Génie logiciel 5A - Parcours SAGI

Patrons de conception

Nicolas Delanoue

Université d'Angers - Polytech Angers

Objectif du cours

Focalisation sur la conception : Bonnes pratiques & “Patron de conception”.

Travaux pratiques

- Exercices : conception, refactorying, patron de conception,
- Principe : programme “mal conçu” à restructurer en appliquant les modèles adaptés.

Objectif du cours

Focalisation sur la conception : Bonnes pratiques & “Patron de conception”.

Travaux pratiques

- Exercices : conception, refactorying, patron de conception,
- Principe : programme “mal conçu” à restructurer en appliquant les modèles adaptés.

Références

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns : Elements of Reusable Object-Oriented Software, 1995,
- Exemples en ligne (utilisés en TD) : `https://www.dofactory.com/net/design-patterns`

Le point de départ : le quoi (analyse des besoins)

Les informations disponibles :

- ensemble de contraintes et besoins non fonctionnels,

Le point de départ : le quoi (analyse des besoins)

Les informations disponibles :

- ensemble de contraintes et besoins non fonctionnels,
- une modélisation cas d'utilisation : ensemble des fonctionnalités du point de vue utilisateur,

Le point de départ : le quoi (analyse des besoins)

Les informations disponibles :

- ensemble de contraintes et besoins non fonctionnels,
- une modélisation cas d'utilisation : ensemble des fonctionnalités du point de vue utilisateur,
- une modélisation objet (classe) : ensemble des entités manipulées par le système,

Le point de départ : le quoi (analyse des besoins)

Les informations disponibles :

- ensemble de contraintes et besoins non fonctionnels,
- une modélisation cas d'utilisation : ensemble des fonctionnalités du point de vue utilisateur,
- une modélisation objet (classe) : ensemble des entités manipulées par le système,
- une modélisation comportementale les scénarios relatifs aux fonctionnalités, faisant intervenir les objets.

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),
    - validable par le client

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),
    - validable par le client
  - Représentation technique, *technical design* :

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),
    - validable par le client
  - Représentation technique, *technical design* :
    - sa structure et le comportement (langage technique),

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),
    - validable par le client
  - Représentation technique, *technical design* :
    - sa structure et le comportement (langage technique),
    - validable par les développeurs.

L'activité de conception : le comment

- Informations produites à destination du client et des développeurs
  - Représentation conceptuelle, *conceptual design* :
    - fonctionnalités de la solution proposée (langage simple),
    - validable par le client
  - Représentation technique, *technical design* :
    - sa structure et le comportement (langage technique),
    - validable par les développeurs.
- Phase itérative : échanges entre le client et le développeur.

De la conception au programme : les activités clés

1. L'identification de certaines caractéristiques/contraintes du système
2. Conception : ce que l'on va programmer (modules, classes)
3. Choix de la méthodologie : prototypage ou non ?
4. Affectation des responsabilités aux développeurs,
5. La programmation : enfin ! ! !

Evidemment

Communiquer, échanger pour converger vers la solution "objectivement" adaptée.

Identifier les caractéristiques/contraintes du système

1. Le choix du ou des langages, voire même du paradigme.
   - Un paradigme/langage est choisi en fonction de plusieurs critères : Mécanismes, Performances, Expertise, Composants disponibles, ...
   - Dans ce cours, on considère le paradigme objet.

Identifier les caractéristiques/contraintes du système

1. Le choix du ou des langages, voire même du paradigme.
   - Un paradigme/langage est choisi en fonction de plusieurs critères : Mécanismes, Performances, Expertise, Composants disponibles, ...
   - Dans ce cours, on considère le paradigme objet.
2. Les composants *off-the-shelf* (pris sur l'étagère) :
   - propriétaires ou opensources,
   - avantage : moins de développement - moins de maintenance,
   - inconvénient : peut imposer des contraintes (adaptation à l'architecture).

Identifier les caractéristiques/contraintes du système

1. Le choix du ou des langages, voire même du paradigme.
   - Un paradigme/langage est choisi en fonction de plusieurs critères : Mécanismes, Performances, Expertise, Composants disponibles, ...
   - Dans ce cours, on considère le paradigme objet.
2. Les composants *off-the-shelf* (pris sur l'étagère) :
   - propriétaires ou opensources,
   - avantage : moins de développement - moins de maintenance,
   - inconvénient : peut imposer des contraintes (adaptation à l'architecture).
3. Les contraintes matérielles et d'environnement :
   - architecture client lourd ou distribuée (création, synchronisation des objets),
   - mono ou multi-plateforme : choix des systèmes d'exploitation.

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier
5. Les contraintes d'accès aux données ou procédures
   - Typiquement cas des utilisateurs / administrateurs

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier
5. Les contraintes d'accès aux données ou procédures
   - Typiquement cas des utilisateurs / administrateurs
6. Le mode de séquencement des actions (*control flow*)
   - *procedure drivencontrol* : séquencement classique,

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier
5. Les contraintes d'accès aux données ou procédures
   - Typiquement cas des utilisateurs / administrateurs
6. Le mode de séquencement des actions (*control flow*)
   - *procedure drivencontrol* : séquencement classique,
   - *event driven control* : une boucle principale infinie attend un événement externe et le répartit aux différents objets selon l'information de l'événement,

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier
5. Les contraintes d'accès aux données ou procédures
   - Typiquement cas des utilisateurs / administrateurs
6. Le mode de séquencement des actions (*control flow*)
   - *procedure drivencontrol* : séquencement classique,
   - *event driven control* : une boucle principale infinie attend un événement externe et le répartit aux différents objets selon l'information de l'événement,
   - *thread* : traitements concurrents, difficulté de débogage, gestion des dépendances (e.g. accès données).

Identifier les caractéristiques/contraintes du système

4. Les contraintes de persistance
   - Le choix des données persistantes (durée de vie supérieure à l'exécution du logiciel)
   - Mode de persistance : base de données ou simple fichier
5. Les contraintes d'accès aux données ou procédures
   - Typiquement cas des utilisateurs / administrateurs
6. Le mode de séquencement des actions (*control flow*)
   - *procedure drivencontrol* : séquencement classique,
   - *event driven control* : une boucle principale infinie attend un événement externe et le répartit aux différents objets selon l'information de l'événement,
   - *thread* : traitements concurrents, difficulté de débogage, gestion des dépendances (e.g. accès données).
7. Les conditions limites :
   - comment le système démarre/s'arrête,
   - politique de gestion des exceptions.

En conclusion,

Ces caractéristiques et contraintes peuvent impacter l'architecture générale, détaillée et le code.

Définition

Une *conception descendante* consiste à d'abord effectuer la conception haut niveau puis la conception bas niveau.

Définition

La conception haut-niveau repose sur la décomposition en sous-systèmes logiques. Les packages/modules & classes ne sont pas détaillées.

Définition

La conception *bas-niveau* se rapproche du code, elle est plus détaillée. En général, on spécifie les interfaces en décrivant :

- la signature des méthodes (nom et paramètres),
- la portée des éléments, i.e. source des dépendances. On parle de *visibilité*.

Définition

La conception *bas-niveau* se rapproche du code, elle est plus détaillée. En général, on spécifie les interfaces en décrivant :

- la signature des méthodes (nom et paramètres),
- la portée des éléments, i.e. source des dépendances. On parle de *visibilité*.

Avantages

- Restructuration des modèles pour réduire la complexité. i.e. réduction du nombre d'associations.
- Intégration de classes complémentaires, e.g. intégration des éléments de couplage avec composants off-the-shelf.

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.
- tirer partie des mécanismes de l'orientation objet (Héritage, polymorphisme, programmation générique),

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.
- tirer partie des mécanismes de l'orientation objet (Héritage, polymorphisme, programmation générique),
- s'inspirer des modèles/principes éprouvés. Solutions *type* à des problèmes de conception récurrents issus de l'expérience :

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.
- tirer partie des mécanismes de l'orientation objet (Héritage, polymorphisme, programmation générique),
- s'inspirer des modèles/principes éprouvés. Solutions *type* à des problèmes de conception récurrents issus de l'expérience :
  - *GRASP* (General Responsability Assignement Software Patterns),

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.
- tirer partie des mécanismes de l'orientation objet (Héritage, polymorphisme, programmation générique),
- s'inspirer des modèles/principes éprouvés. Solutions *type* à des problèmes de conception récurrents issus de l'expérience :
  - *GRASP* (General Responsability Assignement Software Patterns),
  - *Architectural patterns* (e.g. applications web en 3 couches),

Bonnes pratiques en conception

- Réutilisation de l'existant (Composants *off-the-shelf*), i.e. ne pas réinventer la roue.
- tirer partie des mécanismes de l'orientation objet (Héritage, polymorphisme, programmation générique),
- s'inspirer des modèles/principes éprouvés. Solutions *type* à des problèmes de conception récurrents issus de l'expérience :
  - *GRASP* (General Responsability Assignement Software Patterns),
  - *Architectural patterns* (e.g. applications web en 3 couches),
  - *Design patterns* du *Gang of Four*.

Utilisation de modèles (patron de conception)

- Avantages :
  - conception de qualité : utile pour la maintenance, la correction des erreurs
  - modifications et extensions plus facile du logiciel : anticipation des changements
  - meilleur compréhension (modèles concis et connus) : communication inter-développeurs, visualisation des systèmes complexes.
- Inconvénients :
  - Il faut adapter leur implémentation au contexte : requiert un peu d'expérience.

Définition - GRASP (General Responsibility Assignment Software Patterns)

GRASP est une famille de patrons de conception qui donnent des conseils généraux sur l'assignation de responsabilité aux classes et objets dans une application.

Une responsabilité est vue au sens conception (exemples : création, détention de l'information, ...) :

- elle est relative aux méthodes et données des classes,
- elle est assurée à l'aide d'une ou plusieurs méthodes,
- elle peut s'étendre sur plusieurs classes.

En UML, l'assignation de responsabilité peut être appliquée à la conception des diagrammes de collaboration.

Différents modèles et principes

- *Expert en information :* Affecter les responsabilités aux classes détenant les informations nécessaires.
- *Créateur :* Déterminer quelle classe a la responsabilité de créer des instances d'une autre classe.
- *Faible couplage :* Diminuer le couplage des classes afin de réduire leur inter-dépendances dans le but de faciliter la maintenance du code.
- *Forte cohésion :* Avoir des sous-classes terminales très spécialisées.
- *Contrôleur :* Affecter la responsabilité de réception et traitement de messages systèmes.
- *Polymorphisme :* Affecter un nouveau comportement à l'endroit de la hiérarchie de classes où il change.
- *Fabrication pure :* Créer des classes séparées pour des fonctionnalités génériques qui n'ont aucun rapport avec les classes du domaine applicatif.
- *Indirection :* Découpler des classes en utilisant une classe intermédiaire.
- *Protection :* Ne pas communiquer avec des classes inconnues.

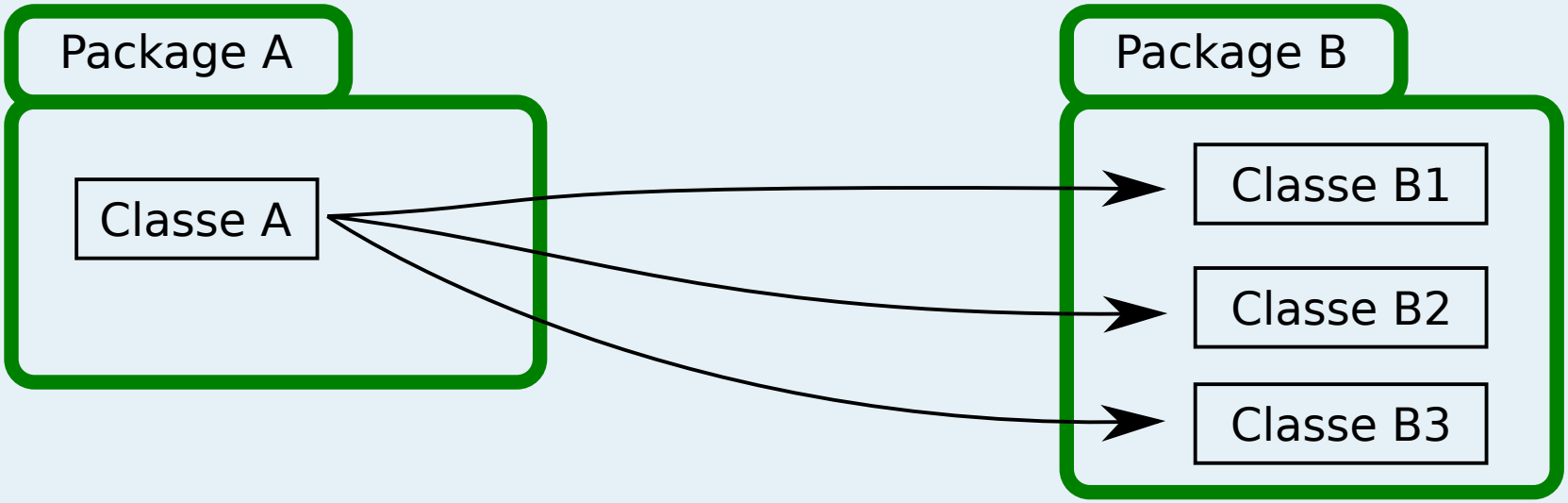
Exemple de patron GRASP : *le couplage faible*

Le couplage mesure à quel point les éléments sont interconnectés.

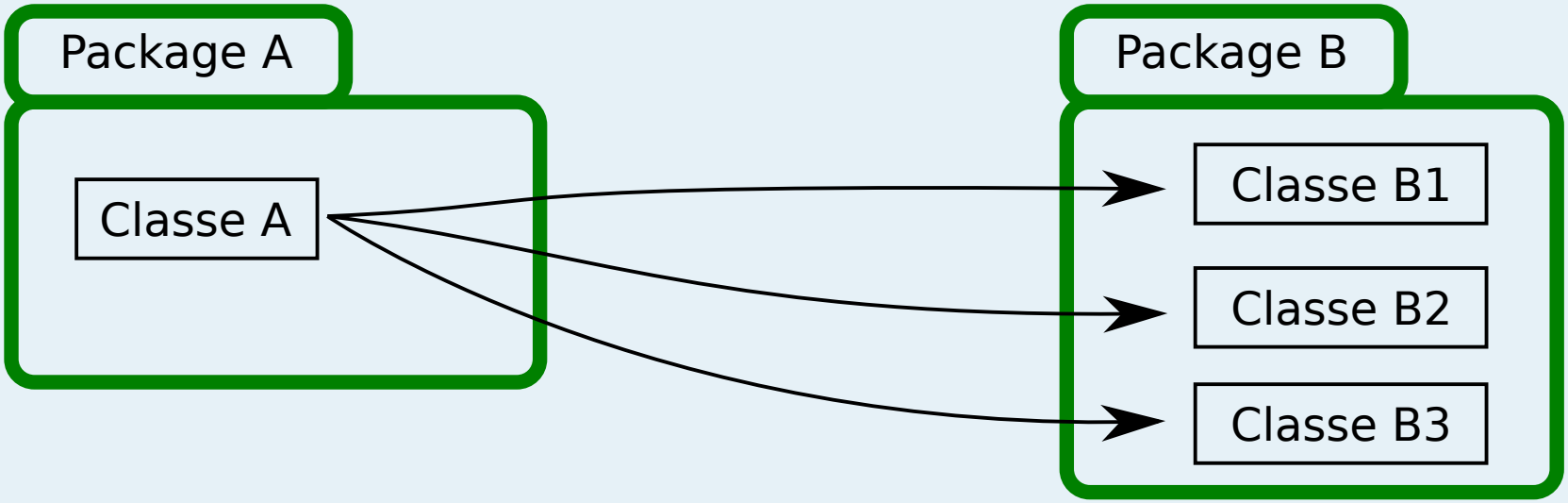
Un couplage faible engendre les avantages suivants :

- Peu de dépendances entre les classes,
- Un changement dans une classe a un faible impact sur les autres classes,
- Forte ré-utilisabilité potentielle.

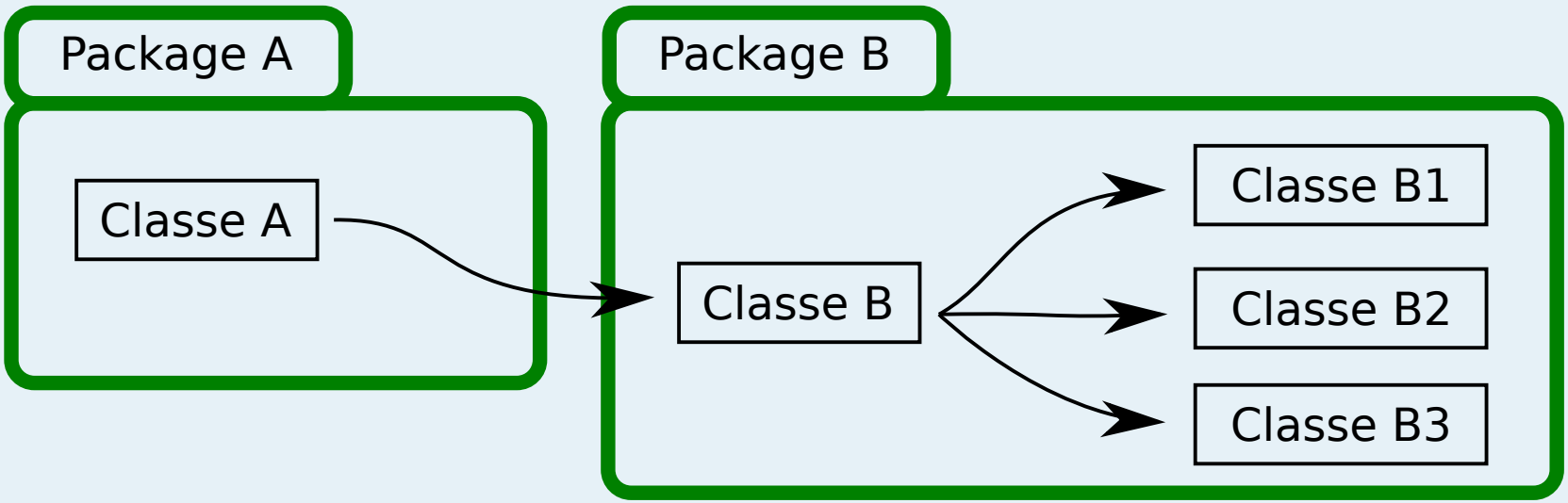
Avant



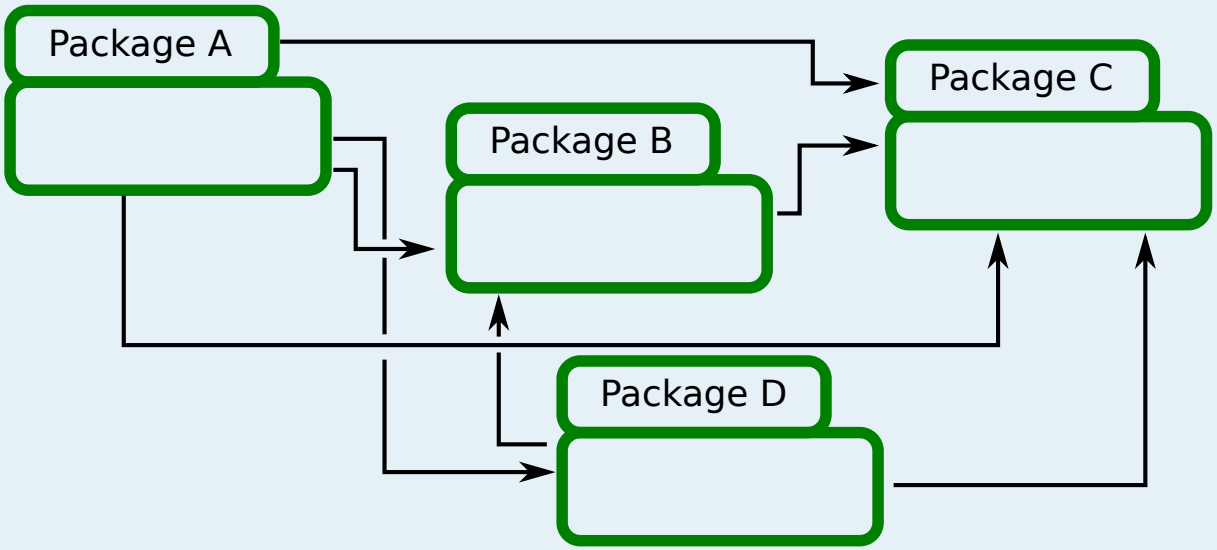
Avant


Package A
Classe A
Package B
Classe B1
Classe B2
Classe B3

Après

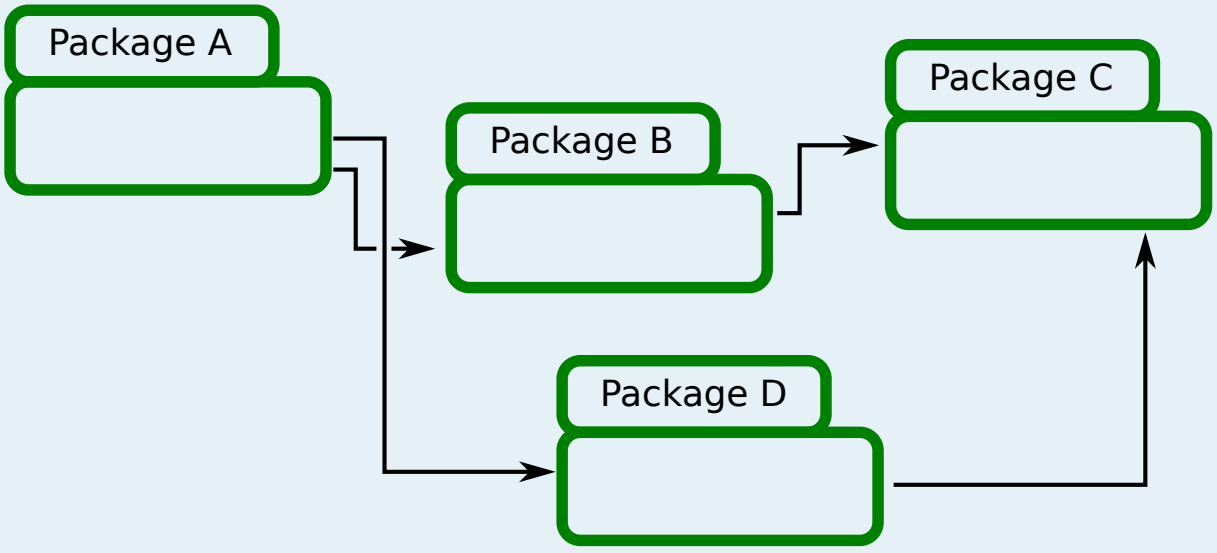

Package A
Classe A
Package B
Classe B
Classe B1
Classe B2
Classe B3

Mauvais exemple



- les dépendances sont difficilement compréhensibles.
- les dépendances sont nombreuses : la moindre modification d'un élément peut impacter tout le système.

Meilleur exemple



- Un sous-système sera moins sensible aux modifications d'autres sous-systèmes.
- remarque applicable au cas des classes (limiter les associations),
- coût : augmentation des indirections, probable impact sur les performances.

Patron GRAPS - Créateur

Ce patron aide à choisir la bonne classe qui aura la responsabilité de créer des objets.

En général, une classe B doit être responsable de la création des instances de la classe A si l'un, ou de préférence plusieurs, des critères suivants s'appliquent :

- Les Instances de B agrègent des instances de A,
- Les Instances de B contiennent des instances de A,
- Les Instances de B sont en étroite collaboration avec les instances de A,
- Les Instances de B détiennent l'information nécessaire pour instancier A.

Définition

Un modèle architectural (*Architectural patterns*) est une solution générale réutilisable à un problème courant de l'architecture logicielle dans un contexte donné.

Liste des 10 Architectural patterns les plus communs

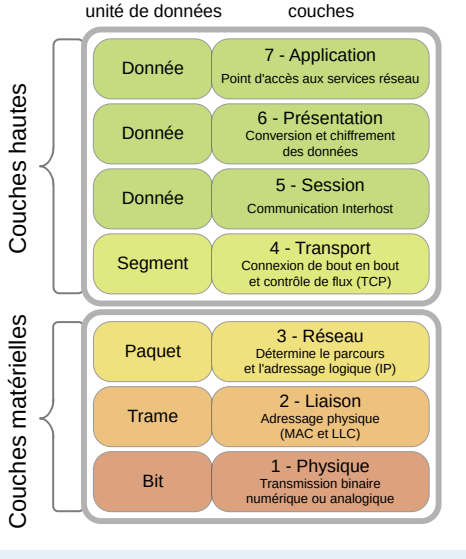
- Layered pattern
- Client-server pattern
- Master-slave pattern
- Pipe-filter pattern
- Broker pattern
- Peer-to-peer pattern
- Event-bus pattern
- Model-view-controller pattern
- Blackboard pattern
- Interpreter pattern

Layered pattern

Ce modèle peut être utilisé pour structurer des programmes pouvant être décomposés en groupes de sous-tâches, chacune d'elles étant à un niveau particulier d'abstraction. Chaque couche fournit des services à la couche immédiatement supérieure.

Quelques exemples

Le modèle OSI, le package swing (java), application web (3-tiers).



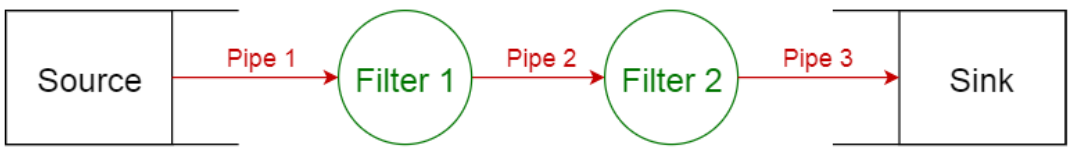
Layered pattern

- Avantages :
  - Faible couplage (restreint aux couches adjacentes) : moindre impact des modifications,
  - Couches facilement interchangeables, e.g. Wifi vs Fibre tout en conservant IP,
  - Cas applications web :
    - L'accès à la base de données n'est pas direct
    - L'interface est changeable facilement sans modifier l'application
    - La couche stockage (ou métier) est partageable entre différentes applications
- Inconvénients :
  - Risque de perte de performances par passage à travers des couches multiples

Pipe and filter pattern

Les sous-systèmes (*filters*) recoivent les données d'entrées et produisent des données de sortie connectées à d'autres sous-systèmes. Les connexions sont gérées par des *pipes*

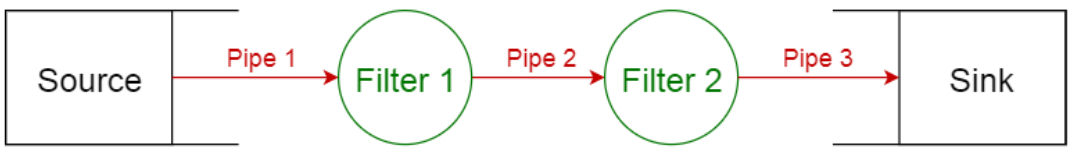
Illustration



Pipe and filter pattern

Les sous-systèmes (*filters*) recoivent les données d'entrées et produisent des données de sortie connectées à d'autres sous-systèmes. Les connexions sont gérées par des *pipes*

Illustration



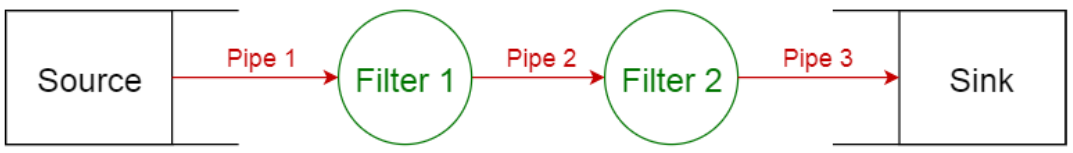
Exemple

Le shell `bash` sous `GNU/Linux`.

Pipe and filter pattern

Les sous-systèmes (*filters*) recoivent les données d'entrées et produisent des données de sortie connectées à d'autres sous-systèmes. Les connexions sont gérées par des *pipes*

Illustration



Exemple

Le shell `bash` sous `GNU/Linux`.

Bonus - Malus

Il facilite la construction de traitements complexes sur des flux de données mais il est limité à des systèmes non interactifs.

Définition

Un *patron de conception* est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.

Définition

Un *patron de conception* est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.

Démonstration avec Inkscape

- Composite,
- Strategy,
- Decorator,
- ...

Liste des 23 patrons de conception

- Les patrons de création décrivent comment régler les problèmes d'instanciation de classes : *Singleton*, *Prototype*, *Fabrique*, *Fabrique abstraite*, *Monteur*

Liste des 23 patrons de conception

- Les patrons de création décrivent comment régler les problèmes d'instanciation de classes : *Singleton*, *Prototype*, *Fabrique*, *Fabrique abstraite*, *Monteur*
- Les patrons de structure décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas : *Pont*, *Façade*, *Adaptateur*, *Objet composite*, *Proxy*, *Poids-mouche*, *Décorateur.*

Liste des 23 patrons de conception

- Les patrons de création décrivent comment régler les problèmes d'instanciation de classes : *Singleton*, *Prototype*, *Fabrique*, *Fabrique abstraite*, *Monteur*
- Les patrons de structure décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas : *Pont*, *Façade*, *Adaptateur*, *Objet composite*, *Proxy*, *Poids-mouche*, *Décorateur.*
- Les patrons de comportement permettent de résoudre les problèmes liés aux comportements, à l'interaction entre les classes : *Chaîne de responsabilité Commande*, *Interpréteur*, *Itérateur*, *Médiateur*, *Mémento*, *Observateur*, *État*, *Stratégie*, *Patron de méthode*, *Visiteur*.

Présentation standardisée

Chaque patron de conception est présentée de la même manière :

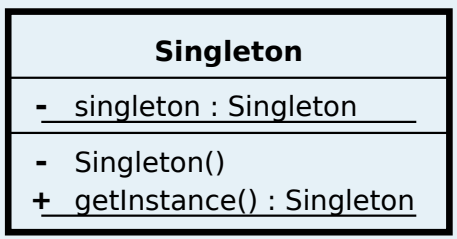
- Nom
- Description du problème à résoudre
- Description de la solution
- Conséquences (avantages / inconvénients)

Exemple de patron de conception : Singleton

Nom : Singleton,

Problème : Restreindre l'instanciation d'une classe à un seul objet,

Solution : .



Préserver la correspondance entre le modèle (UML) et l'implémentation

Le modèle et le code sont rarement strictement identiques car

- certaines notions au niveau modèle sont inexistantes au niveau code (spécificité langage),

Préserver la correspondance entre le modèle (UML) et l'implémentation

Le modèle et le code sont rarement strictement identiques car

- certaines notions au niveau modèle sont inexistantes au niveau code (spécificité langage),
- si le code est écrit/ajusté manuellement, alors on s'expose à un risque d'erreur de correspondance,

Préserver la correspondance entre le modèle (UML) et l'implémentation

Le modèle et le code sont rarement strictement identiques car

- certaines notions au niveau modèle sont inexistantes au niveau code (spécificité langage),
- si le code est écrit/ajusté manuellement, alors on s'expose à un risque d'erreur de correspondance,
- le modèle n'est pas toujours exhaustif en pratique.

Préserver la correspondance entre le modèle (UML) et l'implémentation

Le modèle et le code sont rarement strictement identiques car

- certaines notions au niveau modèle sont inexistantes au niveau code (spécificité langage),
- si le code est écrit/ajusté manuellement, alors on s'expose à un risque d'erreur de correspondance,
- le modèle n'est pas toujours exhaustif en pratique.

Préserver la correspondance entre le modèle (UML) et l'implémentation

Le modèle et le code sont rarement strictement identiques car

- certaines notions au niveau modèle sont inexistantes au niveau code (spécificité langage),
- si le code est écrit/ajusté manuellement, alors on s'expose à un risque d'erreur de correspondance,
- le modèle n'est pas toujours exhaustif en pratique.

Définitions

Modifications du système sans changement des fonctionnalité.

- Modification du modèle : *model transformation*
- Programmation à partir du modèle : *forward engineering*
- Modification du code : *refactorying*
- Modélisation à partir du programme : *reverse engineering*

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),
- privilégier les appels récursifs : code plus lisible,

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),
- privilégier les appels récursifs : code plus lisible,
- présenter clairement (e.g. indentation des blocs de code),

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),
- privilégier les appels récursifs : code plus lisible,
- présenter clairement (e.g. indentation des blocs de code),
- documenter : pour les autres mais aussi pour soi-même,

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),
- privilégier les appels récursifs : code plus lisible,
- présenter clairement (e.g. indentation des blocs de code),
- documenter : pour les autres mais aussi pour soi-même,
  - insérer du pseudo-code

Bonnes pratiques

- Avoir une notation homogène et parlante des variables :
  `PtrMaxFreq` et `PtrMinFreq` : oui !
  `maximumFq` et `FrqyMin` : non !
- penser aux assertions (sous-forme de pre/post conditions),
- utiliser un débogueur,
- structures de contrôle : rester clair et structuré,
  - limiter les tests imbriqués (e.g. `if` englobant récursivement des `if`),
  - bannir les sauts (e.g. `goto` multiples),
- privilégier les appels récursifs : code plus lisible,
- présenter clairement (e.g. indentation des blocs de code),
- documenter : pour les autres mais aussi pour soi-même,
  - insérer du pseudo-code
  - commenter les entrées-sorties, le fonctionnement général de l'algorithme...

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,
  - *architectural pattern* (e.g. n-Tiers),

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,
  - *architectural pattern* (e.g. n-Tiers),
  - *design pattern* (e.g. undo/redo, stategy, ...)

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,
  - *architectural pattern* (e.g. n-Tiers),
  - *design pattern* (e.g. undo/redo, stategy, ...)
- Préserver la correspondance entre le modèle et le code

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,
  - *architectural pattern* (e.g. n-Tiers),
  - *design pattern* (e.g. undo/redo, stategy, ...)
- Préserver la correspondance entre le modèle et le code
- Profiter des mécanismes du langage, .e.g. polymorphisme ...

Synthèse de quelques bonnes pratiques

- Intégrer des fonctionnalités existantes (Composants *off-the-shelf*)
- S'appuyer sur l'expérience :
  - GRASP,
  - *architectural pattern* (e.g. n-Tiers),
  - *design pattern* (e.g. undo/redo, stategy, ...)
- Préserver la correspondance entre le modèle et le code
- Profiter des mécanismes du langage, .e.g. polymorphisme ...
- Présenter correctement son code, i.e. notations, code lisible et structuré, deboggueur, documentation, assertions, exceptions ...

Organisation temporelle

- 9 séances de travaux pratiques où on va décrouvrir quelques patrons de conception,
- 1 évaluation finale pratique,
- 1 évaluation sur papier.

Organisation temporelle

- 9 séances de travaux pratiques où on va décrouvrir quelques patrons de conception,
- 1 évaluation finale pratique,
- 1 évaluation sur papier.

Merci pour votre attention.