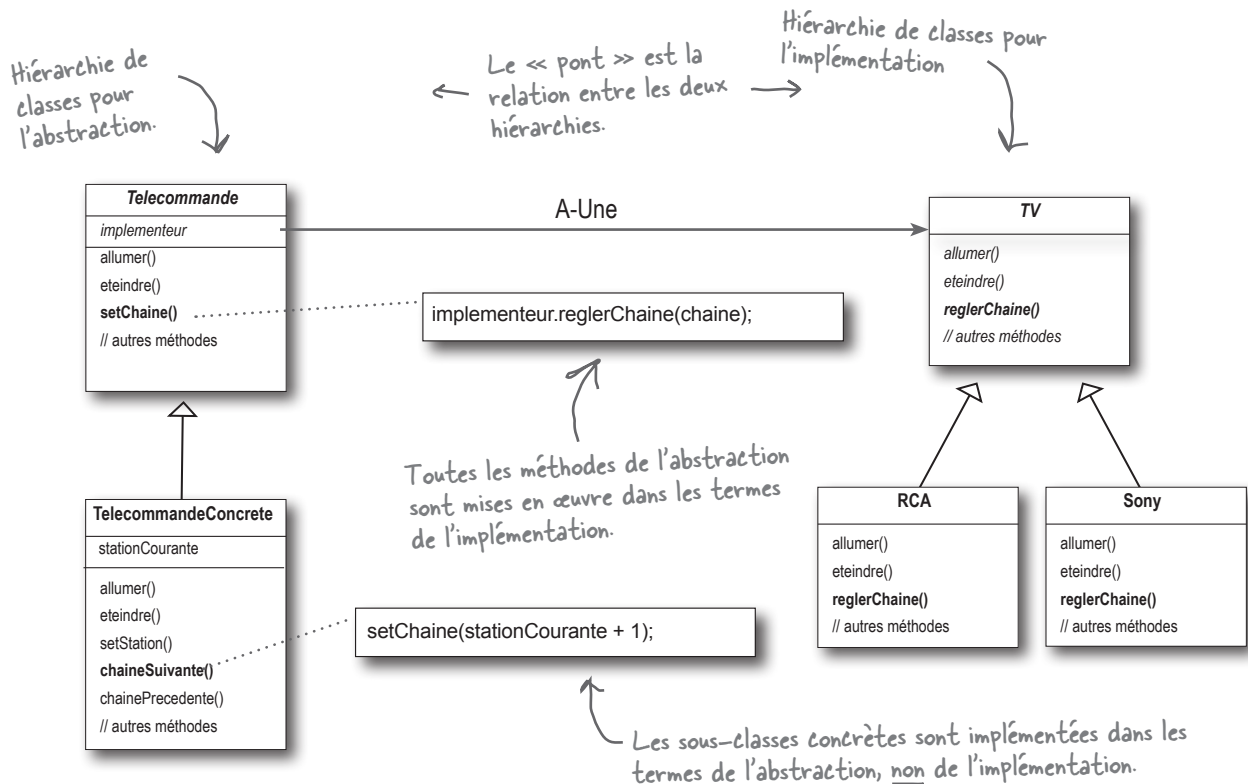
Votre dilemme réside donc dans le fait que les télécommandes vont changer et les téléviseurs aussi. Vous avez déjà *abstrait* l'interface utilisateur pour pouvoir faire varier l'*implémentation* en fonction des différents postes que vos utilisateurs posséderont. Mais vous allez également devoir *faire varier l'abstraction* parce qu'elle va changer au fur et à mesure que vous allez améliorer la télécommande en tenant compte du feedback des utilisateurs.

Comment donc allez-vous créer une conception OO qui vous permette de faire varier l'implémentation *et* l'abstraction ?

Cette conception ne nous permet de faire varier que les implémentations, pas l'interface utilisateur.

Pourquoi utiliser le pattern Pont ?

Le pattern Pont vous permet de faire varier l'implémentation *et* l'abstraction en plaçant les deux dans des hiérarchies de classes séparées..



Vous disposez maintenant de deux hiérarchies séparées : l'une pour les télécommandes et l'autre pour les implémentations de téléviseurs spécifiques à une plate-forme. Le pont vous permet de faire varier indépendamment chaque côté des deux hiérarchies.

Avantages

- Découple une implémentation afin qu'elle ne soit pas liée de façon permanente à une interface.
- L'abstraction et l'implémentation peuvent être modifiées indépendamment.
- Les modifications des classes concrètes de l'abstraction n'affectent pas le client.

Emplois et inconvénients

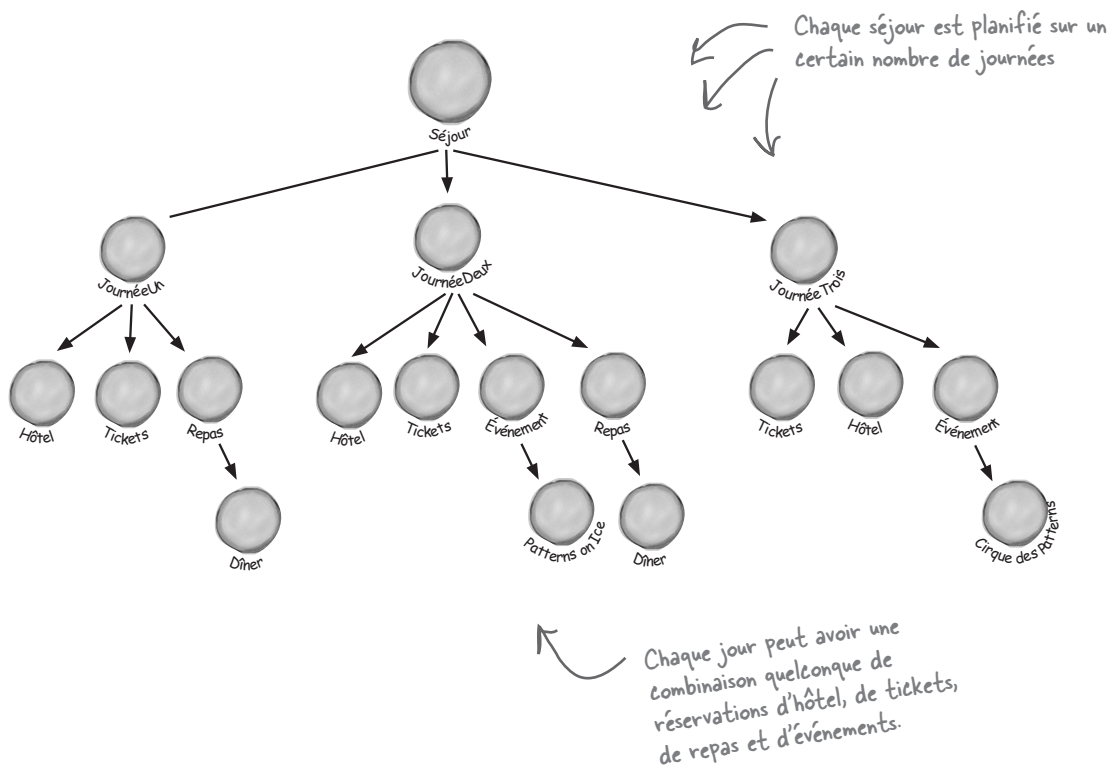
- Utile dans les systèmes graphiques et à base de fenêtres qui doivent fonctionner sur différentes plates-formes.
- Utile chaque fois que vous devez faire varier une interface et une implémentation de différentes manières.
- Augmente la complexité.

Monteur

Utilisez le pattern Monteur pour encapsuler la construction d'un produit et permettre de le construire par étapes.

Un scénario

On vient de vous confier le développement d'un logiciel de planification de séjours pour *Patternsland*, un nouveau parc à thème récemment construit aux portes d'Objectville. Les hôtes du parc peuvent choisir un hôtel et différents types de tickets d'admission, réserver des tables au restaurant et même des places à des événements spéciaux. Pour écrire ce programme, vous devez pouvoir créer des structures telles que celles-ci :



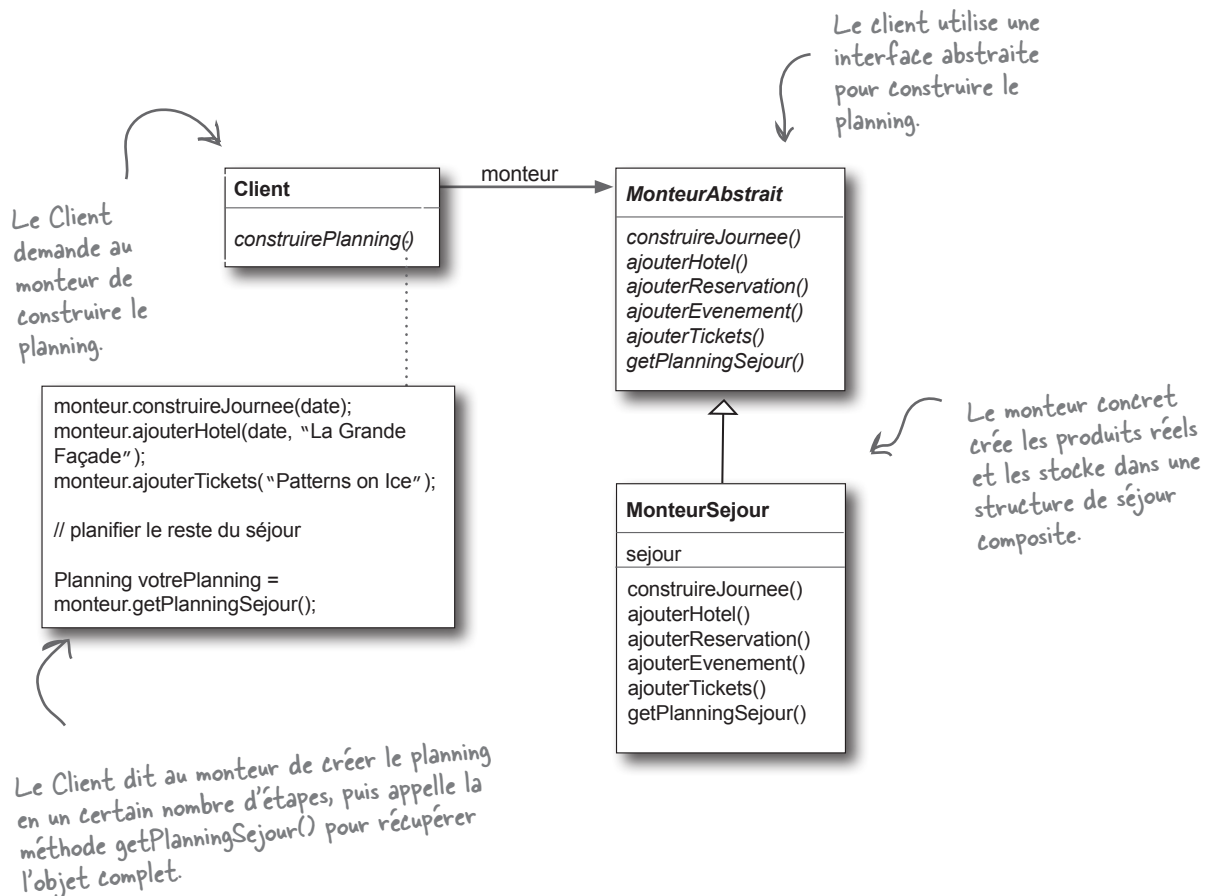
Il vous faut une conception souple

Le planning de chaque hôte peut comprendre un nombre de journées et des types d'activités variables. Par exemple, un résident local n'aura pas besoin d'hôtel mais pourra désirer réserver pour dîner et assister à des événements spéciaux. Un autre arrivera en avion à Objectville et aura besoin de réserver un hôtel, des repas et des tickets d'entrée.

Il vous faut donc une structure de données souple et capable de représenter les plannings des hôtes avec toutes leurs variantes. Vous devez également observer une séquence d'étapes potentiellement complexes pour créer le planning. Quel moyen avez-vous de créer cette structure complexe sans la mélanger avec les étapes nécessaires pour la créer ?

Pourquoi utiliser le pattern Monteur ?

Vous souvenez-vous d'Itérateur ? Nous avons encapsulé l'itération dans un objet distinct et caché la représentation interne de la collection au client. L'idée est la même ici : nous encapsulons la création du planning dans un objet (appelons-le un monteur) et notre client demandera au monteur de construire la structure du planning à sa place.



Avantages

- Encapsule la façon dont un objet complexe est construit.
- Permet de construire des objets en plusieurs étapes selon un processus variable (à la différence des Fabriques).
- Masque la représentation interne du produit au client.
- Les implémentations des produits sont interchangeables parce que le client ne voit qu'une interface abstraite.

Emplois et inconvénients

- Souvent employé pour créer des structures composites.
- La construction des objets demande plus de connaissance du domaine du client que l'emploi d'une Fabrique.

Chaîne de responsabilité

Utilisez le pattern Chaîne de responsabilité quand vous voulez donner à plus d'un objet une chance de traiter une requête.

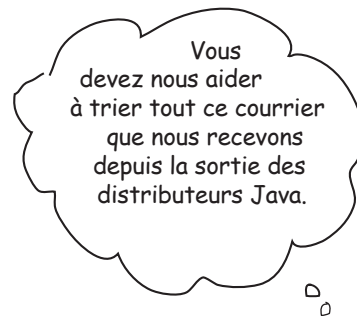
Un scénario

Distribon reçoit plus de courrier électronique qu'il ne peut en gérer depuis la sortie de leurs distributeurs de bonbons équipés de Java. Selon leur propre analyse, ils reçoivent quatre types de courrier : des lettres de fans qui adorent leur nouveau jeu, des réclamations de parents dont les gosses sont accros au jeu et des demandes de mise en place de nouveaux distributeurs. Ils reçoivent également pas mal de spams.

Tout le courrier des admirateurs est transmis directement au P.D.G., les plaintes au service juridique et les demandes de nouveaux distributeurs au service développement. Les messages indésirables doivent être détruits.

Votre tâche

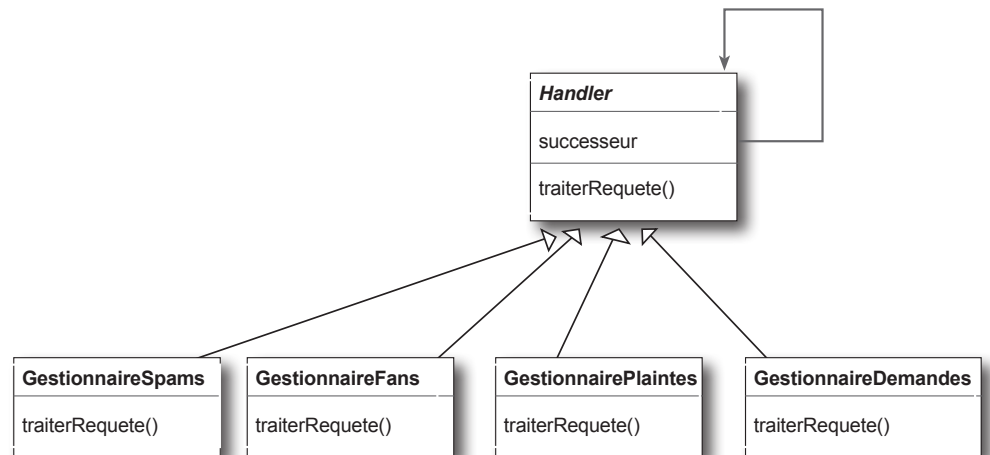
Distribon a déjà écrit des détecteurs intelligents qui peuvent distinguer une lettre de fan d'un spam, d'une réclamation ou d'une demande, mais ils ont besoin de vous pour créer une conception qui permette aux détecteurs de gérer le courrier entrant.



Comment utiliser le pattern Chaîne de responsabilité ?

Avec le pattern Chaîne de responsabilité, vous créez une chaîne d'objets qui examinent une requête. Chaque objet considère la requête à son tour et la traite ou la transmet à l'objet suivant dans la chaîne.

Chaque objet de la chaîne joue le rôle de gestionnaire et a un objet successeur. S'il peut traiter la requête, il le fait, sinon il la transmet à son successeur.



Quand un message est reçu, il est transmis au premier gestionnaire : GestionnaireSpams. Si GestionnaireSpams ne peut pas traiter la requête, il la transmet à GestionnaireFans et ainsi de suite...

Chaque message est passé au premier gestionnaire



Le message n'est pas traité s'il « tombe » de la fin de la chaîne – mais vous pouvez toujours implémenter un autre gestionnaire qui intercepte tout.

Avantages

- Découple l'émetteur de la requête de ses récepteurs.
- Simplifie votre objet, car il n'a pas besoin de connaître la structure de la chaîne ni de conserver des références directes à ses membres.
- Permet d'ajouter ou de supprimer dynamiquement des responsabilités en changeant les membres ou l'ordre de la chaîne.

Emplois et inconvénients

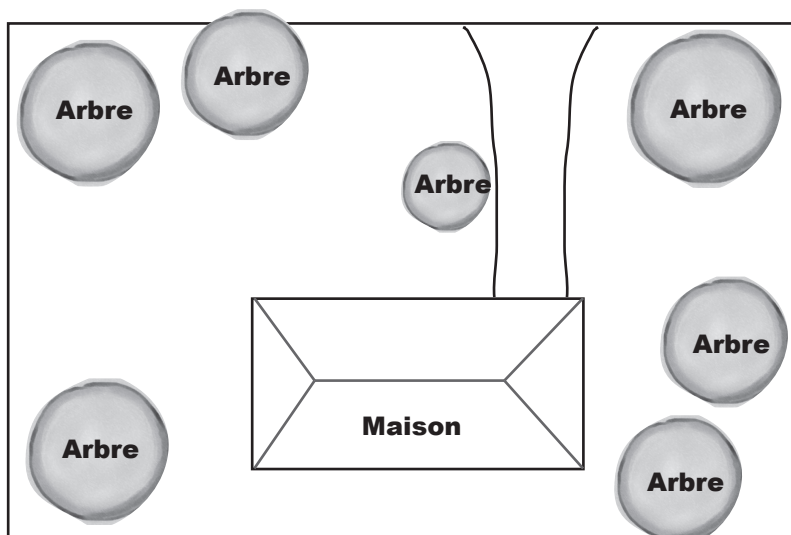
- Couramment utilisé dans les interfaces graphiques pour gérer des événements comme les clics de souris ou les entrées au clavier.
- L'exécution de la requête n'est pas garantie : elle peut échouer si aucun objet ne la traite (ce qui peut être un avantage ou un inconvénient).
- L'observation des caractéristiques à l'exécution et le débogage peuvent être difficiles.

Poids-mouche

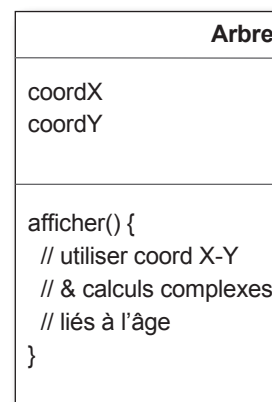
Utilisez le pattern Poids-mouche quand une instance d'une classe peut servir à fournir plusieurs « instances virtuelles ».

Un scénario

Vous avez écrit une super application de conception paysagère qui permet notamment d'ajouter des arbres autour d'une maison. Dans cette application, les arbres ne font pas grand-chose : ils ont un emplacement X-Y et savent se dessiner eux-mêmes dynamiquement en fonction de leur âge. Le problème est qu'un utilisateur peut vouloir d'importantes quantités d'arbres dans l'un de ses plans paysagers. Celui-ci pourrait ressembler au schéma ci-dessous :



Chaque instance d'Arbre maintient son propre état.



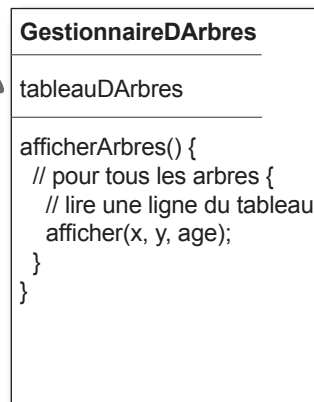
Le dilemme de votre client

Vous venez d'obtenir un marché avec un grand compte. C'est un client capital que vous dorlotez depuis des mois. Il va acheter mille licences de votre application et utiliser votre logiciel pour réaliser les études paysagères de quartiers entiers. Après avoir testé ce logiciel une semaine, le client se plaint que, lorsqu'il crée des bosquets importants, l'application commence à ramer sérieusement....

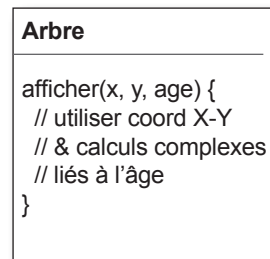
Pourquoi utiliser le pattern Poids-mouche ?

Et si au lieu d'avoir des milliers d'objets Arbre vous pouviez reconcevoir votre système de façon à n'avoir qu'une seule instance d'Arbre et un objet client qui maintiendrait l'état de TOUS vos arbres ? Eh bien c'est le pattern Poids-mouche !

Tout l'état, pour TOUS vos objets Arbre virtuels, est stocké dans ce tableau à deux dimensions.



Un seul et unique objet Arbre, sans état.



Avantages

- Réduit le nombre d'instances présentes au moment de l'exécution et économise de la mémoire.
- Centralise l'état de nombreux objets « virtuels ».

Emplois et inconvénients

- On utilise Poids-mouche lorsqu'une classe a de nombreuses instances qui peuvent toutes être contrôlées de manière identique.
- Un inconvénient de Poids-mouche est que, une fois que vous l'avez implémenté, les instances logiques individuelles de la classe ne sont plus capables de se comporter indépendamment des autres instances.

Interprète

Utilisez le pattern Interprète pour construire un interpréteur pour un langage.

Un scénario

Vous souvenez-vous du Simulateur de mare aux canards ?
Vous avez l'intuition qu'il pourrait devenir un outil pédagogique pour apprendre aux enfants à programmer.
Grâce au simulateur, chaque enfant peut contrôler un canard au moyen d'un langage simple. Voici un exemple du langage :

```
droite;  
tantque(faitjour) voler;  
cancaner;
```

Faire tourner le canard à droite.

Voler toute la journée...

...puis cancaner.

Vous puisez maintenant dans vos souvenirs de vos cours d'introduction à la programmation et vous écrivez la grammaire :

```
expression ::= <commande> | <sequence> | <repetition>  
sequence  ::= <expression> ';' <expression>  
commande  ::= droite | voler | cancaner  
repetition ::= tantque '(' <variable> ')' <expression>  
variable  ::= [A-Z,a-z]+
```

Un programme est une expression composée de séquences de commandes et de répétitions (les instructions « tantque »).

Une séquence est un ensemble d'expressions séparées par des points-virgules.

Nous avons trois commandes : droite, voler et cancaner.

Une instruction « tantque » est simplement composée d'une variable conditionnelle et d'une expression.

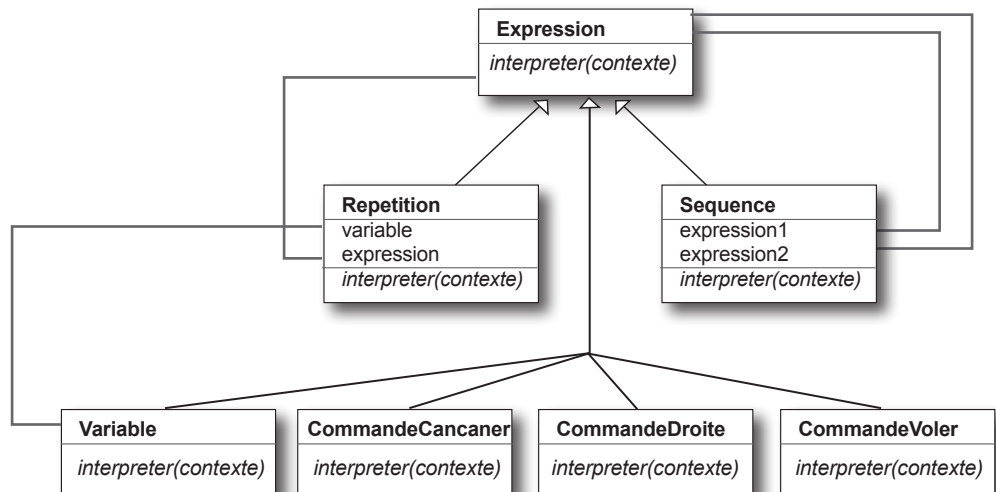
Et maintenant ?

Vous avez une grammaire. Il ne vous manque plus qu'un moyen de représenter et d'interpréter les phrases afin que vos élèves voient les effets de leur programmation sur les canards virtuels.



Comment implémenter un interpréteur

Quand vous devez implémenter un langage simple, le pattern Interprète permet de définir des classes pour représenter sa grammaire et un interpréteur pour interpréter les phrases. On utilise une classe pour représenter chaque règle du langage. Voici le langage Canard traduit en classes. Remarquez la correspondance directe avec la grammaire.



Pour interpréter le langage, appeler la méthode `interpreter()` sur chaque type d'expression. Cette méthode reçoit en argument un contexte qui contient le flot d'entrée que nous analysons, apparie l'entrée et l'évalue.

Avantages

- Représenter chaque règle de grammaire par une classe rend le langage facile à implémenter.
- Comme la grammaire est représentée par des classes, le langage est facile à modifier ou à étendre.
- En ajoutant des méthodes supplémentaires à la structure de classes, vous pouvez ajouter de nouveaux comportements qui dépassent l'interprétation, comme un affichage esthétique ou une validation plus élaborée.

Emplois et inconvénients

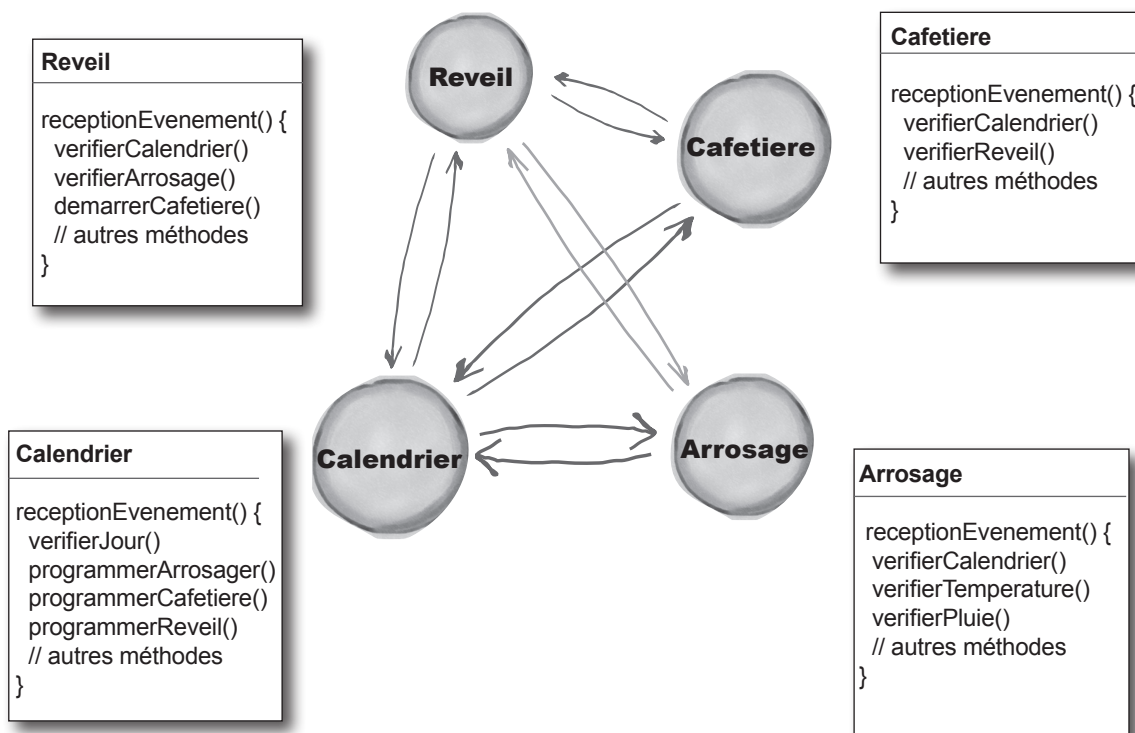
- Utilisez Interprète lorsque vous devez implémenter un langage simple.
- Interprète est approprié lorsque la grammaire est simple et que la simplicité prime sur l'efficacité.
- Interprète est utilisé dans les langages de scripts et de programmation.
- Ce pattern peut devenir difficile à manier quand le nombre de règles de grammaire est important, auquel cas un analyseur syntaxique et un générateur de compilation peuvent être préférables.

Médiateur

Utilisez le pattern Médiateur pour centraliser le contrôle et les communications complexes entre objets apparentés.

Un scénario

Léo possède une maison entièrement automatisée à l'aide de code Java grâce aux bons services de MaisonsDuFutur. Tous ses appareils sont conçus pour lui faciliter la vie. Quand Léo cesse de taper sur le bouton d'arrêt de son réveil, ce dernier dit à la cafetière de commencer à faire le café. Mais même si Léo a la belle vie, lui et les autres clients ne cessent de réclamer de nouvelles fonctionnalités : pas de café le week-end... arrêter l'arrosage automatique si une averse est annoncée... mettre le réveil plus tôt les jours de ramassage des encombrants...



Le dilemme de MaisonsDuFutur

Il devient vraiment très difficile de mémoriser dans quels objets se trouvent les règles et comment les objets sont liés les uns aux autres.

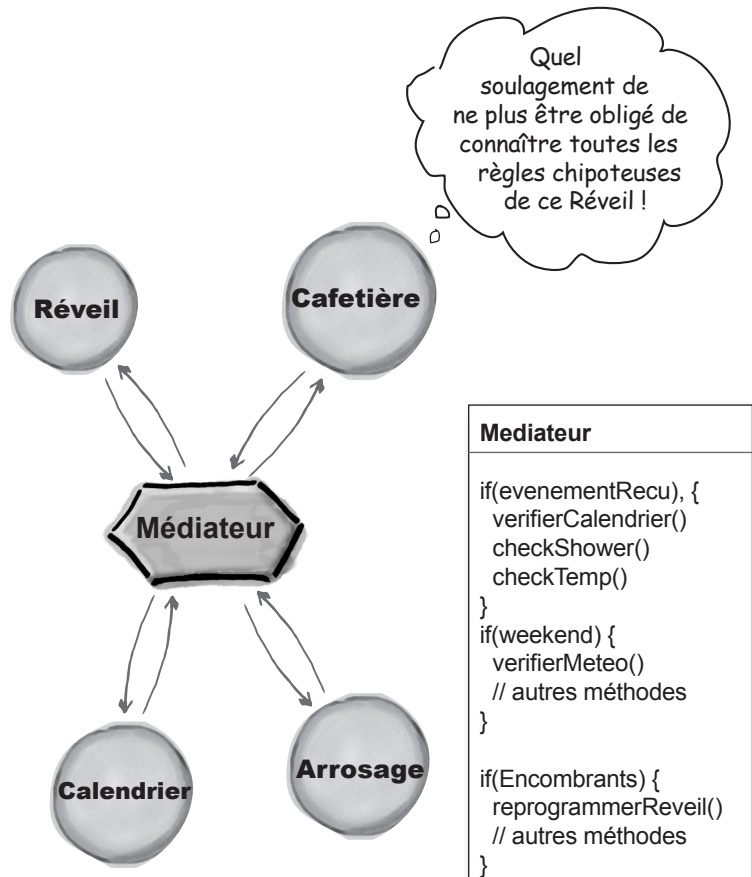
Médiateur en action...

L'ajout d'un Médiateur au système permet de simplifier grandement tous les appareils :

- Ils informent le médiateur quand leur état change.
- Ils répondent aux requêtes du Médiateur.

Avant l'insertion du Médiateur, tous les appareils devaient se connaître... ils étaient étroitement couplés. Avec le Médiateur en place, ces objets sont *totalemtent découplés* les uns des autres.

Le Médiateur contient toute la logique de contrôle pour l'ensemble du système. Quand un appareil existant a besoin d'une nouvelle règle ou quand un nouvel appareil est ajouté au système, vous savez que toute la logique nécessaire sera ajoutée au Médiateur.



Avantages

- Augmente la réutilisabilité des objets pris en charge par le Médiateur en les découplant du système.
- Simplifie la maintenance du système en centralisant la logique de contrôle.
- Simplifie et réduit la variété des messages échangés par les différents objets du système.

Emplois et inconvénients

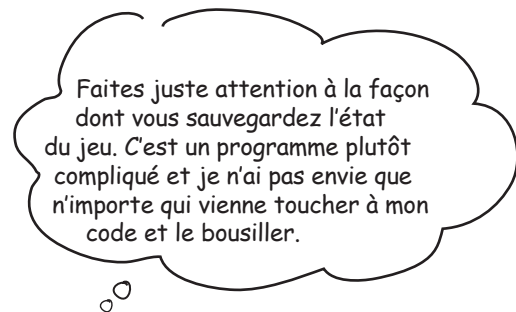
- Médiateur est couramment employé pour coordonner des composants d'IHM.
- En l'absence de conception soignée, l'objet Médiateur lui-même peut devenir exagérément complexe.

Memento

Utilisez le pattern Memento quand vous avez besoin de restaurer l'un des états précédents d'un objet, par exemple si l'utilisateur demande une « annulation ».

Un scénario

Votre jeu de rôles interactif rencontre un immense succès et a créé des légions de fans qui essaient tous d'atteindre le légendaire « niveau 13 ». Plus les utilisateurs terminent des niveaux de jeu de difficulté croissante, plus les chances de rencontrer une situation qui mettra fin au jeu augmentent. Les joueurs qui ont passé des jours et des jours à parvenir à un niveau avancé sont bien légitimement froissés quand leur personnage se casse la pipe et qu'ils doivent tout recommencer. Ils réclament à cor et à cri une commande de sauvegarde qui leur permette de mémoriser leur progression et de récupérer la plus grande partie de leurs efforts quand leur personnage est déloyalement « effacé ». La fonction de sauvegarde doit être conçue afin de permettre à un joueur ressuscité de reprendre le jeu au dernier niveau terminé avec succès.

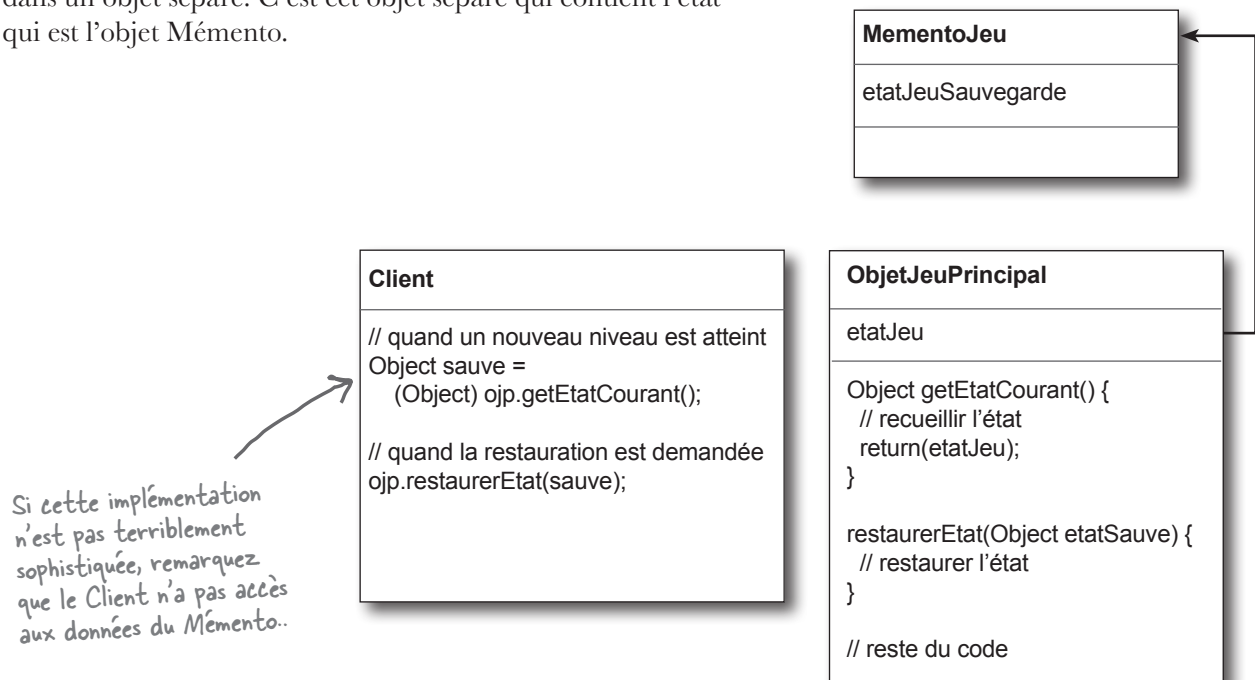


Memento au travail

Le Memento a deux objectifs :

- Sauvegarder un état important d'un objet clé d'un système.
- Maintenir l'encapsulation de l'objet clé.

Pour respecter le principe de responsabilité unique, il est également judicieux de mémoriser l'état que vous sauvegardez dans un objet séparé. C'est cet objet séparé qui contient l'état qui est l'objet Memento.



Avantages

- Séparer l'état sauvegardé de l'objet clé contribue à maintenir la cohésion.
- Les données de l'objet clé demeurent encapsulées.
- Les fonctions de restauration sont faciles à implémenter.

Emplois et inconvénients

- Le Memento est utilisé pour sauvegarder un état.
- L'inconvénient de l'emploi de Memento est que la sauvegarde et la restauration d'un état peuvent être très longues.
- Dans les systèmes Java, envisagez d'employer la sérialisation pour sauvegarder l'état d'un système.

Prototype

Utilisez le pattern Prototype quand la création d'une instance d'une classe donnée est coûteuse ou compliquée.

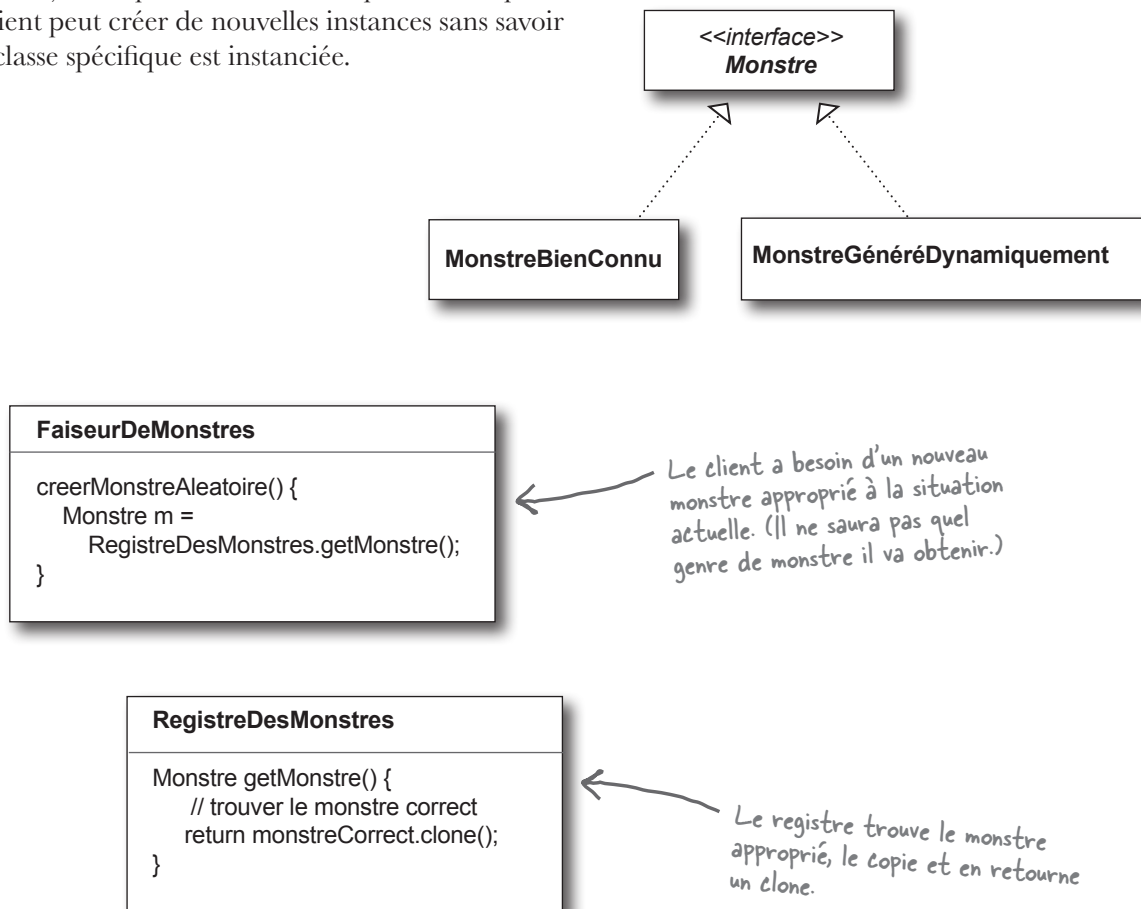
Un scénario

Votre jeu de rôles interactif est doté d'un insatiable appétit pour les créatures monstrueuses. À mesure que vos héros poursuivent leurs pérégrinations dans un paysage créé dynamiquement, ils rencontrent une chaîne sans fin d'ennemis qu'ils doivent anéantir. Vous aimeriez que les caractéristiques de vos monstres évoluent avec les changements du paysage. Ce ne serait pas très logique de laisser des dragons ailés suivre vos personnages dans les royaumes sous-marins. Enfin, vous désirez permettre aux joueurs avancés de créer leurs propres monstres personnalisés.



Prototype à la rescousse

Le pattern Prototype permet de créer de nouvelles instances en copiant des instances existantes. (En Java, cela signifie généralement l'emploi de la méthode `clone()`, ou de la désérialisation quand on a besoin de copies en profondeur.) Un aspect essentiel de ce pattern est que le code client peut créer de nouvelles instances sans savoir quelle classe spécifique est instanciée.



Avantages

- Masque au client les complexités de la création des nouvelles instances.
- Permet au client de générer des objets dont le type n'est pas connu.
- Dans certaines circonstances, la copie d'un objet peut être plus efficace que la création d'un nouvel objet.

Emplois et inconvénients

- Envisagez Prototype quand un système doit créer de nouveaux objets de types différents dans une hiérarchie de classes complexe.
- Un inconvénient de l'emploi de Prototype est que la copie d'un objet peut parfois être compliquée.

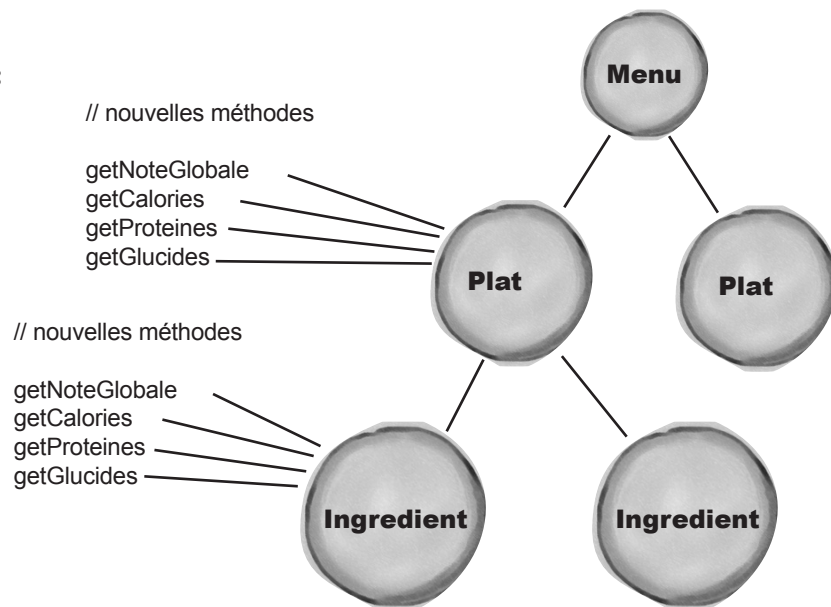
Visiteur

Utilisez le pattern Visiteur quand vous voulez ajouter des capacités à un ensemble composite d'objets et que l'encapsulation n'est pas importante.

Un scénario

Les clients qui fréquentent la Cafeteria d'Objectville et la Crêperie d'Objectville se préoccupent de plus en plus de leur santé. Ils demandent des informations nutritionnelles avant de commander leur repas. Comme les deux établissements acceptent de prendre des commandes spéciales, certains clients réclament même ces informations ingrédient par ingrédient.

La solution proposée par Léon :

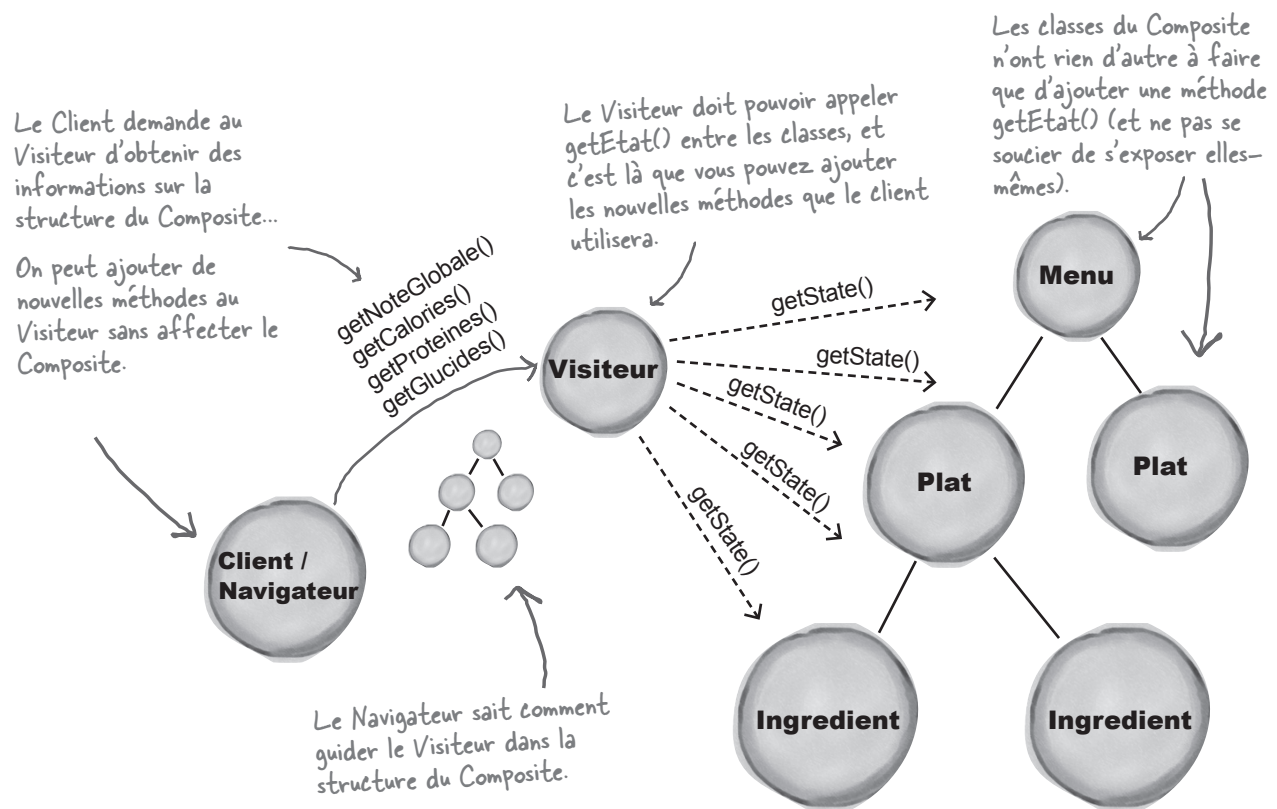


Les inquiétudes de Noël...

« On dirait que nous sommes en train d'ouvrir la boîte de Pandore. Qui sait quelle nouvelle méthode il va falloir encore ajouter ? Et chaque fois qu'on ajoute une méthode, il faut le faire en deux endroits. De plus, que se passe-t-il si nous voulons améliorer l'application de base avec, par exemple, une classe Recette ? Dans ce cas, il faudra trois séries de modifications... »

Voici le Visiteur

Le Visiteur doit parcourir chaque élément du Composite : cette fonctionnalité se trouve dans un objet Navigateur. Le Visiteur est guidé par le Navigateur et recueille l'état de tous les objets du Composite. Une fois l'état recueilli, le Client peut demander au Visiteur d'exécuter différentes opérations sur celui-ci. Quand une nouvelle fonctionnalité est requise, seul le Visiteur doit être modifié.



Avantages

- Permet d'ajouter des opérations à la structure d'un Composite sans modifier la structure elle-même.
- L'ajout de nouvelles opérations est relativement facile.
- Le code des opérations exécutées par le Visiteur est centralisé.

Inconvénients

- L'encapsulation des classes du Composite est brisée.
- Comme une fonction de navigation est impliquée, les modifications de la structure du Composite sont plus difficiles.

Index

A

- abstraction [280](#), [612](#)
- Adaptateur [235](#)
 - définition [243](#)
 - explication [241](#)
- adaptateurs
 - combinaison de patterns [504](#)
 - de Swing [306](#)
 - d'objet et de classe [244](#)
 - orientés objet [237](#)
- adapter une Enumeration à un Iterator [249](#)
- algorithmes [275](#)
- améliorer l'itérateur [332](#)
- Annexe [611](#)
- anti-patterns [606](#)
- appel de méthode [438](#)
- appels de méthode
 - limiter [267](#)
- applets [307](#)
- A-UN [23](#)
- à vos crayons [5](#), [42](#), [54](#), [61](#), [83](#), [84](#), [99](#), [124](#), [137](#), [205](#), [225](#), [268](#), [284](#), [322](#), [396](#), [400](#), [409](#), [421](#), [483](#), [520](#), [589](#)

B

- bande des quatre. Voir Gang of Four
- bases de l'OO [32](#)
- BeanPersonne [476](#), [479](#)
- boîte à outils de concepteur [32](#), [74](#), [105](#), [162](#), [186](#), [230](#), [270](#), [311](#), [380](#), [423](#), [491](#), [560](#), [608](#)
- BufferedInputStream [100](#)

C

- Chaîne de responsabilité [616](#)
- changement [111](#), [339](#)
 - anticiper [14](#), [53](#)
 - développement [8](#)
- Chargement de l'Invocateur [201](#)
- Christopher Alexander [602](#)
- classes
 - comportement [14](#)
 - diagramme [52](#)
 - étendre [86](#)
 - proxy Java [474](#)
- client contrôleur [454](#)
- client distant [448](#)
- code prêt à l'emploi [469](#), [494](#), [496](#), [564](#), [566](#), [568](#), [570](#), [572](#), [574](#), [576](#)
- Cohésion [339](#)
- Commande [191](#)
 - définition [206](#)
 - diagramme de classes [207](#)
- commandes, annulation [216](#)
- communication sur le réseau [434](#)
- Comparable [302](#)
- compareTo() [301](#)
- comportement
 - encapsuler [22](#)
 - modifier [20](#)
- Composite [315](#), [533](#), [558](#)
 - comportement [361](#), [363](#)
 - définition [356](#)
 - diagramme de classes [358](#)
 - et Itérateur [368](#)
 - implémenter [362](#)
 - patterns combinés [513](#)
 - sécurité [367](#)

composition 23, 85, 93, 309, 397
 conception souple 614
 Contrôleur 542
 couplage faible 53, 109
 créer des objets 134
 créer un itérateur 326

D

Décorateur 79, 88, 105
 définition 91
 décorateurs d'E/S Java 100
 écrire 102
 Décoration des classes de java.io 101
 découpler 334
 dépendances 138
 des objets 138
 inversion 139
 Dépendez d'abstractions. Ne dépendez pas de classes
 concrètes. 139
 Design Pattern. Voir aussi patterns
 catégories 592-593
 classes 591
 définition 579
 et Model2 557
 objets 591
 organiser 589
 utiliser 29
 diffusion 45

E

Efforcez-vous de coupler faiblement les objets qui
 interagissent. 53
 encapsuler 9
 algorithmes 275
 appels de méthode 206
 construction 614
 création des objets 114
 invocation 191
 itération 323
 objets 242, 473, 508
 requête 206
 Encapsulez ce qui varie 9, 75, 136, 397
 Enumeration 248

EnumerationIterator 250
 EST-UN 23
 État 385
 définition 410
 états
 classes et conception 399
 implémenter les classes 401
 transition 408
 étendre facilement les classes 86

F

Fabrication 131
 définition 134
 Fabrication et Fabrique Abstraite 158
 Fabrique 114
 Fabrique Abstraite 153
 définition 156
 Fabrique Simple, définition 117
 Façade 260
 235, 258
 définition 264
 Ne parlez pas aux inconnus 269
 vs. Adaptateur 260
 Face à face
 Décorateur et Adaptateur 252
 État et Stratégie 418
 Patron de méthode et Stratégie 308
 Proxy et Décorateur 472
 Sujet et Observateur 62
 Factory Method 134
 Famille d'algorithmes. Voir Pattern Stratégie
 FilterInputStream 100, 102
 for/in 349

G

Gamma, Erich 601
 Gang of Four
 Gamma, Erich 601
 Helm, Richard 601
 Johnson, Ralph 601
 Vlissides, John 601
 gestionnaires d'invocations 480
 getInstance() 181

H

Helm, Richard [601](#)
 héritage
 réutilisation [5-6](#)
 vs. composition [93](#)
 hiérarchies de classes parallèles [132](#)
 Hillside Group [603](#)
 Home Automation or Bust, Inc. [192](#)

I

identifiez les aspects de votre application qui varient
 et séparez-les de ceux qui demeurent constants [9](#)
 impact du changement [84](#)
 implémentation [57](#), [612](#)
 distante
 créer [443](#)
 implémenter
 la façade [262](#)
 une méthode adaptateur [292](#)
 InputStreams [102](#)
 instanceof [367](#)
 instancier [110](#)
 l'objet Proxy [483](#)
 UN SEUL OBJET [170](#)
 interface [6](#)
 Commande [203](#)
 distante [452](#)
 créer [442](#)
 Etat [399](#)
 Interprète
 [620](#)
 inversion des dépendances [139](#)
 appliquer [140](#)
 conseils [143](#)
 Itérateur [513](#)
 [315](#), [325](#)
 définition [336](#)
 itérateur Composite [369](#)
 itérateur null [372](#)
 itérateurs et collections [348](#)
 en Java 5/6 [349](#)
 Iterator [248](#)

J

java.rmi.Remote [442](#)
 java.util.Iterator [333](#)
 java.util.Observable [71](#)
 JDK [72](#)
 Johnson, Ralph [601](#)
 JVM [179](#), [181](#), [436](#)

L

Le principe d'Hollywood [296](#)
 Les classes doivent être ouvertes à l'extension, mais
 fermées à la modification. [86](#)
 LineNumberInputStream [100](#)
 Loi de Déméter [267](#)

M

machine virtuelle Java. Voir JVM
 macro-commande [225](#)
 Maître et disciple [23](#), [85](#), [136](#), [592](#), [596](#)
 Médiateur
 [622](#)
 Memento
 [624](#)
 méthodes
 abstraite [120](#)
 adaptateur
 utiliser [293](#)
 adaptateurs et Patron de méthode [292](#)
 de fabrique [125](#)
 distantes [437](#)
 mieux vivre avec les patterns [577](#)
 Model 2 [550](#)
 Modèle [546](#)
 Modèle-Vue-Contrôleur [526](#), [529](#)
 Monteur [614](#)
 mots-croisés [33](#), [76](#), [163](#), [187](#), [231](#), [271](#), [310](#), [378](#), [490](#)
 multithread [180](#)
 MVC
 et le Web [549](#)

N

Ne nous appelez pas, nous vous appellerons. 296
 Ne parlez pas aux inconnus. Ne parlez qu'à vos amis immédiats. 265
 new 110

O

Objectville 488, 577
 objets
 contrôler l'accès 429
 Observable 71
 Observateur 37, 74, 557
 définition 51
 Java 64
 Observer/Observable
 implémentation 72
 OOPSLA 603
 ordre d'évaluation des notifications 70
 Ouvert-Fermé 86

P

paint() 306
 Patron de méthode 275, 286
 définition 289
 tri 300
 pattern
 Memento 624
 patterns 28
 Adaptateur et Façade 235-274
 catalogue 584
 catégorie 590
 Chaîne de responsabilité 616
 Commande 191
 comportemental 589
 composés 499-576
 diagramme de classes 524
 contre-indications 596, 598
 créateurs 589
 d'application 604
 d'architecture 604
 de classes 591
 de conception d'interfaces 605
 Décorateur 79
 de domaine 604

de processus 605
 d'organisation 605
 écrire 587
 État 385-428
 fabriques 109
 Fabriques 109-168
 intégré 67
 Interprète 620
 introductions aux 1-36
 Itérateur et Composite 315-384
 Médiateur 622
 Monteur 614
 MVC 532
 Observateur 37-78
 OO 32
 Patron de méthode 275-314
 penser 594
 Poids-mouche 618
 Pont 612
 Prototype 626
 Proxy 429-498
 ressources 602
 restants 611
 Singleton 169-190
 structurels 589
 Visiteur 628
 zoo 604

Poids-mouche 618
 points d'impact 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608
 Pont 612
 POO 2
 Préférez la composition à l'héritage 23
 principe Ouvert-Fermé 86
 principes de conception 9, 23, 53, 86, 139, 265, 339
 enfreindre 82
 principes OO 32
 Problème de conception 25, 133, 279, 395, 468, 492, 542, 563
 programmer
 une implémentation 12
 une interface 12
 vs. implémentation 11
 un supertype 12
 Prototype 626
 proxy 456
 distant 434, 462
 types de 488
 virtuel 462
 Proxy 429
 définition 460

Q

Qui fait quoi ? [202](#), [254](#), [298](#), [379](#), [422](#), [487](#), [588](#)

R

ramasse-miettes [184](#)

refactorisation [595](#)

relation [51](#)

RemoteException [442](#), [443](#), [452](#)

remove() [250](#)

requêtes

files [228](#)

journaliser [229](#)

responsabilité des classes [339](#)

rmi [436](#)

RMI

vue d'ensemble [440](#)

rmic [441](#)

rmiregistry [441](#), [444](#)

S

service distant [441](#)

servlets [551](#)

Singleton [169](#)

définition [177](#)

implémentation [173](#)

souches [441](#)

et squelettes [444](#)

souscription [45](#)

squelettes [441](#)

Stratégie [533](#), [545](#), [558](#)

[24](#), [545](#)

Swing [306](#)

U

un-à-plusieurs [51](#)

Une classe ne doit avoir qu'une seule raison de changer.
[339](#)

UnicastRemoteObject [443](#), [452](#)

V

verrouillage à double vérification [182](#)

Visiteur [628](#)

Vlissides, John [601](#)

vocabulaire [600](#)

Vue [540](#)

Colophon



Toutes les maquettes intérieures ont été conçues par Eric Freeman, Elisabeth Freeman, Kathy Sierra et Bert Bates. Kathy et Bert ont créé le look & feel de la collection Tête la première. L'édition française de cet ouvrage a été traduite par **Marie-Cécile Baland**.

Marie-Cécile Baland est une traductrice passionnée. Elle a traduit de nombreux ouvrages de référence dans le domaine des langages objets, dont le célèbre *Java, Tête la première*. Sa culture ne se limite pas aux domaines de l'informatique : elle impressionne aussi ses amis qui sont souvent, est-ce un hasard, des informaticiens, par ses connaissances en littérature, en histoire, en botanique ou... en cuisine. Son humour subtil, toujours présent, vous aura rendu, nous l'espérons, la lecture en français de cet ouvrage agréable. Le code a été localisé par Frédéric Laurent et Marie-Cécile ; la traduction a été relue par Frédéric et Michel Beteta. **Frédéric Laurent** est spécialiste des architectures objet et des technologies J2EE et XML. Il s'intéresse à l'exploitation multisupport de contenu (grâce, notamment, aux technologies XML) ainsi qu'aux informations sémantiques (web sémantique et métadonnées). Il a par ailleurs co-traduit *Java et XSLT*, *Java plus rapide, plus léger* et *XML en concentré*. **Michel Beteta**, consultant en nouvelles technologies et architectures Java, est impliqué dans toutes sortes de prestations de conseil autour des infrastructures J2EE. Ses heures de relecture au son d'une musique électro (merci à Mylo !) et devant environ une tonne de chips l'ont conforté dans sa vision d'une approche didactique, pragmatique et claire de l'apprentissage des bonnes pratiques de programmation et qui se retrouve dans la collection *Tête la première*.

Institut Tête la première

Et maintenant, un dernier mot de l'institut Tête la première...

Nos chercheurs de renommée mondiale travaillent jour et nuit, menant une course effrénée pour découvrir les mystères de la vie, de l'univers et du Tout — avant qu'il ne soit trop tard. Jamais auparavant une équipe poursuivant des buts aussi nobles et d'une telle envergure n'a été réunie. Actuellement, nous consacrons toute notre énergie collective et tous nos pouvoirs mentaux à créer le nec plus ultra des machines à apprendre. Une fois que nous l'aurons perfectionnée, vous, et bien d'autres pourrez vous joindre à notre quête !

Vous êtes aujourd'hui l'un des heureux possesseurs du prototype de cette machine. Mais seuls des raffinements nous permettront d'atteindre notre but. En tant qu'utilisateur pionnier de cette nouvelle technologie, vous pouvez nous aider en nous envoyant vos notes de terrain sur votre progression à fieldreports@wickedlysmart.com.





Distribon



Sans votre aide, la prochaine génération ne connaîtra peut-être jamais les joies de notre distributeur de bonbons. Aujourd'hui, un code mal conçu et totalement dépourvu de souplesse met nos distributeurs Java en danger. Distribon ne laissera pas cette catastrophe se produire. Nous nous sommes dévoués à une cause : vous aider à améliorer vos compétences en Java et en conception OO, afin que vous puissiez nous aider à développer notre prochaine génération de machines Distribon.

Allons, les grilles-pain Java font tellement années 90.

Rendez-nous visite sur <http://www.distribon.com>.



Distribon, SARL.
Le monde dans lequel les
distributeurs
ne sont jamais vides