Chapitre 11 : Tests

Construction et maintenance de logiciels

Guy Francoeur

basé sur les travaux d'Alexandre Blondin Massé, professeur

29 avril 2019

UQÀM | Département d'informatique

Table des matières

Vue générale

Plusieurs types de tests :

- Tests en **boîte blanche**;
- Tests en **boîte noire**;
- Tests **unitaires**;
- Tests de **régression**;

Cadres de tests

- Un **cadre de tests** est un ensemble de tests **complémentaires**;
- Ils doivent fournir un certain niveau de **confiance** :
  - Bonne **couverture** de branchement;
  - Bonne **couverture** des cas **fréquents**, **moins** fréquents;
- Ils doivent **minimiser la redondance** :
  - Chaque test doit être **pertinent**;
  - Si le **retrait** d'un test ne change pas le niveau de confiance, alors il n'est **pas pertinent**.
  - Pourquoi est-ce problématique s'il y a **trop** de tests?

Tests en boîte blanche - Définition

- Définition : Le terme test de boîte blanche fait référence à un test qui nécessite de connaître le fonctionnement interne du système, donc le code source.
- Les tests de boîte blanche ne permettent généralement pas de découvrir les fonctionnalités manquantes du système.

Tests en boîte blanche

- Les cas **peu fréquents** sont plus à risque, car le code est **moins souvent parcouru**;
- Aussi, certaines portions de code peuvent être parcourues **sans qu'on s'y attende**;
- Il faut définir un **cadre de tests** qui vérifie autant les cas **usuels** que les cas **limites** et les cas **peu fréquents**;
- Cette étude nécessite de vérifier les **structures internes** des modules;
- Elle dépend directement du code utilisé dans l'**implémentation**;

Tests en boîte noire

- Ils sont **complémentaires** aux tests en **boîte blanche**;
- Basée sur les **spécifications fonctionnelles** du module;
- Tests utilisant **des valeurs aléatoires**?
- On peut aussi tester en **partitionnant les domaines** de valeurs en **classes d'équivalence** :
  - Tests des **limites** de la classe;
  - Test avec une valeur **représentative** de la classe.
- Une bonne **couverture** :
  - Tester les **cas typiques**;
  - Ne pas oublier les **valeurs limites**.

Tests unitaire

- CUnit est un outil intéressant pour construire des tests unitaire;
- Permet de faire correspondre un résultat à une valeur en entré;
- Il est donc possible de statuer sur le succès ou l'échec;
- Si une fonctionnalité brise le test il faudra agir;
  - Réécrire le test;
  - Modifier le code pour le rendre conforme;

Tests de regression

- Les tests de regression existe afin de savoir si nous avons brisé des fonctionnalités;
- Est utile pour répondre à la question : Est-ce que tout fonctionne comme avant?
- make et le Makefile sont surement une façon rapide de vérifier automatiquement si tout fonctionne comme avant.

Table des matières

Niveau de couverture

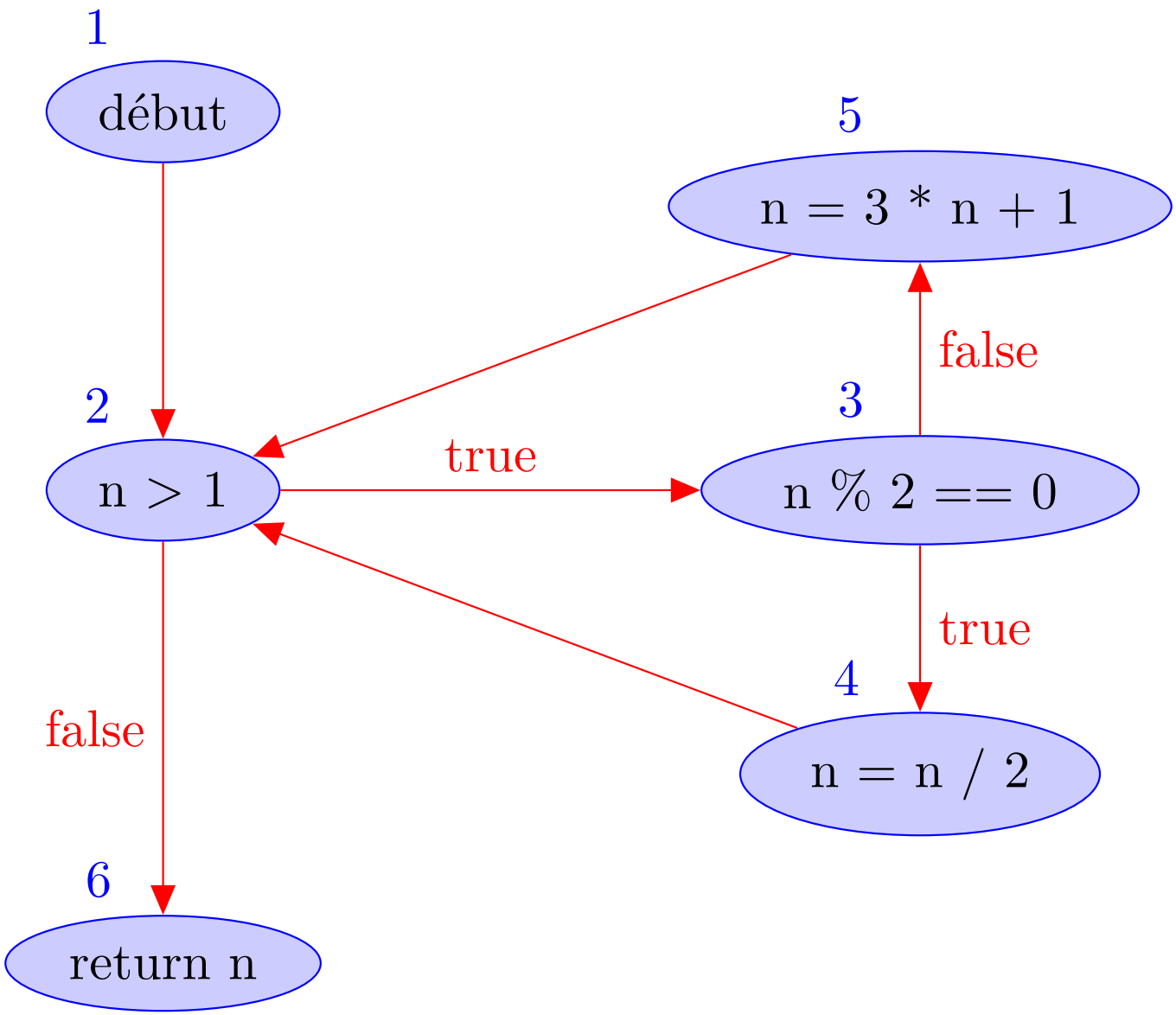
- Couverture des **instructions** : chaque ligne de code est parcourue au moins une fois; En quoi est-ce **insuffisant** ?
- Couverture des **branchements** :
  - Garantit la couverture des **instructions**;
  - On essaie de vérifier tous les **enchaînements possibles**;
  - Impossible en cas de boucle.
- L'étude du **graphe de flux** permet d'assurer une telle couverture.

Graphe de flux (1/2)

- Reprenons l'exemple de la fonction syracuse :

```
1  unsigned int syracuse(unsigned int n) {
2      while (n > 1) {
3          if (n % 2 == 0)
4              n = n / 2;
5          else
6              n = 3 * n + 1;
7      }
8      return n;
9  }
```

Graphe de flux (2/2)



Chemins indépendants (1/3)

- On **numérote** les sommets;
- La complexité **cyclomatique** est 3;
- On cherche un ensemble de 3 **chemins indépendants** (non **redondants**);
- Ils doivent tous **commencer** par 1 et **terminer** par 6 :

$$(1,2,6), \quad (1,2,3,4,2,6), \quad (1,2,3,5,2,6),$$

$$(1,2,3,5,2,3,4,2,6), \quad (1,2)+(3,4,2)^5+(6), \ldots$$

- Il y en a une **infinité**!
- Or, certains chemins sont **obtenus** à partir d'autres :

$$(1,2,3,5,2,3,4,2,6) = (1,2,3,4,2,6)+(1,2,3,5,2,6).$$

Chemins indépendants (2/3)

- L'ensemble de chemins

$$\{(1,2,6), \quad (1,2,3,4,2,6), \quad (1,2,3,5,2,6)\}$$

est donc **suffisant**.

- Ensuite, on cherche les valeurs de n dans

```
unsigned int syracuse(unsigned int n);
```

qui réalisent chacun des chemins.

- $(1,2,6)$: $n = 1$;
- $(1,2,3,4,2,6)$: $n = 2$;
- $(1,2,3,5,2,6)$: n'existe pas!

Chemins indépendants (3/3)

- Ainsi, le chemin $(1, 2, 3, 5, 2, 6)$ n'est **pas réalisable**.
- On peut construire un **3e chemin réalisable** facilement : n'importe quel chemin qui passe par le sommet 5;
- Par exemple, si $n = 5$, alors on obtient le chemin

$$(1, 2) + (3, 5, 2) + (3, 4, 2)^3 + (6).$$

- Ce chemin est **indépendant** des deux autres.
- Ainsi, les valeurs $n = 1$, $n = 2$ et $n = 6$ forment un **cadre de tests non redondant** qui recouvre bien les **branchements**.

Table des matières

Valeurs limites typiques

- Pour les **chaînes de caractères** : chaîne **vide**, de **longueur un**, et de taille **maximale**;
- Pour les **tableaux** et les **vecteurs**, même chose : de taille **zéro** et de taille **maximale**;
- Pour les valeurs définies par **énumération**, vérifier avec la **première** valeur et la **dernière**;
- En prenant en compte
  - les tests **structurels** (chemins indépendants),
  - les tests en **partitionnant** en classe et
  - les **préconditions** et les **postconditions**,

  vous devriez obtenir un cadre de tests **relativement robuste**.

Exemple

- La fonction `int factorielle (int n);`
- Si entiers sont codés sur d **octets**, alors on peut partionner en **trois classes** de valeurs :
  - Les valeurs **strictement négatives**;
  - Les valeurs **positives** (incluant zéro) telles que le **résultat** est codable sur d **octets**;
  - Les valeurs telles que le **résultat** dépasse d **octets**.
- Si $d = 4$, alors il y a débordement pour $n \geq 13$:

$$\begin{aligned} n! &= 6227020800 \\ 2^{32} &= 4294967296 \end{aligned}$$

- Dans le cas $d = 4$, des valeurs de tests seraient $-5, 0, 1, 8, 12, 13, 28$.