



08

Chapitre

Test logiciel

1I2AC1 : Génie logiciel et Conception orientée objet

Régis Clouard, ENSICAEN - GREYC

« Durant le débogage, les novices insèrent du code correctif,
alors que les experts suppriment du code défectueux. »

Richard Pattis

Objectif du chapitre

- Initiation aux tests logiciels et en particulier aux tests dynamiques

Plan du chapitre

1
Généralités
sur les tests

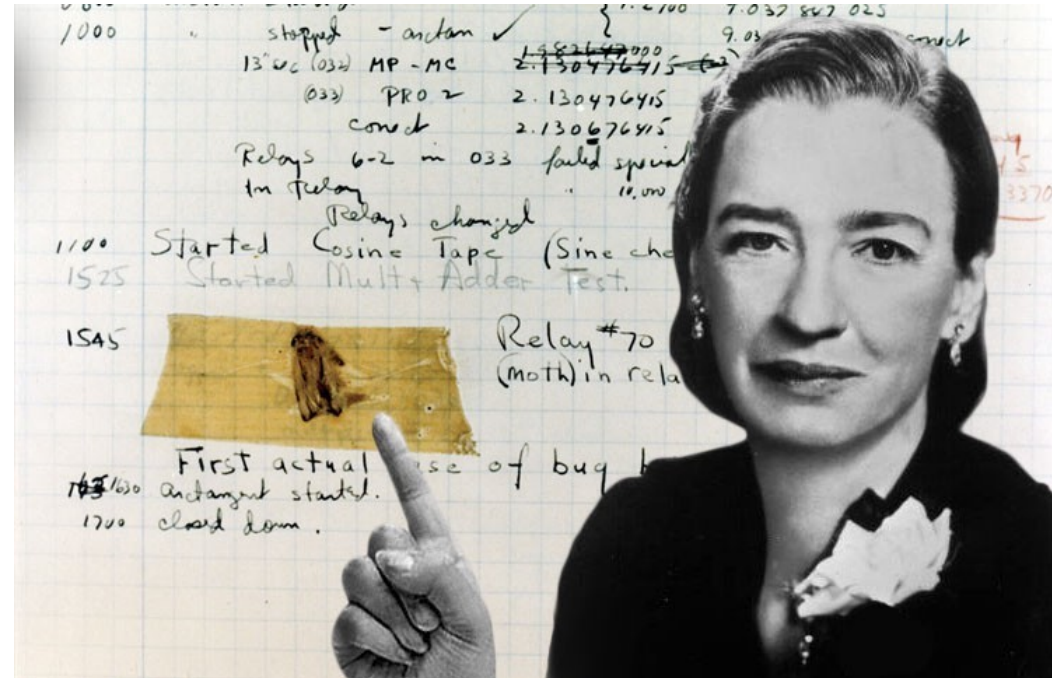
Pourquoi tester ?

4

- 1) Chasser les bugs
- 2) Prémunir contre la régression logicielle

Bug

- Mot popularisé en 1947 par **Grace Hopper** pionnière de l'informatique
- Catégories de bug :
 - Bohrbug
 - Heisenbug
 - Mandelbug
 - Schroedinbug



Grace Hopper
Pionnière de l'informatique

La chasse aux bugs

- Les bugs sont inhérents à l'activité de codage
 - Le code zéro bug n'existe pas (sauf exceptions)
 - Rappel de l'estimation : 1 à 10 bugs / KLOC
- Mais il est possible de détecter certains bugs en testant les programmes pour limiter la casse

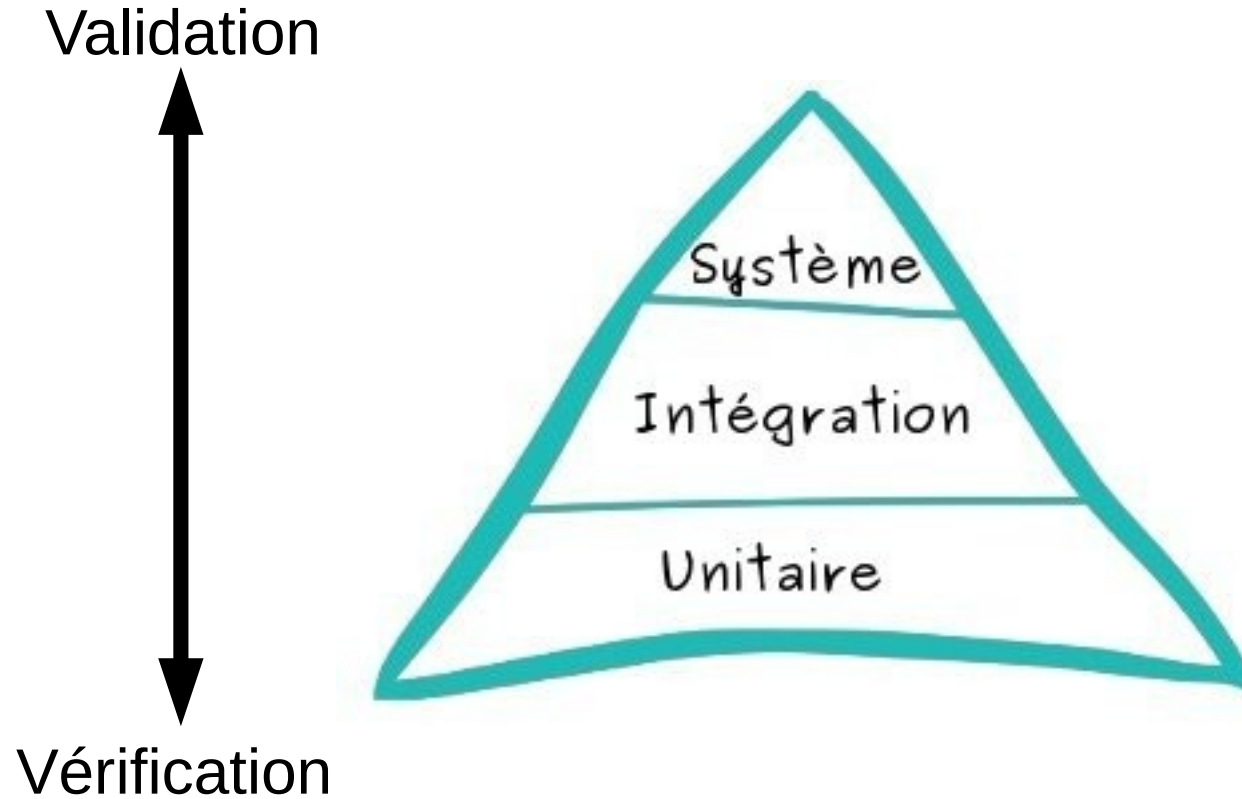
La régression logicielle

- C'est un bug logiciel qui fait qu'une fonctionnalité cesse de fonctionner après une modification du code (refonte ou maintenance)

Qu'est ce qu'un test ?

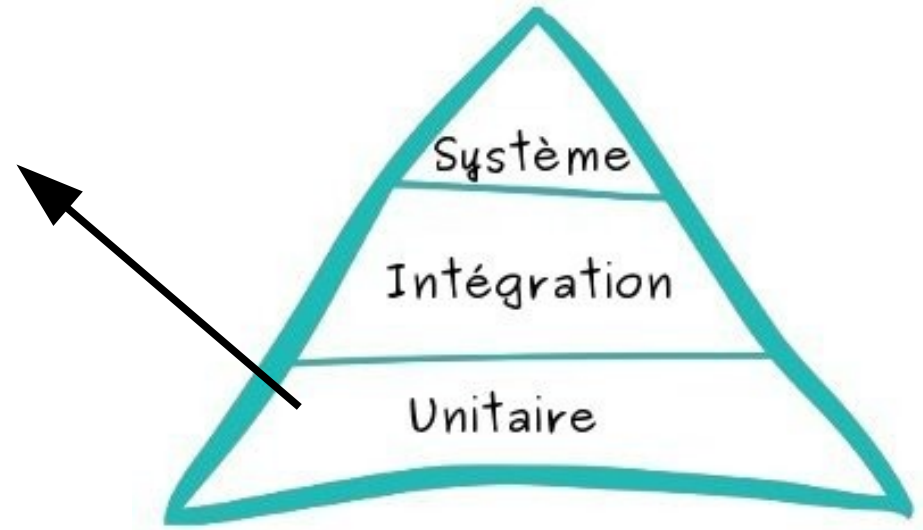
- Procédé de Vérification & Validation
 - Vérification : **le logiciel fonctionne-t-il correctement ?**
 - Validation : **a-t-on construit le bon logiciel ?**

Pyramide des tests



Niveaux de test

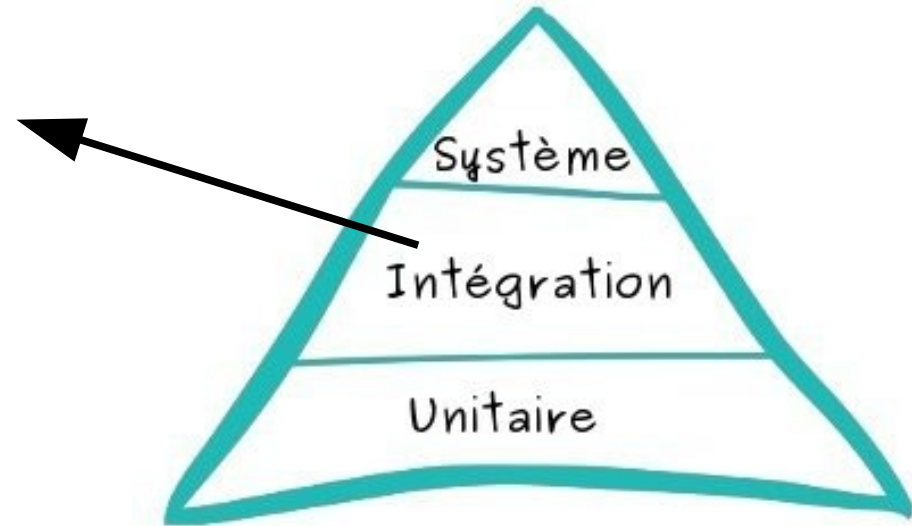
- **Test unitaire** : guider le développeur
 - Erreurs recherchées :
 - ▶ erreurs de codage
 - ▶ erreurs fonctionnelles



Proportion 80 - 90 %

Niveaux de test

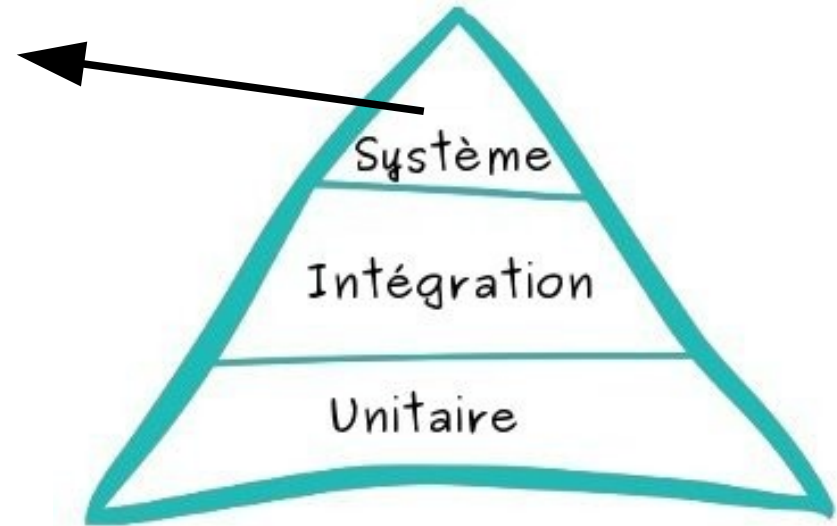
- **Test d'intégration** : guider l'équipe de développement
 - Erreurs recherchées :
 - ▶ erreurs d'interface entre unités



Proportion 5 - 15 %

Niveaux de test

- **Test système** : critiquer le produit
 - Erreurs recherchées :
 - ▶ absence de fonctionnalités
 - ▶ fonctionnalités mal mises en œuvre



Proportion 1 - 5 %

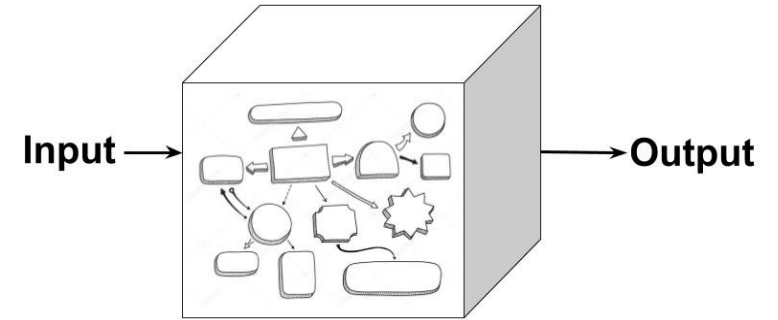
Le test dynamique

- Dans ce qui suit, nous nous focalisons sur les **tests dynamiques**
 - Le test dynamique est un bout de code qui est exécuté avec l'intention de vérifier ou valider un code

Types de test dynamique

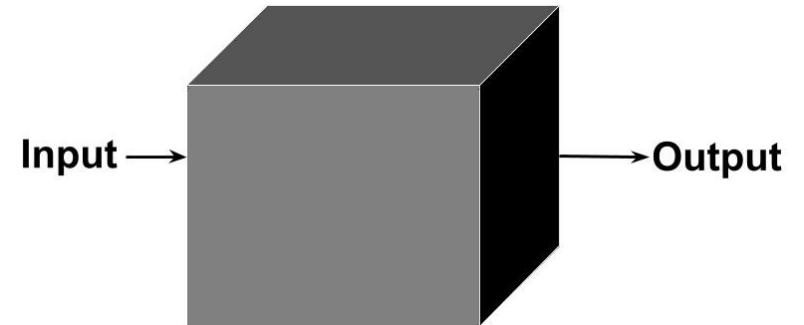
- Test **boîte blanche** (*white-box testing*)

- Tient compte de la structure interne de l'unité testée



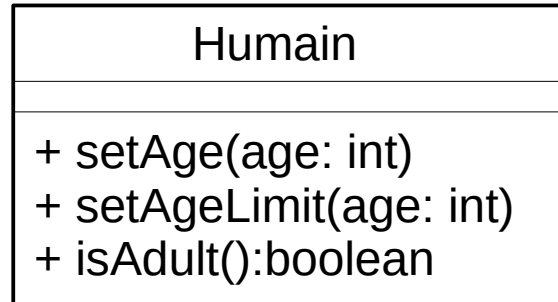
- Test **boîte noire** (*black-box testing*)

- Ne connaît pas la structure interne de l'unité testée



Exemple d'un test

- On suppose une classe Human :



- Dans le détail
 - `void setAge(int age)`
 - `void setAgeLimit(int age)`
 - `boolean isAdult()` throws Exception qui:
 - ▶ lève une exception si `age ∉ [0, ageLimit]`
 - ▶ retourne `true` si `age ∈ [18, ageLimit]`
 - ▶ retourne `false` si `age ∈ [0, 18[`

Cas de test

- L'ensemble du test exécutable s'appelle un cas de test :

```
Human h = new Human();           // fixture
h.setAgeLimit(150);               // donnée de test
try {
    h.setAge(35);                  // donnée de test
    if (h.isAdult()) {            // oracle
        System.out.println("test réussi"); // verdict
    } else {
        System.out.println("test echoue");
    }
} catch (Exception e) {
    System.out.println("erreur, test échoué");
}
```


Quand s'arrêter de tester ?

17

- Les tests sont incomplets par nature
 - On a testé `isAdult()` pour une valeur d'entrée seulement
 - Il faudrait faire d'autres cas de test (eg, limites)
 - Mais, on ne peut pas tester toutes les valeurs !
- Savoir quand s'arrêter est une question d'expérience
 - Un indice, la **couverture** de code

Couverture de code (code coverage)

- La couverture de code est une mesure qui permet d'identifier la proportion du code testé

The screenshot displays an IDE window for a project named 'spring-petclinic'. The main editor shows the code for `PetValidator.java`, with a tooltip indicating 'Hits: 4' for a specific line. The project explorer on the left shows the directory structure and coverage statistics for various classes. The coverage table on the right provides a detailed view of the coverage for all classes in the project.

Element	Class, %	Method, %	Line, %
aj			
antlr			
apple			
ch			
com			
db	100% (0/0)	100% (0/0)	100% (0/0)
images			
io			
java			
javafx			
javassist			
javax			
jdk			
messages	100% (0/0)	100% (0/0)	100% (0/0)
META-INF	100% (0/0)	100% (0/0)	100% (0/0)
net			
netscape			
oracle			
org	100% (20...	93% (78/...	93% (22...
OSGI-INF			
resources			
static	100% (0/0)	100% (0/0)	100% (0/0)
sun			
templates	100% (0/0)	100% (0/0)	100% (0/0)
toolbarButt...			

Limite des tests

- Le test ne certifie pas le code :

« Le test de programme peut être utilisé pour prouver la présence de bugs, mais jamais leur absence »

Edsger Dijkstra

- Parfois le code source est faux, mais le code de test aussi !
- Question : Doit-on faire des tests de tests ?

Plan du chapitre

1

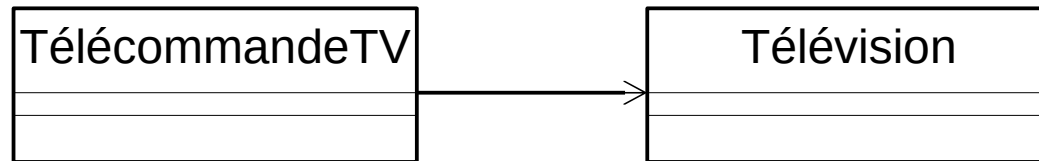
Généralités
sur les tests

2

Tests
unitaires

Test unitaire

- Procédé de vérification d'une unité seulement
 - ie les méthodes d'une **classe** seulement en **Java**
- Par exemple:
 - On teste les méthodes publiques et protégées de la classe `TelecommandeTV` en isolation
 - Si on teste la classe `TelecommandeTV` en utilisant une `Television` dans les tests, c'est du **test d'intégration**



Qualité d'un test unitaire : FIRST

- **[F]**ast (Rapide)
- **[I]**solated (Isolé)
- **[R]**epeatable (Répétable)
- **[S]**elf-validating (Auto-évaluable)
- **[T]**imely (Juste à temps)

Plan du chapitre

23

1

Généralités
sur les tests

2

Tests
unitaires

3

Code testable

Testabilité d'un code

- On n'écrit pas du code testable comme du code classique

Testabilité d'un code

- Exemple : Soit la méthode suivante qui déclenche une alarme d'un agenda à une heure butoir donnée

```
public final class Agenda {  
    ...  
    public void check() {  
        if (System.currentTimeMillis() > 100) {  
            new Bell().ring();  
        }  
    }  
}
```

- Ce simple bout de code est considéré comme mauvais au sens du génie logiciel
- Voyez-vous pourquoi ce code est mauvais ?

Testabilité d'un code

- Mauvais parce que non testable :
 - 1) Le jeu de test est incontrôlable - Pas FI(R)S (repeatable)

```
public final class Agenda {  
    ...  
    public void check() {  
        if (System.currentTimeMillis() > 100) {  
            new Bell().ring();  
        }  
    }  
}
```

Testabilité d'un code

- Mauvais parce que non testable :
 - 1) L'oracle est invérifiable - Pas FIR(S)T (self-validating)

```
public final class Agenda {  
    ...  
    public void check() {  
        if (System.currentTimeMillis() > 100) {  
            new Bell().ring();  
        }  
    }  
}
```

Code testable

- Solution : Encapsuler et externaliser les dépendances pour rendre un code testable :


```
public final class Agenda {
    private Clock _clock;
    private Bell _bell;

    public Agenda( ) { // Par défaut
        _clock = System; _bell = new Bell(); _limit = 100;
    }
    public Agenda( int limit, Clock clock, Bell bell ) { // Pour les tests
        _clock = clock; _bell = bell; _limit = limit;
    }
    public void check() {
        if (_clock.getTimeInMillis() > _limit) {
            _bell.ring();
        }
    }
}
```

Code testable (2)

- Décomposer une méthode en 2 pour la rendre testable :

```
double getActualRouteLengthInMeters() {
    List<GeodesicPoint> routePoints = _database.getRoute();
    if (routePoints.isEmpty()) { return 0; }
    double distance = 0;
    for (int i = 1; i < routePoints.size(); i++) {
        distance += routePoints.get(i - 1).distanceToInMeters(routePoints.get(i));
    }
    return distance;
}
```



```
double getActualRouteLengthInMeters() {
    return _getActualRouteLengthInMeters(_database.getRoute());
}
static double _getActualRouteLengthInMeters( String gpsRoute ) {
    if (routePoints.isEmpty()) { return 0; }
    double distance = 0;
    for (int i = 1; i < routePoints.size(); i++) {
        distance += routePoints.get(i - 1).distanceToInMeters(routePoints.get(i));
    }
    return distance;
}
```

Plan du chapitre

30

1

Généralités
sur les tests

2

Tests
unitaires

3

Code testable

4

Frameworks de test
JUnit
Mockito

Automatisation du verdict

- Pour l'instant :

```
Human h = new Human();           // fixture
h.setAgeLimit(150);
try {
    h.setAge(35);                 // donnée de test
    if (h.isAdult()) {           // oracle
        System.out.println("test passe"); // verdict
    } else {
        System.out.println("test echoue");
    }
} catch (Exception e) {
    System.out.println("erreur, test inconclusif");
}
```

1/ Le framework JUnit

- Permet de faciliter l'écriture de test pour le langage Java.
- Intégré à IntelliJ.
- Ce qu'il offre :
 - Des assertions expressives pour automatiser le verdict.
 - ▶ cf. assert du C.
 - La visualisation du verdict.
 - La possibilité de lancer facilement les tests.
 - La couverture de test

2/ Framework Mockito

- Solution pour développer des tests en isolation
- Par exemple, le test nécessite :
 - Un composant dont le code n'est pas encore disponible
 - Un composant dont le comportement est non déterministe (p. ex. réseau)
 - Un composant dont le code est très lent
- On parle de **doublure** ou de **bouchon**
- Remarque : les bouchons peuvent aussi être employées dans le code source

Démonstration JUnit et Mockito

- Exemple d'une classe Calculator