



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

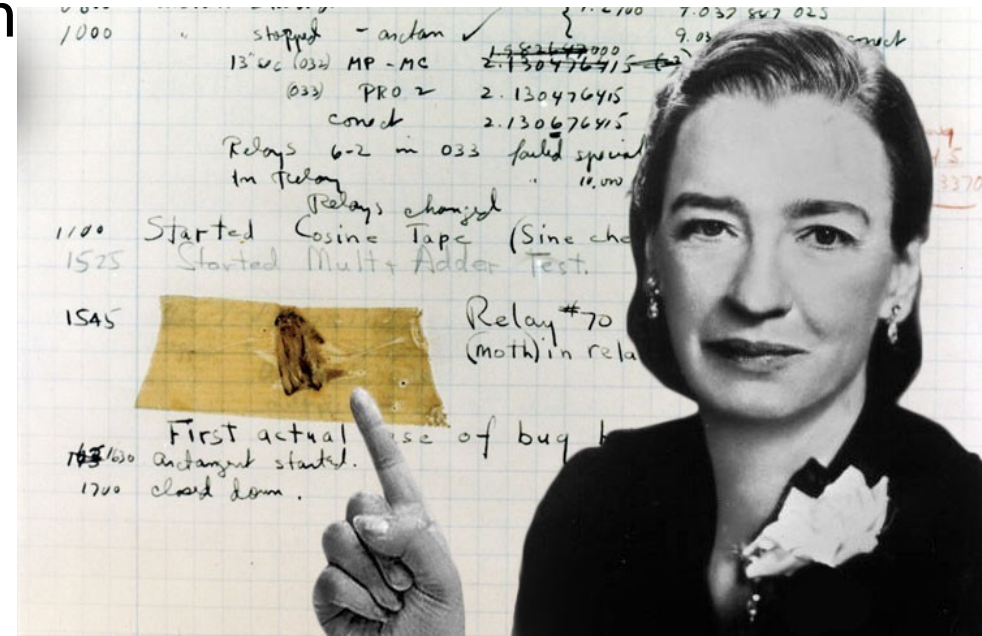
Pourquoi tester ?

4

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

1) La chasse aux bugs

- Un bug est un défaut de conception ou de réalisation d'un logiciel qui provoque un dysfonctionnement
- Bug : mot popularisé en 1947 par **Grace Murray Hopper** pionnière de l'informatique
- Catégories de bug :
 - Bohrbug : nominal
 - Heisenbug : aléatoire
 - Mandelbug : compliqué à reproduire
 - Schrödinbug : découvert à la lecture du code



Grace Murray 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

2) La régression logicielle

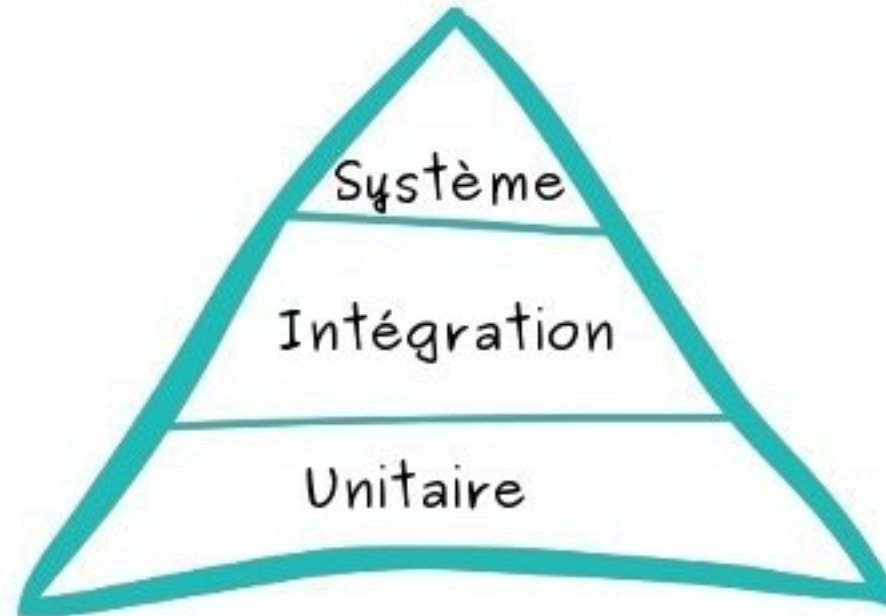
7

- C'est un bug logiciel qui fait qu'une fonctionnalité cesse de fonctionner après une modification du code (refonte ou maintenance)
 - Test : garde-fou pour détecter des modifications erronées du code

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 ?**

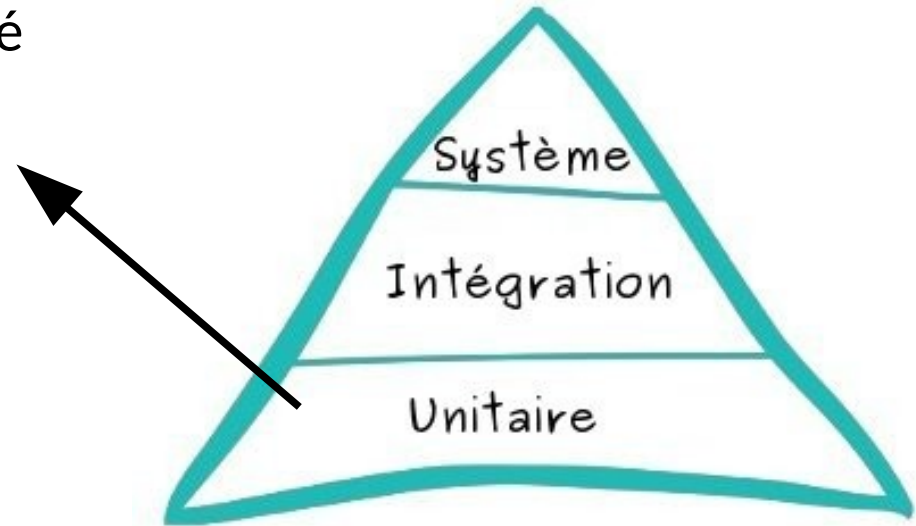
Pyramide des tests



Niveaux de test

■ Test unitaire

- Objectif : vérifier le code de chaque unité
- Erreurs recherchées :
 - ▶ erreurs de codage
 - ▶ erreurs fonctionnelles

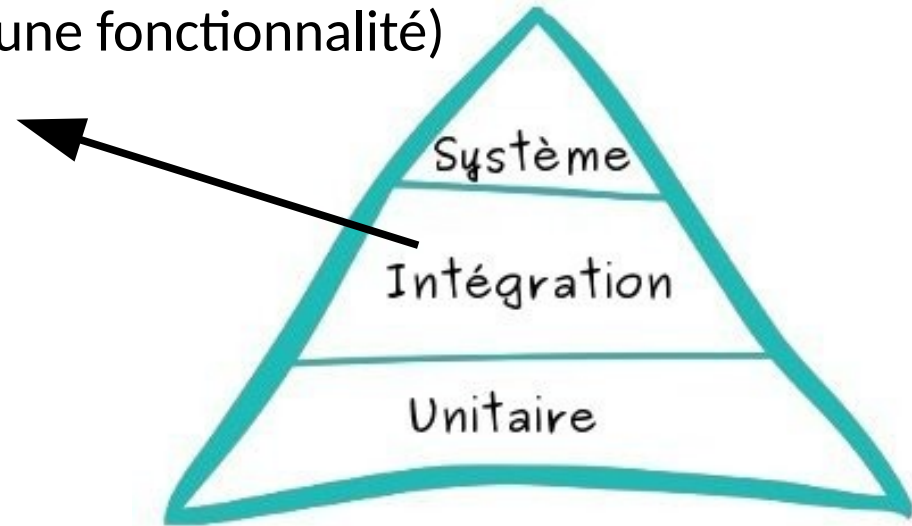


Proportion 80 - 90 %

Niveaux de test

■ Test d'intégration

- Objectif : vérifier l'assemblage d'unités (une fonctionnalité)
- Erreurs recherchées :
 - ▶ erreurs d'interface entre unités

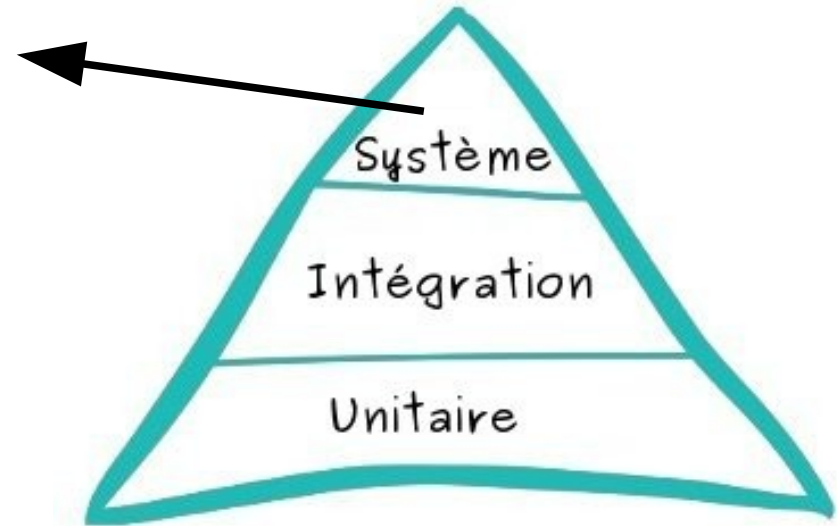


Proportion 5 - 15 %

Niveaux de test

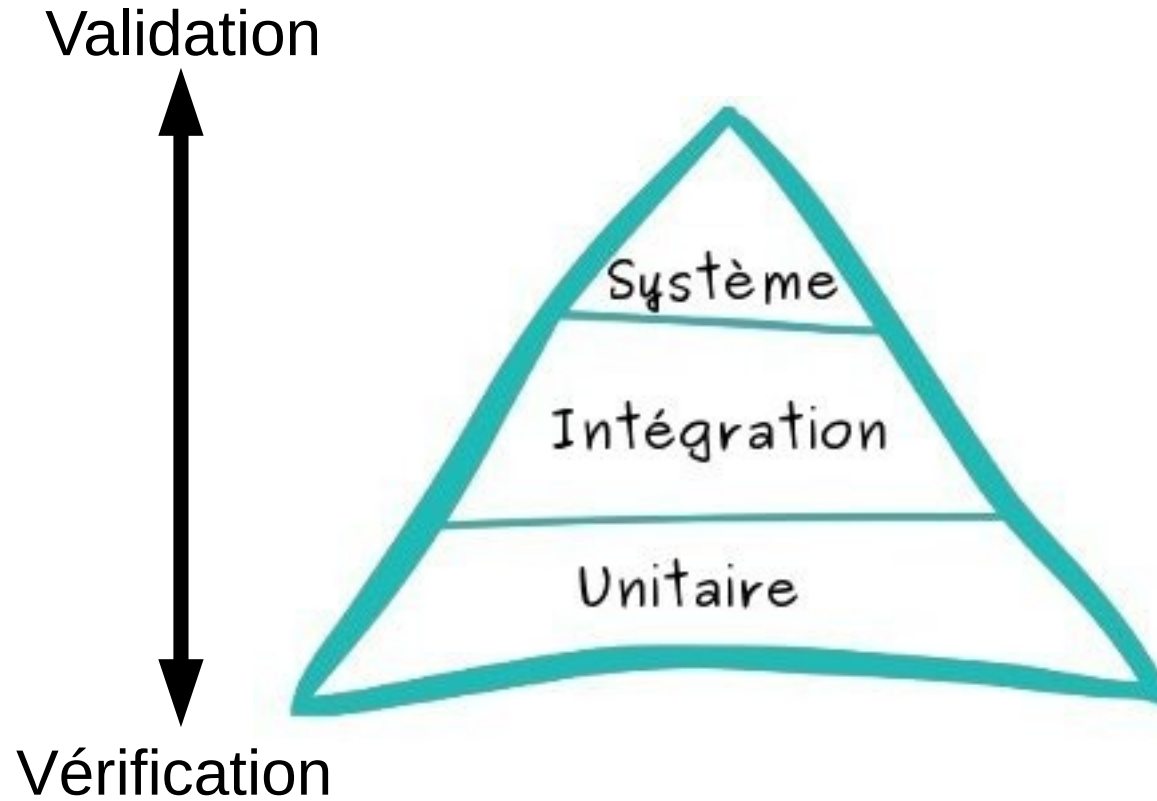
■ Test système

- Objectif : critiquer le produit construit
- Erreurs recherchées :
 - ▶ absence de fonctionnalités
 - ▶ fonctionnalités mal mises en œuvre



Proportion 1 - 5 %

Niveaux de test



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
- Autres types de test
 - tests statiques : p. ex. revue de code
 - tests manuels : p. ex. utilisateur alpha ou bêta
 - tests d'acceptance (recette) : procédures de tests préalablement décrits (scénarios)

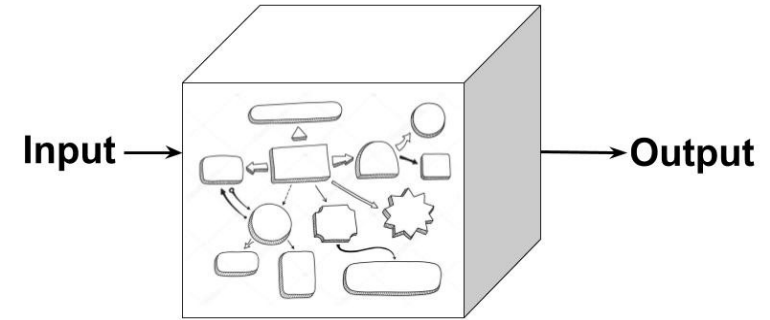
Types de test dynamique

15

- 2 types de test dynamique

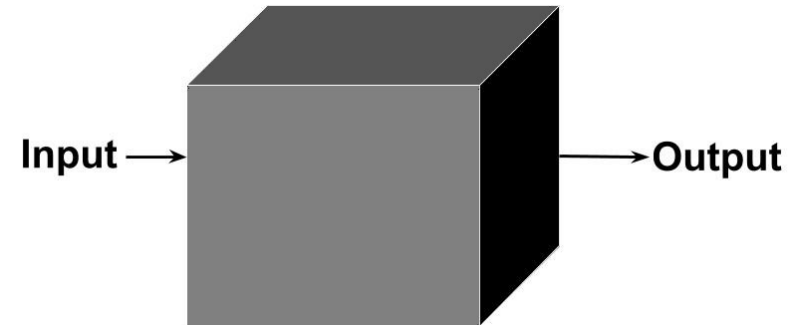
- 1) Test **boîte blanche** (*white-box testing*)

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



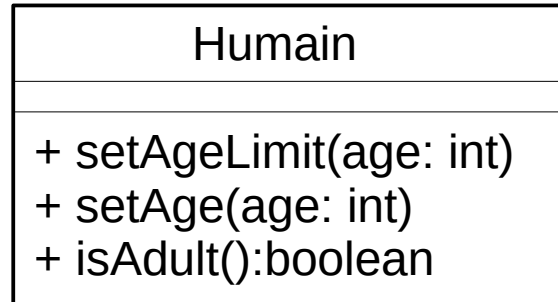
- 2) Test **boîte noire** (black-box testing)

- ▶ Ne connaît pas la structure interne de l'unité testée, on se base uniquement sur les entrées - sorties



Exemple d'un test

- On suppose une classe Human :



- Dans le détail
 - `void setAgeLimit(int age)`
 - `void setAge(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 ?

18

- Les tests sont incomplets par nature
 - On a testé `isAdult()` pour une valeur d'entrée seulement
 - Il faudrait faire d'autres cas de test (p. ex. tests aux 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 left sidebar shows a project tree with coverage percentages for classes and methods. The main editor shows the source code of 'PetValidator.java' with a 'validate' method. A tooltip shows 'Hits: 4' for a specific line. The right sidebar shows a table of 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...			

100 % de test est impossible et inutile

20

- On ne peut pas tout tester
 - Par exemple, tout ce qui est dépendant du réseau, d'une infrastructure personnalisée ne peut être testé parce qu'il est non reproductible
- On ne doit pas tout tester
 - Par exemple, on ne teste pas le langage
 - ▶ `Int getAge() { return age ; }`

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

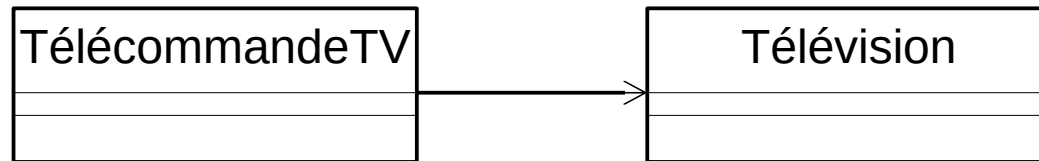
Généralités
sur les tests

2

Tests
unitaires

Test unitaire

- Procédé de vérification d'une unité seulement
 - En Java, les méthodes d'une **classe** seulement
- 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

24

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

Plan du chapitre

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

- Ce code est mauvais parce qu'il est 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

- Ce code est mauvais parce qu'il est 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;    // Ajout d'associations
    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 (exemple 2)

- Décomposer une méthode en 2 pour tester la logique :

```
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() { // Partie à ne pas tester  
    return _getActualRouteLengthInMeters(_database.getRoute());  
}  
static double _getActualRouteLengthInMeters( List<GeodesicPoint> routePoints ) {  
    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

1

Généralités
sur les tests

2

Tests
unitaires

3

Code testable

4

Frameworks de test
JUnit
Mockito

Automatisation du verdict

- Pour l'instant : le verdict n'est pas automatisé

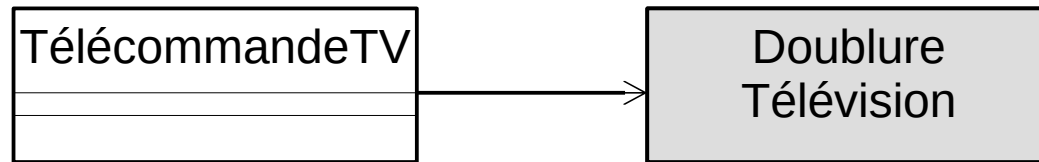
```
Human h = new Human();           // fixture
h.setAgeLimit(150);
try {
    h.setAge(35);                 // donnée de test
    if (h.isAdult()) {           // oracle
        System.out.println("test passe"); // verdict
    } else {
        System.out.println("test echoue");
    }
} catch (Exception 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.
 - ▶ `assertTrue`, `assertEquals`, `assertArrayEquals`, ...
 - 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**
 - Par exemple :



- Par exemple :
 - ▶ Doublure des classes `Be ll` et `Sys t e m` pour l'exemple de l'alarme
- Remarque : les mocks peuvent aussi être employés dans le code source (on parle de **bouchon**)

Démo JUnit et Mockito : calculatrice

Plan du chapitre

37

1

Généralités
sur les tests

2

Tests
unitaires

3

Code testable

5

Développement
dirigé par
les tests

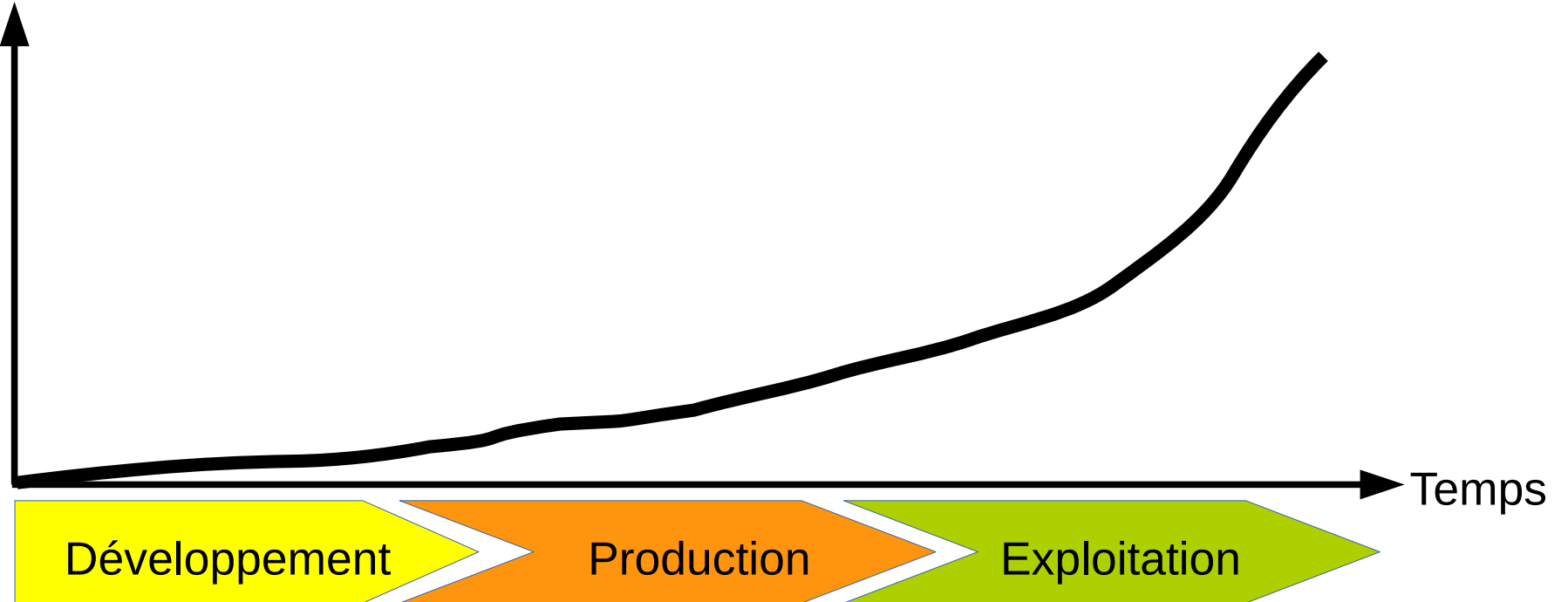
4

Frameworks de test
JUnit
Mockito

Quand tester ?

- Le plus tôt possible
 - Coût d'un bug dans le cycle de vie d'un logiciel :

Charge de travail



Quand tester ?

- Dans l'approche agile, il est du devoir du développeur de tester son code
 - Les tests sont programmés :
 - ▶ Après l'écriture du code d'une fonctionnalité
 - ▶ Après la découverte d'un bug
 - Activité de développement
 - 1) Vite fait, pas Propre : *Quick and Dirty*
 - 2) Codage des tests
 - 3) Refonte : *Refactor*

TDD : Test Driven Development

40

- Et si, les tests étaient programmés avant la fonctionnalité ?
 - TDD : développement dirigé par les tests
 - Activité de développement
 - 1) Codage des tests
 - 2) Vite fait, Pas propre : *Quick and Dirty*
 - 3) Refonte : *Refactor*
- Règle d'or du TDD :
 - « Ne jamais écrire une ligne de code fonctionnel sans qu'une ligne de code de test ne l'exige. »

Démo TDD : bowling

Que retenir de ce chapitre ?

- Les tests sont une obligation
- Ils doivent forcément accompagner le code d'une application
- Il y a différents types de test selon le niveau considéré
 - Unitaire
 - Intégration
 - Système
- Les tests peuvent être écrits avant ou après l'écriture d'une fonctionnalité
- Le code des tests étant du code, il doit donc être propre au même titre que le code source