Génie Logiciel

Tests et preuves de programmes

Nicolas Delanoue

Université d'Angers - Polytech Angers

Définition

Un bug est un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement.

Code python de l'addition avec un bug

```
def addition(x,y):
    if x==0:
        somme = x
    else:
        somme = x + y
    return somme
```

Exemples classiques de systèmes défaillants

- Automobile (2004) - régulateur de vitesse
- USS Yorktown (1998) - Une division par zéro coupe les moteurs
- Therac-25 (1985-1987) - radiologie et contrôle d'injection de substances radioactives
- London Ambulance System (1992) - central et dispatch ambulances
- Ariane 5 (1996) - Mauvaise réutilisation du code . . .
- SI du FBI (2005) - SI qui n'a pas pu être déployé
- Mars Climate Orbiter (1999) - Plusieurs unités de mesure : kilos vs pounds
- Bourse de Londres (Taurus, 1993) - SI qui n'a pas pu être déployé

Therac-25 (official report)

- The software code was not independently reviewed.
- The software design was not documented with enough detail to support reliability modelling.
- The system documentation did not adequately explain error codes.
- AECL personnel were at first dismissive of complaints.
- The design did not have any hardware interlocks to prevent the electron-beam from operating in its high-energy mode without the target in place.
- Software from older models had been reused without properly considering the hardware differences.
- The software assumed that sensors always worked correctly, since there was no way to verify them.
- Arithmetic overflows could cause the software to bypass safety checks.

Il y a principalement deux manières de réduire les bugs :

1. tester le logiciel afin de chercher des bugs,
2. prouver que le programme est sans bugs.

Remarque

Chacune de ces approches peut être effectuée “manuellement” ou bien avec l’aide d’un assistant (souvent un programme).

Nature des tests : statique ou dynamique

- Test dynamique
  - on exécute le programme avec des valeurs en entrée et on observe le comportement
- Test statique
  - relecture / revue de code
  - analyse automatique (vérification de propriétés, règles de codage, typage ...)

Définition

Une spécification exprime ce qu'on attend du système.

Elle peut être décrite de plusieurs manières :

- un cahier des charges (en langue naturelle)
- commentaires dans le code
- contrats sur les opérations (à la Eiffel)
- un modèle UML
- une spécification formelle (automate, modèle B...)

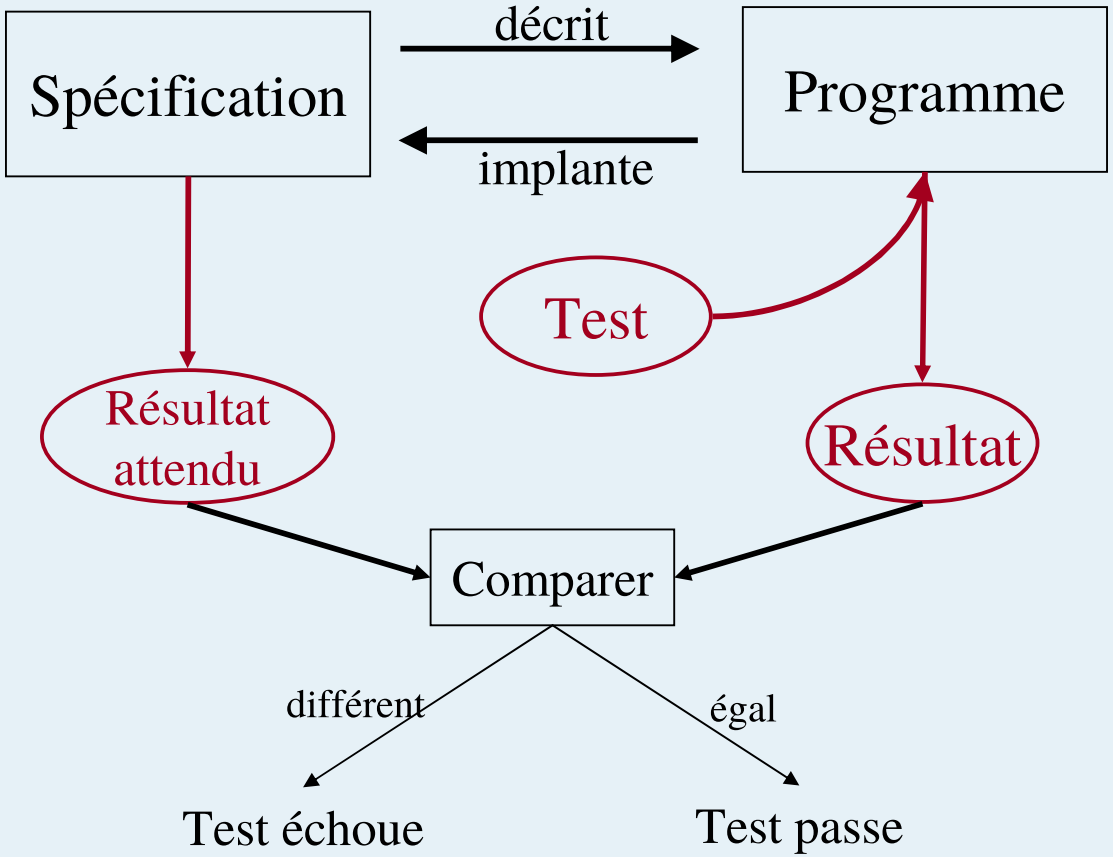


FIGURE – Schéma global de la mise en œuvre de tests.

Qu'est ce qu'on teste ?

- fonctionnalité
- sécurité / intégrité
- utilisabilité
- cohérence
- maintenabilité
- efficacité
- robustesse
- sûreté de fonctionnement

Quand est ce qu'on teste ?

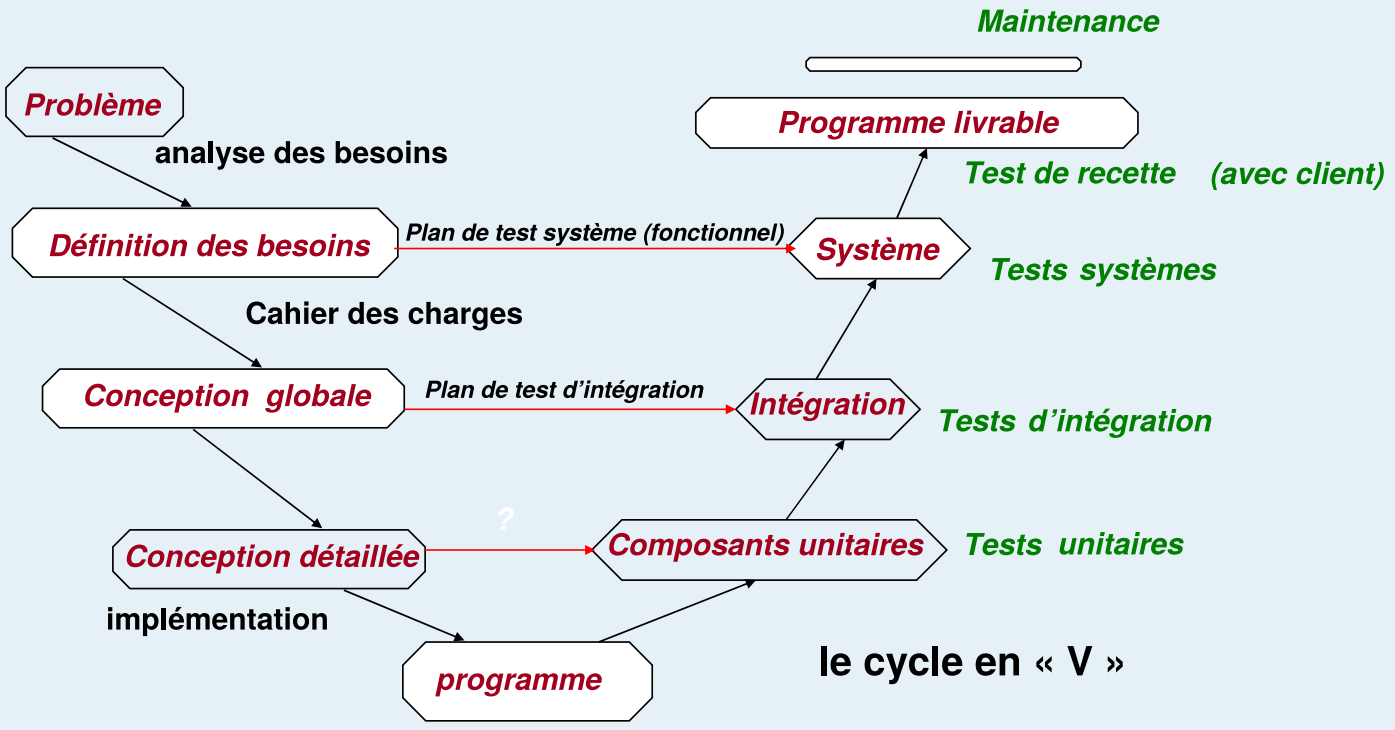


FIGURE – Hiérarchisation des tests

En principe, on teste un peu tout le temps !

Définition

Un *test unitaire* est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un programme.

Remarque

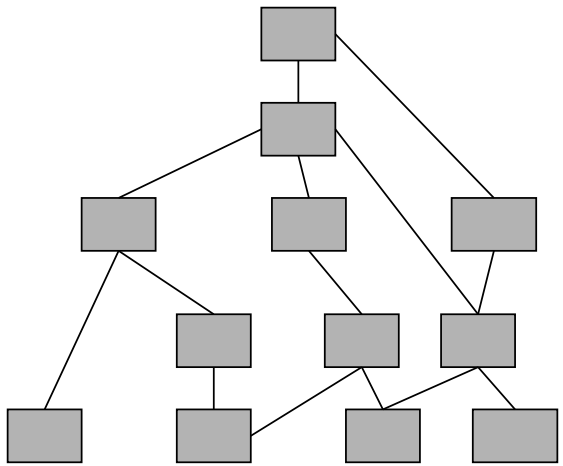
- En programmation procédurale, une unité est une fonction.
- En programmation orientée objet, une unité est une classe.

Exemple

On pourra tester une classe en vérifiant que toutes ces méthodes n'engengrent pas d'erreur d'exécution.

Définition

Un *test d'intégration* est une procédure permettant de vérifier le bon fonctionnement d'un logiciel suite à l'assemblage de modules indépendants.



Difficultés

Construire un ordre permettant de tester graduellement l'assemblage des unités (problème de dépendance).

Définition

Un *test de validation* permet de vérifier si toutes les exigences client, décrites dans le document de spécification du logiciel, sont respectées.

Lors de ces tests, on valide la globalité du système, i.e. les fonctions offertes, souvent à partir de l'interface.

Remarque

Ce genre de tests aussi peut être automatisé.

Définition

Un test de *non régression* consiste à vérifier que des modifications apportées au logiciel n'ont pas introduit de nouvelle erreur.

Instant

Dans la phase de maintenance,

- Après refactoring,
- ajout/suppression de fonctionnalités,
- après la correction d'une faute.

Remarques :

1. Un jeune diplômé sur trois commence par faire du test
2. 50% des start-up échouent à cause du trop grand nombre de bugs :
   - mauvaise campagne de test,
   - maintenance difficile,
   - pas de non régression.

Définition

Le test est un ensemble d'activités utilisé pour tester le code source afin de découvrir (et corriger) les erreurs avant la livraison du logiciel client.

Définitions - 2 types de tests

1. Lors d'un test *fonctionnel*, on se réfère uniquement à l'observation de la sortie pour certaines valeurs d'entrée.
2. Lors d'un test *structurel*, on exploite la connaissance de la structure et la mise en œuvre du logiciel.

Remarque

1. Pour un test fonctionnel, il n'y a aucune tentative d'analyser le code, qui produit la sortie.
2. Tests fonctionnels et structurels sont aussi respectivement qualifiés de test boite noire et test boite blanche.

Code python de l'addition

```
def addition(x,y):
    if y==0:
        somme = x
    else:
        somme = x + y
    return somme
```

Code python de l'addition

```
def addition(x,y):
    if y==0:
        somme = x
    else:
        somme = x + y
    return somme
```

Exemple de test fonctionnel

```
def test_addition():
  x = 4
  y = 5
  valeur_attendue = 9
  if addition(x,y) == valeur_attendue:
     print("test réussi")
  else:
     print("test échoué")
```

Code python pgcd

```
def pgcd(p,q):
  while(p!=q):
   if p>q:
     p=p-q
   else:
     q=q-p
  return p
```

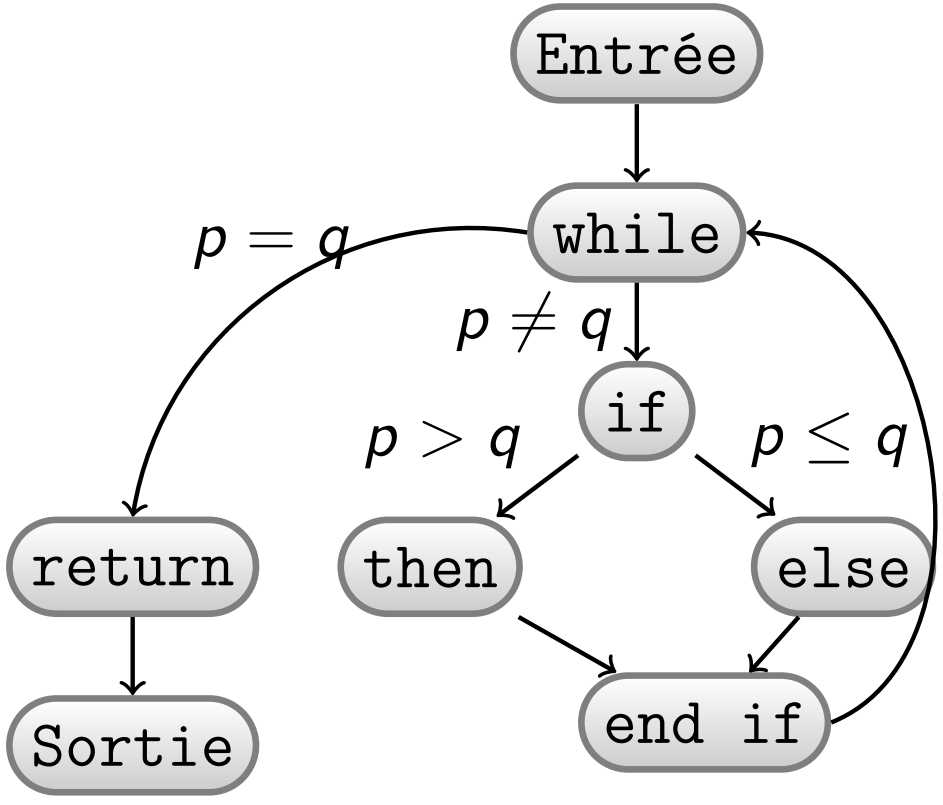


FIGURE – Graphe de contrôle

On cherche des jeux de tests à partir du graphe de contrôle afin de couvrir toutes les instructions, tous les chemins, ...

Niveau de couverture

On définit plusieurs niveaux (objectifs) de couverture

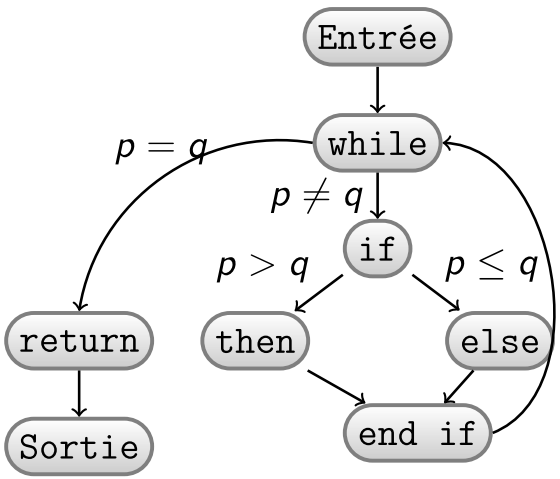
- $C0$: tester toutes les instructions au moins une fois,
- $C1$: tester toutes les décisions possibles au moins une fois,
- Ci : tester toutes les branches ($C1$) mais en passant i fois dans chaque boucle,
- $C\infty$: tester tous les chemins possibles (virtuellement impossible).

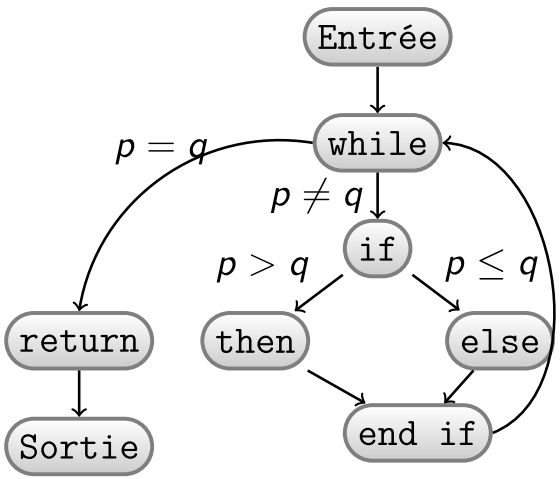
Remarques

- $C0$ et $C1$ sont différents,
- En pratique, le test d'un programme consiste à satisfaire $C0$ et $C1$.

Tous les noeuds (C0) :

- (Entrée, while, if, then, return, Sortie)
- (Entrée, while, if, else, return, Sortie)



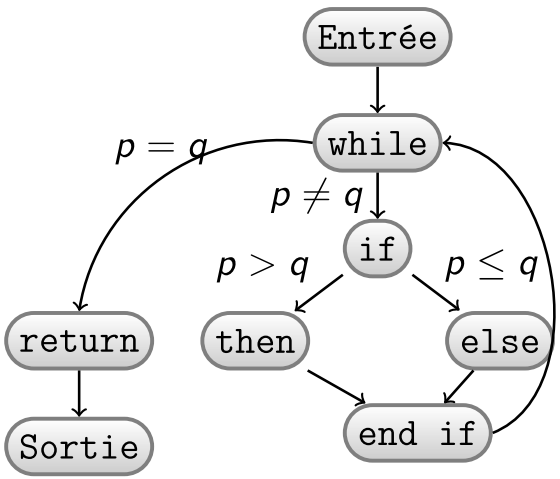


Tous les noeuds (C0) :

- (Entrée, while, if, then, return, Sortie)
- (Entrée, while, if, else, return, Sortie)

Tous les décisions (C1) :

- Chemins de (C0)
- (Entrée, while, return, Sortie)



Tous les noeuds (C0) :

- (Entrée, while, if, then, return, Sortie)
- (Entrée, while, if, else, return, Sortie)

Tous les décisions (C1) :

- Chemins de (C0)
- (Entrée, while, return, Sortie)

Tous les 2-chemins (C-2) :

- Chemins de ($C1$)
- (E, while, if, then, while, if, then,return)
- (E, while, if, then, while, if, else,return)
- (E, while, if, else, while, if, then,return)
- (E, while, if, else, while, if, else,return)

Code python de l'addition avec un bug

```
def addition(x,y):
    if x==0:
        somme = x
    else:
        somme = x + y
    return somme
```

Exemple de détection de bug

Le bug est détectée par exécution du chemin :
(Entrée , if , Sortie)

Théorème de Rice

Toute propriété sémantique non triviale d'un programme est indécidable.

Explications ?

```
def addition(x,y):
    if x==0:
        sum = y
    else:
        sum = x + y
    return sum
```

Si j'ai confiance dans l'instruction $+$, alors on peut prouver que cette fonction réalise bien l'addition.

Preuve

On regarde les deux cas :

- si $x = 0$, alors la somme est bien égale à y, et c'est la valeur retournée sum.
- si $x \neq 0$, alors la somme est bien égale à x+y, et c'est la valeur retournée sum.

Fin de la preuve.

```
def pgcd(p,q):
  while(p!=q):
   if p>q:
     p=p-q
   else:
     q=q-p
  return p
```

Problème de la boucle . . .
Réponse : on invente un invariant I qui dépend de p et q.

Soit $I(p, q)$ le pgcd de p et q.

Preuve par récurrence

- si $p = q$, alors leur pgcd est p, c'est ce que retourne le programme.
- si $p > q$, alors $I(p, q) = I(p - q, q)$.
  En effet, si d est un diviseur de p et q, alors d est un diviseur de $p - q$ et q.
  Effectivement, avec $p = p' \times d$ et $q = q' \times d$ on a
  $p - q = (p' - q') \times d$ et $q = q' \times d$.
- si $q > p$, raisonnement identique.

Donc, le programme fabrique une suite strictement décroissante $\{(p, q)_n\}_{n \in \mathbb{N}}$ et minorée.
Fin de la preuve.

Conclusion

- Les Tests : une discipline à part entière
- Très importants dans le cycle de développement
- D'autres tests (non abordés dans le TP) : tests d'intégration, Fonctionnels, non-régression
- Les tests unitaires :
  - Malheureusement ne couvrent pas tous les cas,
  - Restent néanmoins considérés comme un gage de qualité
  - Même si vous couvrez 99% des cas, le 1% restant peut cacher un bug majeur ! ! !

Le test de programmes peut être une façon très efficace de montrer la présence de bugs mais est désespérément inadéquat pour prouver leur absence - Edsger Dijkstra