# Qualité logicielle
# Que peut-on mesurer ?

Clémentine Nebut

Faculté Des Sciences Université de Montpellier

2021

# Qualité du logiciel

Norme ISO/IEC 25010 :2011

- la capacité fonctionnelle : capacité du logiciel à satisfaire aux exigences et besoins des utilisateurs.
- la facilité d'utilisation/ utilisabilité, qui porte sur l'effort nécessaire pour apprendre à manipuler le logiciel.
- la fiabilité : capacité d'un logiciel de rendre des résultats corrects quelles que soient les conditions d'exploitation.
- la performance : rapport entre la quantité de ressources utilisées (moyens matériels, temps, personnel), et la quantité de résultats délivrés.
- la maintenabilité, qui mesure l'effort nécessaire à corriger ou transformer le logiciel.
- la portabilité, c'est-à-dire l'aptitude d'un logiciel à fonctionner dans un environnement matériel ou logiciel différent de son environnement initial (yc facilité d'installation et de configuration dans le nouvel environnement).

# Indicateurs de qualité

Dans ce cours nous nous intéressons à 2 types d'indicateurs :

- les indicateurs de qualité des tests
- les indicateurs de qualité du cours source

# Sommaire

# Les tests peuvent-ils mesurer la qualité du code ?

- non ...
- Ne font qu'augmenter la confiance en le logiciel
- Des tests nombreux ne sont pas garants de qualité
- Les tests peuvent être mauvais, et ne pas détecter les erreurs potentielles.
- quels indicateurs de qualité pour les tests ?

# Quels indicateurs de qualité pour les tests ? La couverture

- Une mesure classique de qualité des tests est l'analyse de la couverture du code qu'ils couvrent.
- Plusieurs mesure de couverture peuvent être utilisées
- La plupart des mesures se basent sur une représentation sous forme de graphe de flot de contrôle ou de données des programmes testés.

# Graphe de flot de contrôle

C'est un graphe orienté et connexe $(N, A, e, s)$ :

- $N$ : ensemble de sommets
  - sommet = bloc d'instructions exécutées en séquence
- E : relation $N \times N$
  - débranchements possibles du flot de contrôle
- e : sommet d'entrée du programme
- s : sommet de sortie du programme

# Graphe de flot de contrôle : arcs

Un arc entre deux noeuds n1 et n2 représente le transfert du contrôle de n1 à n2 :

- avec étiquetage par une condition booléenne c : transfert si c est vraie ;
- sans étiquetage : transfert inconditionnel.

# Graphe de flot de contrôle : nœuds

Les noeuds représentent soit :

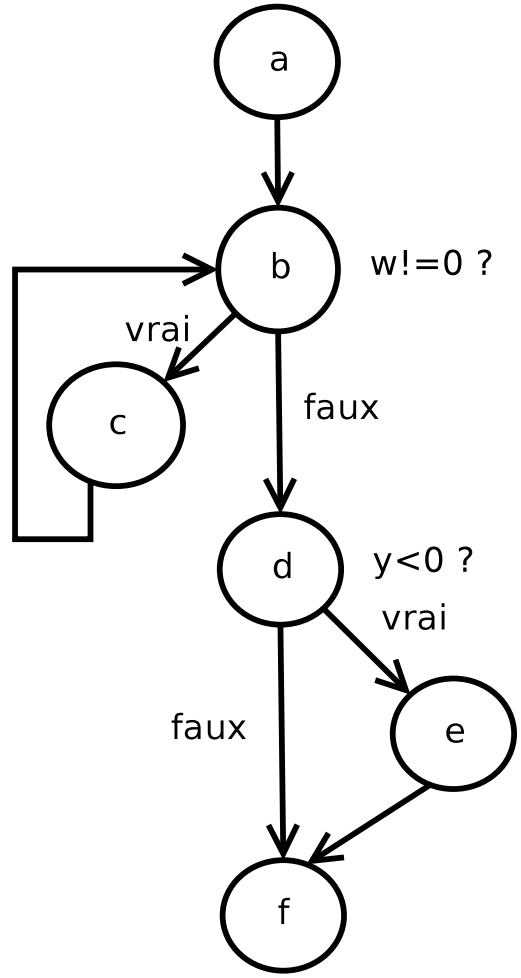
- un bloc d'instructions séquentielles - graphiquement un rectangle ;
- un prédicat (nœud de décision) qui permet le transfert du contrôle - graphiquement un losange.

Deux noeuds particuliers :

- un unique nœud d'entrée (sans prédécesseur)
- un unique nœud de sortie (sans successeur)

# Graphe de flot de contrôle : exemple

```
1   double P(short x, short y){
2
3       short w=abs(y);
4
5       double z=1.0;
6
7       while (w!=0){
8
9           z=z{*}x;
10
11          w=w-1;
12
13      }
14
15      if (y<0)
16
17      z=1.0/z;
18
19      return (z);
20
21  }
```



# Couverture(s)

- couverture des noeuds $\rightarrow$ couverture des instructions exécutables
- couverture des branches
- couverture des 1-chemins, des k-chemins $\rightarrow$ explosion combinatoire
- d'autres mesures e couvertures basées sur un graphe de flot de données (GFC étiqueté par les manipulations de données

# La couverture est-elle un bon indicateur de qualité des tests ?

- non ...
- la couverture des instructions ne suffit pas !
- celle des branches est meilleure mais non satisfaisante
- Des tests couvrant toutes les instructions mais sans aucune assertion ne détectent rien par exemple !

# Quels indicateurs de qualité pour les tests ? Analyse par mutation / mutation testing

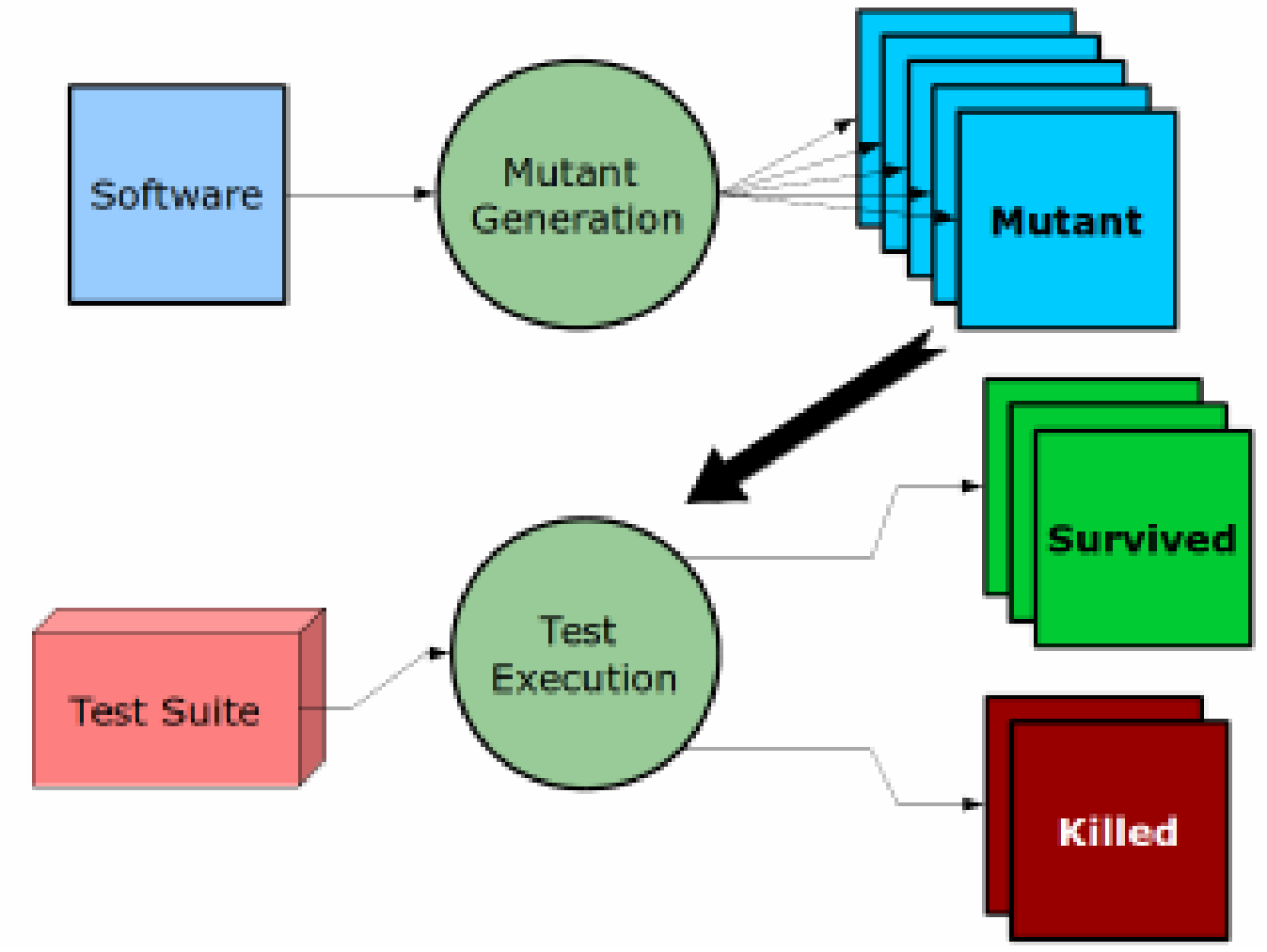
## Qu'est ce qu'un bon test ?

- Un bon test détecte des erreurs s'il y en a
- Un mauvais test ne détecte pas d'erreurs quand il y en a
- Si un test ne détecte rien, comment savoir s'il est bon (et qu'il n'y a pas d'erreur) ou mauvais ?

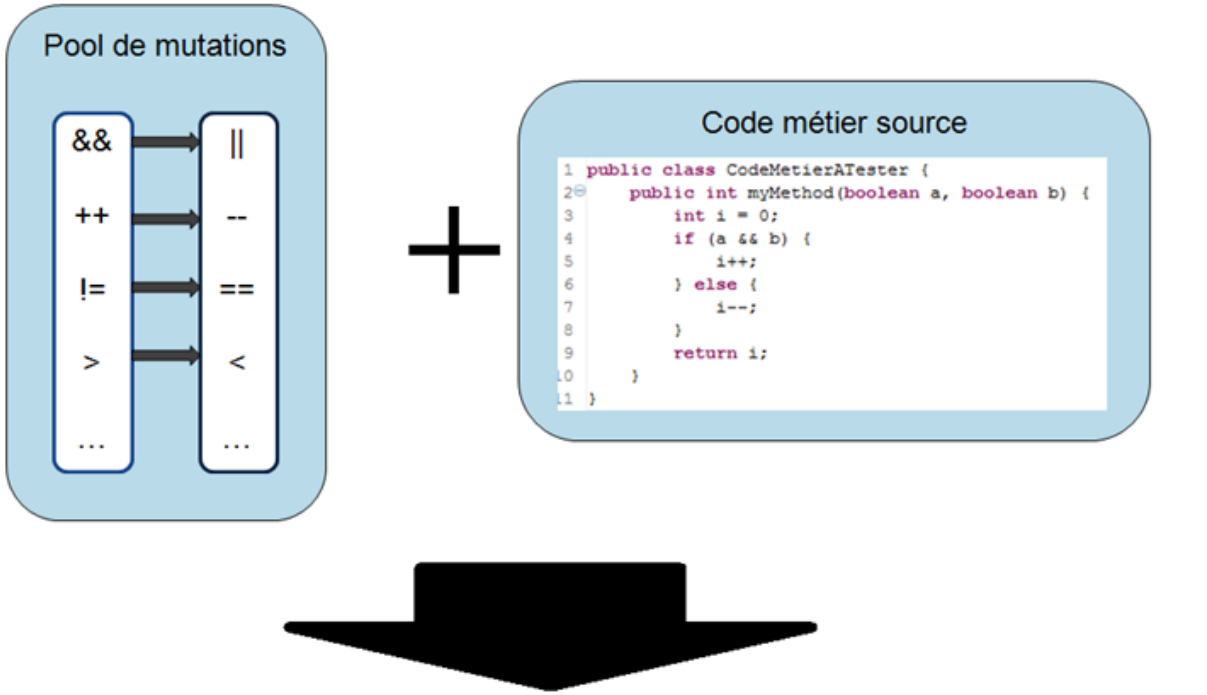
## Mutation testing : principe de base

- On introduit volontairement des erreurs dans le SUT
- On regarde si les tests les détectent

# Mutation testing



# Mutation


Pool de mutations
&&
++
!=
>
...
||
--
==
<
...
Code métier source
1 public class CodeMetierATester {
2 public int myMethod(boolean a, boolean b) {
3 int i = 0;
4 if (a && b) {
5 i++;
6 } else {
7 i--;
8 }
9 return i;
10 }
11 }
Mutant 1
1 public class CodeMetierATester {
2 public int myMethod(boolean a, boolean b) {
3 int i = 0;
4 if (a || b) {
5 i++;
6 } else {
7 i--;
8 }
9 return i;
10 }
11 }
Mutant 2
1 public class CodeMetierATester {
2 public int myMethod(boolean a, boolean b) {
3 int i = 0;
4 if (a && b) {
5 i--;
6 } else {
7 i--;
8 }
9 return i;
10 }
11 }

# Les mutants

- Simulent des fautes classiques
- Modifient le comportement du programme donc y insèrent des fautes
- Les tests doivent trouver la faute : exécution des jeux de test sur le mutant

## Mutation testing

- But du test : Tuer les mutants
- Score = nb mutants tués / nb mutants total
- Analyse des mutants survivants

# Les mutants et les opérateurs de mutation

- Des mutants non viables
- Différents opérateurs de mutation
- Exemple d'opérateur de mutation :
  - changer un $<$ en $<=$
  - changer un $+$ en -
  - altérer les valeurs (changer $a = 1$; en $a = 2$)
  - changer un && en $||$

# Les mutants survivants

- Tests incomplets : il manque un test/une assertion permettant de détecter cette faute
- Mutant équivalent : la mutation opérée a produit un programme SUT équivalent au SUT
  - Certains mutants équivalents peuvent être détectés automatiquement
  - Mais dans le cas général indécidable
- Mutants survivants car pas de couverture

# Mutation analysis

- Il faut beaucoup de mutants !
- L'analyse par mutation est longue ...
  - temps de génération des mutants
  - temps d'exécution de tous les jeux de tests sur chaque mutant
  - (temps de synthèse des résultats)
- Processus trop lent pour guider l'élaboration des tests
- malgré de nombreuses optimisations possibles
- à utiliser pour améliorer une suite de tests pré-existante
- le taux de mutation initial est alors en général déjà élevé

# Sommaire

# Des indicateurs "classiques" de complexité/qualité

- Nombre total de lignes de codes
- Nombre de lignes de codes par classe
- Nombre de méthodes par classe
- Nombre total de méthodes
- Ratio lignes de codes/nombre de méthodes
- Ratio lignes de codes/nombre de classes
- Ratio lignes de commentaires/lignes de codes

Qu'en déduire ? ? ?

# Indice de spécialisation

$$\frac{NORM \times DIT}{NoM}$$

Avec :

- NORM : Number of Overriden Methods, le nombre de méthodes redéfinies.
- DIT : Depth in Inheritance Tree, la profondeur dans l'arbre d'héritage.
- NoM : Number Of Methods, le nombre de méthodes de la classe.

Il est ensuite possible d'effectuer des moyennes pour un paquetage, un ensemble de paquetages, ou un projet. $\rightarrow$ mais attention aux moyennes qui "lissent"...

- Augmente avec le nombre de méthodes redéfinies ou la profondeur d'héritage
- Diminue quand le nombre de méthodes spécifiques à la classe augmente ou quand le nombre de méthodes redéfinies diminue
- Trop grand : la classe redéfinit trop de méthodes dont elle hérite

# Beaucoup d'autres métriques

- que nous ne regarderons pas ici ...
- par exemple : couplage, instabilité d'un package, niveau d'abstraction, complexité cyclomatique, ...
- il ne s'agit QUE d'indicateurs, dont la valeur peut être normale dans un certain contexte même si trop élevée/basse
- Sont donc à analyser avec précautions.
- Certains indicateurs sont liés/inversement liés

# Calcul des indicateurs

- par analyse statique du code
- cette analyse est plus ou moins complexe selon les métriques à calculer
- les indicateurs sont calculés par des outils d'analyse

# Sommaire

# Code smells

- Code smells : soupçons de mauvaises pratiques de conception logicielle, pouvant conduire à des défauts (comme des problèmes de maintenabilité/évolutivité)
- Ne sont pas nécessairement symptômes d'erreurs
- Peuvent en général se détecter par des combinaisons de métriques

# Bad smells, exemple

- Feature envy (jalousie de caractéristique) : quand une méthode (ou plusieurs) d'une classe est plus intéressée par les membres (attributs et méthodes) d'une autre classe que les siens
- peut être détecté à l'aide des indicateurs :
  - LAA (Locality of Attribute Accesses) : représente la proportion de méthodes de la classe accédant à plus d'attributs de classes extérieures qu'à ses propres attributs.
  - FDP (Foreign Data Providers) : représente la proportion d'utilisation d'attributs « extérieurs » et utilisés dans très peu d'autres classes.
  - ATFD (Access to Foreign Data) qui représente la proportion de méthodes de la classe qui accèdent aux attributs directement ou via des accesseurs d'autres classes extérieures.

# Bad smells et outils

- les code smells détectées dans les outils sont souvent d'un autre niveau
- exemples de code smell détecté dans SonarQube
  - Unnecessary imports should be removed
  - The members of an interface or class declaration should appear in a pre-defined order
  - "==" and " !=" should not be used when "equals" is overridden

# Sommaire

# Que faire quand on détecte un potentiel défaut par des indicateurs ?

- On analyse ...
- On peut en déduire qu'il n'y a pas de problème $\rightarrow$ rien à faire
- On peut en déduire qu'il y a un défaut $\rightarrow$ on effectue un refactoring
- Refactoring (réusinage) : opération consistant à retravailler le code source d'un programme informatique
  - pour obtenir un code fonctionnellement équivalent
  - mais avec une lisibilité améliorée (ou d'autres caractéristiques extra-fonctionnelles améiorées)
- peut nécessiter une co-évolution des tests ...

# Refactoring, exemple

- Suite à détection d'une feature envy
- Déplacer une (partie de) méthode d'une classe dans une autre si elle utilise plus les données de celle-ci que les siennes.
- Suite à une détection de Blob
- Séparation du blob en plusieurs classes, en suivant une séparation des préoccupations et hiérarchiser au besoin

# Sommaire

## Indicateurs de qualité des tests

- Critères de couverture du code source par les tests
- analyse par mutation

## Indicateurs de complexité/qualité

- indicateurs nombreux, éventuellement interdépendants
- se calculent par analyse statique du code
- à analyser avec précaution
- se méfier des moyennes d'indicateurs sur un projet qui cachent les valeurs remarquables