ENSICAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

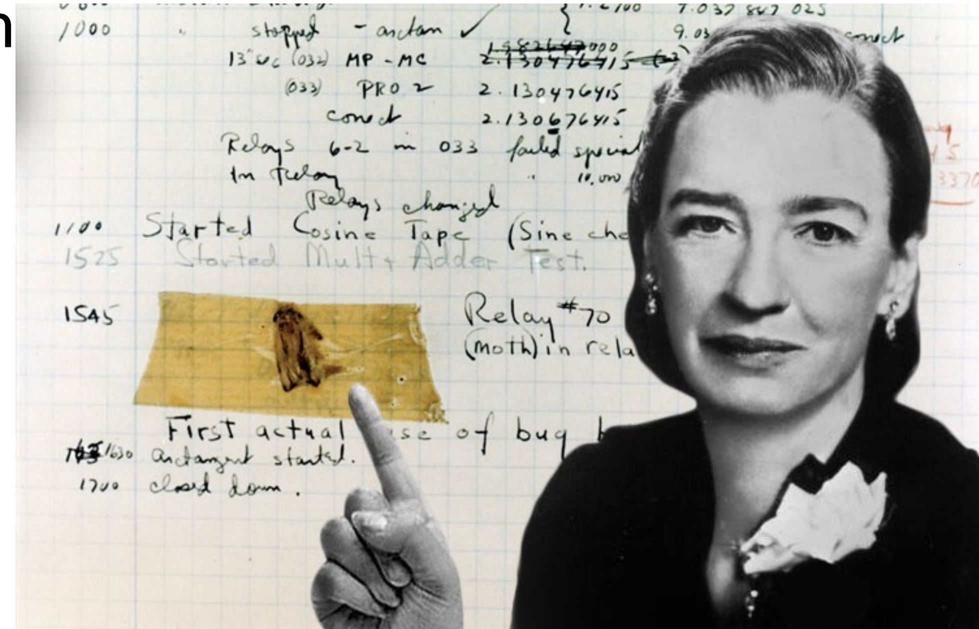
1
Généralités
sur les tests

Pourquoi tester ?

1) Chasser les bugs
2) Prémunir contre la régression logicielle
3) Documenter
4) Rassurer le client

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
  - **Schroedinbug** : 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

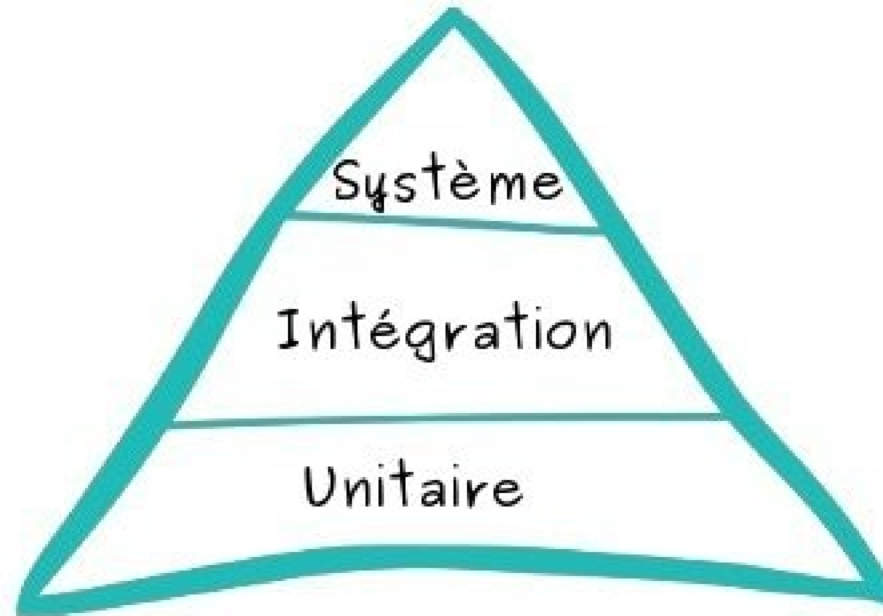
2) Prémunir contre la régression logicielle

- La régression logicielle 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 ?

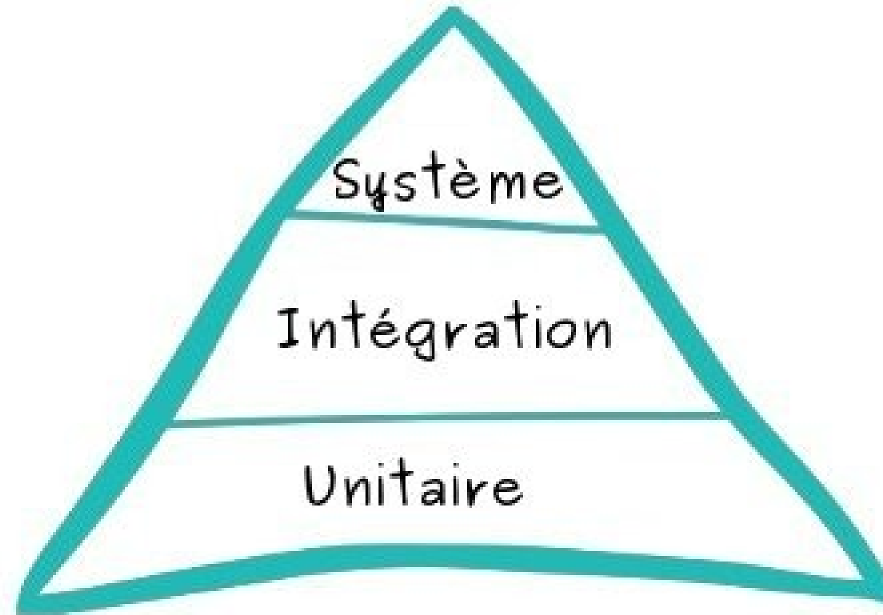
- Procédé de **vérification** et de **validation**
  - Vérification : **le logiciel fonctionne-t-il correctement ?**
  - Validation : **a-t-on construit le bon logiciel ?**

Pyramide des tests



Niveaux de test

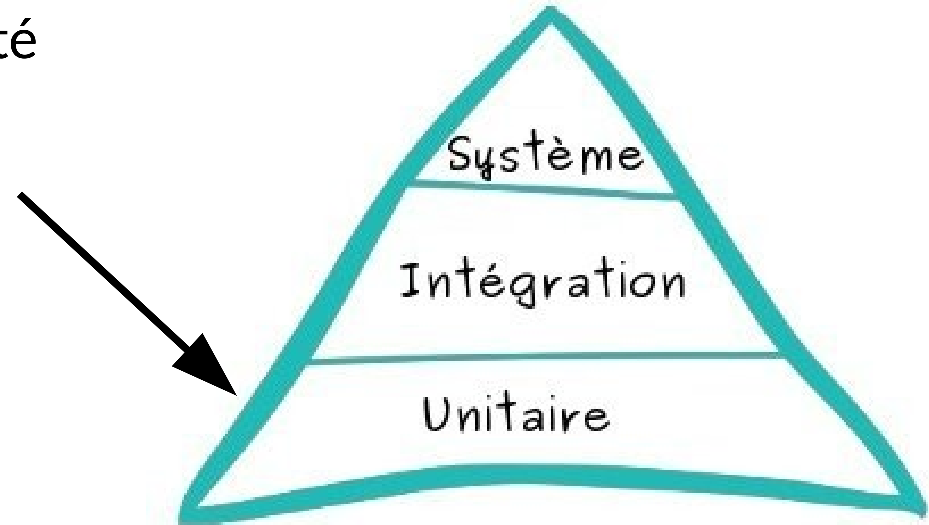
Validation



Vérification

Niveaux de test

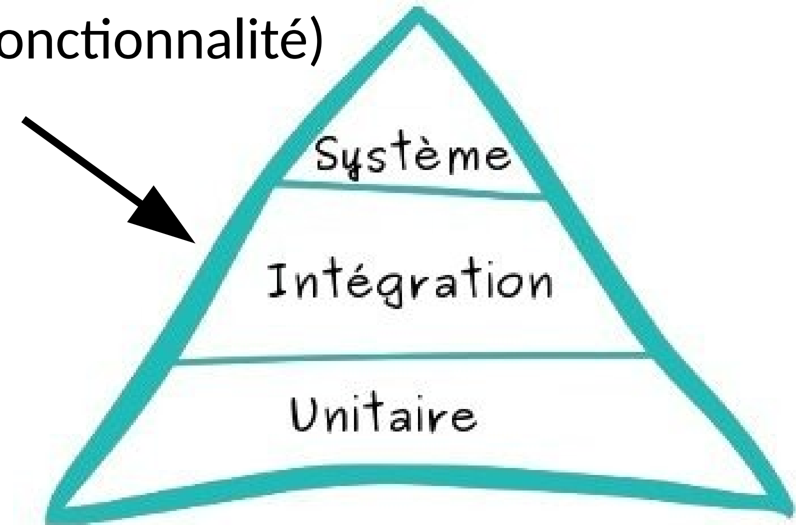
- **Test unitaire**
  - Objectif : vérifier le code de chaque unité
  - Erreurs recherchées :
    - erreurs de codage
    - erreurs fonctionnelles



Proportion 80 - 90 %

Niveaux de test

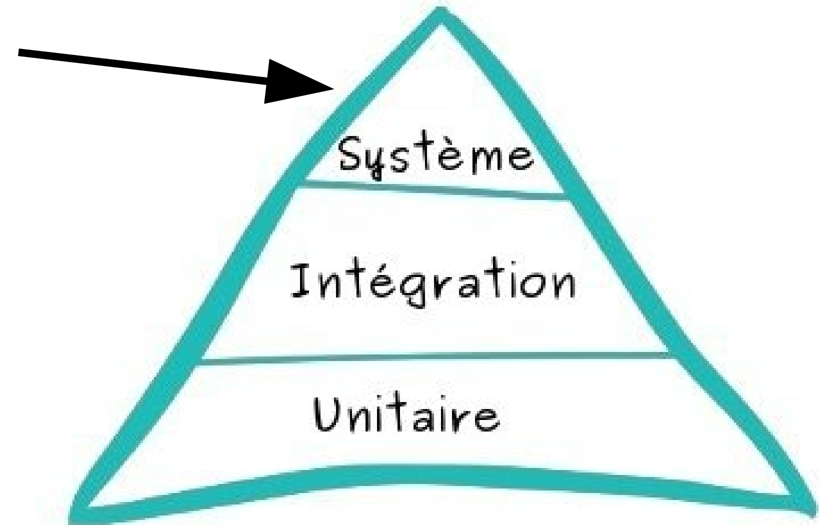
- **Test d'intégration**
  - Objectif : vérifier l'assemblage d'unités (une fonctionnalité)
  - Erreurs recherchées :
    - erreurs d'interface entre unités



Proportion 5 - 15 %

Niveaux de test

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



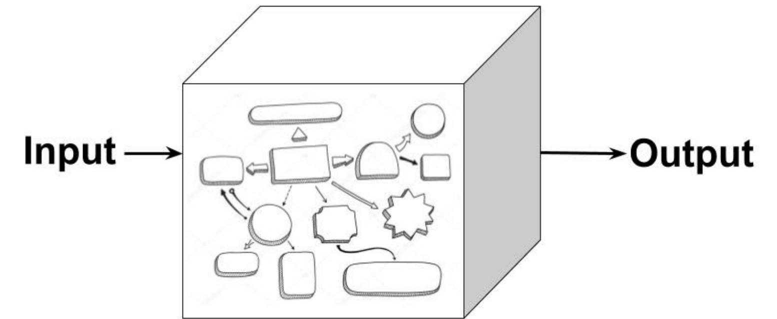
Proportion 1 - 5 %

Le test dynamique

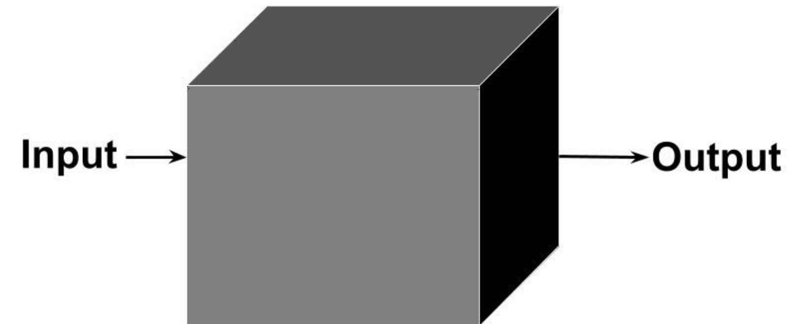
- Dans ce qui suit, nous nous focalisons sur les **tests dynamiques**
  - Le test dynamique est un bout de code qui est exécuté avec l'intention de vérifier ou valider un code
- Autres types de test
  - tests statiques : p. ex. revue de code
  - tests manuels : p. ex. utilisateur alpha ou bêta
  - tests d'acceptance (recette) : test de validation à partir de scénarios préalablement décrits

Types de test dynamique

- 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*)
     - Test uniquement basé sur les entrées - sorties



Exemple d'un test

- On suppose une classe `Human` :

Human
+ setAgeLimit(age: int) + setAge(age: int) + isAdult():boolean

- Unité à tester :
  - `boolean isAdult( ) thows Exception`
    - lève une exception si `age` $\notin [0, \text{ageLimit}]$
    - retourne `true` si `age` $\in \left[18, \text{ageLimit}\right]$
    - retourne `false` si `age` $\in \left[0, 18\right[$

Cas de test

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

```
Human h = new Human();                  // Fixture


h.setAgeLimit(150);
h.setAge(35);                           // Données de test


try {

    if (h.isAdult()) {                  // Oracle
        System.out.println("test a réussi");
    } else {
        System.out.println("Le test a échoué");

    }
} catch (Exception e) {
    System.out.println("Le test a échoué");

}
```

Fixture
Données de test
Oracle
Verdict

Quand s'arrêter de tester ?

- 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

100 % de test est impossible et inutile

- On ne peut pas tout tester, par exemples :
  - Tout ce qui est dépendant du réseau ou d'une infrastructure privée ne peut être testé, parce que les tests sont non reproductibles
  - Le code d'une interface graphique, parce que les tests sont essentiellement visuels
- 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 ?

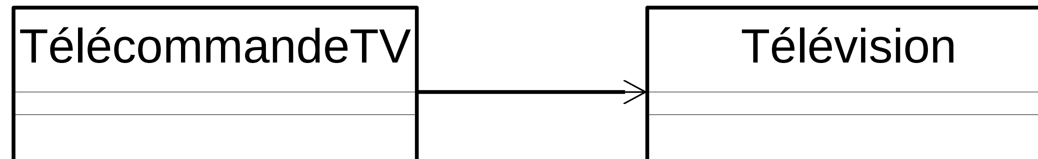
Plan du chapitre

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

- **[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 1)

- 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.getTimeInMillis() > 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.getTimeInMillis() > 100) {
            new Bell().ring();
        }
    }
}
```

Testabilité d'un code

- Ce code est mauvais parce qu'il est non testable :
  2) L'oracle est invérifiable - Pas FIR(**S**)T (self-validating)

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

Code testable

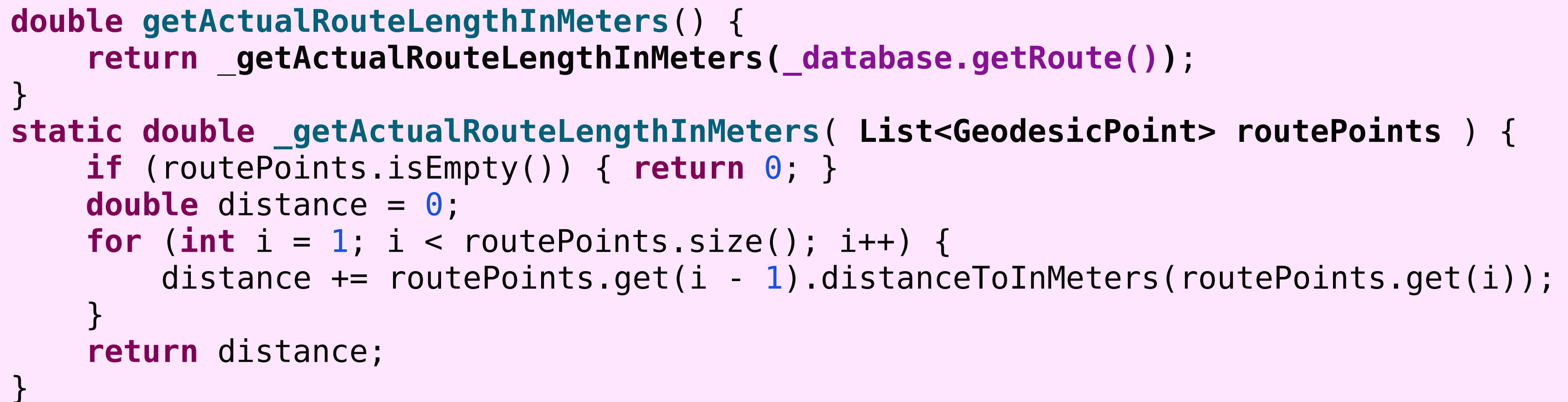
- Solution : Encapsuler et externaliser les dépendances pour rendre un code testable :

```
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 :

```java
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;
}
```



```java
double getActualRouteLengthInMeters() {
    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

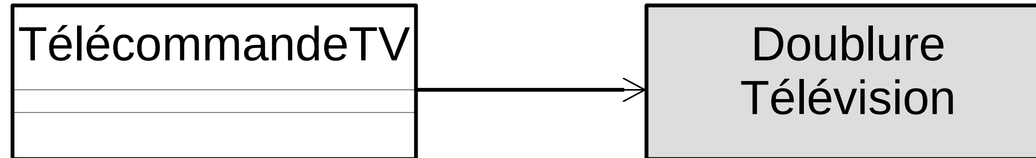
```
Human h = new Human();                              // fixture
h.setAgeLimit(150);
h.setAge(35);                                       // donnée de test

try {
    if (h.isAdult()) {                              // oracle
        System.out.println("Le test a réussi");  // verdict
    } else {
        System.out.println("Le test a échoué");
   }
} catch (Exception e) {
    System.out.println("Le test a échoué");
}
```

1/ Le framework JUnit

- Permet de faciliter l'écriture de test pour le langage Java
- Intégré à IntelliJ IDEA / VisualStudio
- Ce qu'il offre :
  - Des assertions expressives pour automatiser le verdict (des variantes du `assert` du langage C)
    - `assertTrue, assertEquals, assertArrayEquals, ...`
  - La visualisation du verdict
  - La possibilité de lancer facilement les tests
  - La couverture de test

2/ Framework Mockito

- Une 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 Mock : **doublure**
  - Par exemple :



  - Par exemple :
    - Doublure des classes `Bell` et `System` pour l’exemple de l’alarme
  - Remarque : les mocks peuvent aussi être employés dans le code source (on parle de **bouchons**)
- Les Mocks permettent aussi d’espionner l’utilisation d’un objet

Démo JUnit et Mockito : calculatrice