Chapitre 8 : Test logiciel

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

1. Objectif du chapitre

Ce chapitre porte sur une initiation aux tests logiciels et en particulier aux tests dynamiques.
À l'issue de ce chapitre :

- Vous serez sensibilisé à l'importance des tests logiciels dynamiques.
- Vous serez capable de construire du code testable.
- Vous saurez programmer des tests unitaires et utiliser des « bouchons » (ie, *mock*).

2. Généralités sur les tests

2.1. Pourquoi tester ?

- **Seul moyen pour chasser les bugs**. Le code zéro bug n'existe pas, sauf exceptions. Les bugs sont inhérents à l'activité de codage (rappel de l'estimation : 1 à 10 bugs / KLOC). De plus, il n'existe pas de preuve formelle pour des programmes quelconques, donc pas de programme automatique pouvant calculer une preuve de programme. Mais il est possible de détecter certains bugs en testant les programmes pour limiter la casse.
- **Éviter les mauvaises surprises** comme la découverte de bugs après la mise en production.
- **Prémunir contre la régression de code**. Les tests sont aussi des garde-fous pour détecter la régression qu'un développeur peut introduire de façon involontaire lorsqu'il modifie le code du logiciel. Les tests de non-régression assurent que les modifications du code lors de la refonte de code, de l'ajout de fonctionnalités ou de la correction de bugs n'affectent pas le bon fonctionnement des fonctionnalités préexistantes.
- **Rassurer le développeur**. Lorsque le code a atteint une certaine complexité, on commence à craindre les modifications.
- **Rassurer le client**. Le produit final contient des preuves de test du produit.

2.2. Bug

Un bug est un défaut de conception ou de réalisation d'un logiciel qui provoque un dysfonctionnement.
Le mot a été popularisé en 1947 par **Grace Hopper** pionnière de l'informatique.
Il existe plusieurs catégories de bug :

- **Bohrbug** (inspiré de l'atome de Niels Bohr) : un bug qui a toutes les bonnes propriétés, en particulier, il est répétable dans les mêmes conditions. C'est le bug classique.
- **Heisenbug** (inspiré du principe d'incertitude de Heisenberg) : un bug dont le comportement est modifié quand on essaye de l'isoler. Le cas typique est celui des exécutions sous débermineur. Comme le

dévermineur réserve plus de mémoire à l'exécution que l'exécution normale, une erreur d'adressage mémoire qui apparaît lors de l'exécution du logiciel (le fameux « Segmentation fault ») peut ne pas se révéler sous dévermineur parce que, par malchance, la mauvaise adresse utilisée fait partie du bloc de mémoire réservé.

- **Mandelbug** (inspiré des fractales de Mandelbrot) : un bug dont les étapes pour le reproduire sont si complexes qu'il semble se reproduire de façon aléatoire et chaotique. Par exemple, les situations de compétition entre « thread » peuvent entraîner des Mandelbug où le comportement du programme est différent à chaque fois que celui-ci est exécuté.
- **Schrödinbug** (inspiré du chat de Schrödinger) : un bug qui ne se révèle pas à l'exécution, mais qui est découvert lorsque quelqu'un relit le code source ou utilise le logiciel d'une façon inhabituelle.

2.3. Qu'est-ce qu'un test ?

Un test est un procédé de vérification et validation (V&V).

- Vérification : **le logiciel fonctionne-t-il correctement ?**
  - Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que des exigences spécifiées ont été remplies.
- Validation : **a-t-on construit le bon logiciel ?**
  - Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que les exigences, pour un usage ou une application voulue, ont été remplies.

2.4. Niveaux de test

On distingue globalement trois niveaux de test, ce que schématise la pyramide des tests.

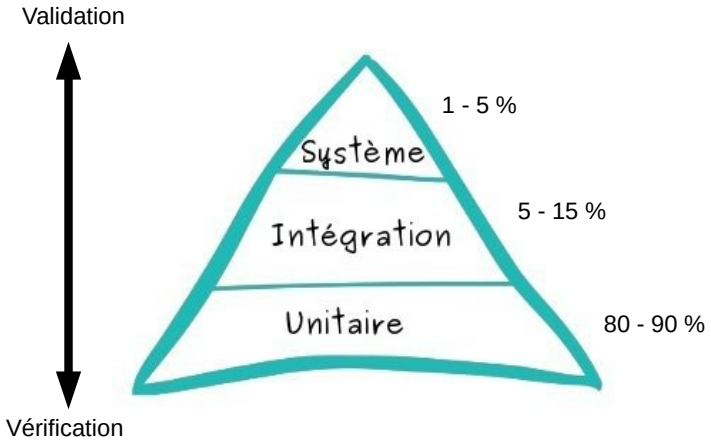


Figure 1 :La pyramide des tests.

De bas vers le haut, on distingue :

1. Tests unitaires : Ils ont pour but de guider le développeur. Ce sont des tests en isolation des unités de développement.
   - Erreurs recherchées : les erreurs de codage et les erreurs fonctionnelles dans les unités.
2. Tests d'intégration : Ils ont pour but de guider l'équipe de développement. Ce sont des tests qui portent sur l'assemblage des unités.
   - Erreurs recherchées : les erreurs d'interface entre unités.
3. Tests système : Ils ont pour but de critiquer le produit. Ils testent le système dans son ensemble et en

particulier l'interaction homme machine (IHM).

- Erreurs recherchées : l'absence ou défaillance de fonctionnalités.
- En production, on distingue plusieurs niveaux de tests du système.
  - Version alpha : tests auprès d'utilisateurs internes au projet.
  - Version bêta : tests auprès d'utilisateurs externes mais avertis.

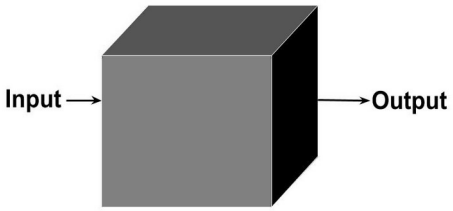
Les tests unitaires sont essentiellement de l'ordre de la vérification tandis que les tests systèmes sont essentiellement de l'ordre de la validation.

2.5. Tests dynamiques

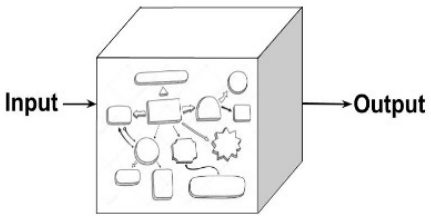
Dans ce qui suit, nous nous focalisons sur les tests dynamiques. Un test dynamique est un bout de code qui est exécuté avec l'intention de vérifier ou valider un bout de code fonctionnel.
Remarque : le test logiciel inclut aussi des tests manuels qui ne sont pas présentés ici (Voir les quadrants de test Agile dans la littérature).

2.6. Types de tests dynamiques

1. Tests **boîte noire** (*black-box testing*)
   - L'écriture des tests se fait sans connaître la structure interne du code à tester.
   - Ils ne s'intéressent qu'aux entrées et sorties.



2. Tests **boîte blanche** (*white-box testing)*
   - L'écriture des tests tient compte de la structure interne de l'unité testée. En particulier, ils testent chaque chemin possible dans le code, par exemple chaque branche d'un '`if`'.



2.7. Exemple fil rouge

On suppose une classe `Human` possédant les méthodes publiques suivantes :

- `void setAge(int age)`
- `void setAgeLimit(int age)`
- `boolean isAdult() thows Exception`, qui :
  - lève une exception si age $\notin$ [`0`, `ageLimit`]
  - retourne `true` si `age` $\in$ [`18`, `ageLimit`]
  - retourne `false` si `age` $\in$ [`0`, `18`[

2.7.1. Étape 1 : Objectif de test

On choisit une caractéristique ou une fonction à tester : c'est l'objectif de test.

- Par exemple : On décide de tester la méthode `isAdult()` dans le cas nominal où le paramètre age est dans `[18, ageLimit]`.

2.7.2. Étape 2 : Jeu de test

Ensuite, on choisit les données pour le test : c'est le jeu de test.
Pour notre exemple, il faut choisir :

- une valeur pour le seuil d'âge limite, par exemple `ageLimit` = 150.

- une valeur pour le paramètre age qui soit dans l'intervalle : [18, `ageLimit`], par exemple : `age` = 35.

2.7.3. Étape 3 : Fixture

Avant d'exécuter le code du test, il faut amener l'objet dans l'état attendu : c'est la fixture.

- Dans notre exemple : création de l'objet `Human` et positionnement de son état.

```
Human h = new Human();
h.setAgeLimit(150);
```

2.7.4. Étape 4 : Oracle

On compare le résultat obtenu au résultat attendu : c'est l'oracle. Il est implémenté par une assertion, ie, une expression booléenne censée être vraie.

- Dans notre exemple :
  - `h.isAdult()` retourne `true`

2.7.5. Étape 5 : Verdict

On en déduit si le test a réussi ou échoué : c'est le verdict.

- si on récupère `true` : le test passe
- si on récupère `false` : le test échoue
- s'il y a une levée d'`exception` non attendue, c'est une erreur, le test est inconclusif.

2.7.6. Étape 5 : Test exécutable : Cas de test

L'ensemble du test exécutable s'appelle un cas de test.
Définition IEEE 610 : Un cas de test est un ensemble de **données de test**, de **pré-conditions** d'exécution et de **résultats attendus** développés pour un objectif, tel qu'exécuter un chemin particulier d'un programme (test de type boîte blanche) ou vérifier le respect d'une exigence spécifique (test de type boîte noire).

```
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 passe");     // verdict
  } else {
    System.out.println("test echoue");
  }
} catch (Exception e) {
  System.out.println("erreur, test inconclusif");
}
```

2.8. Quand s'arrêter de tester ?

Les tests sont incomplets par nature. On a testé `isAdult()` pour une valeur d'entrée seulement, mais il faudrait les tester toutes. Ici, ce n'est pas possible puisque que `age` $\in \mathbb{N}$. Savoir quand s'arrêter est une question d'expérience. Un indice toutefois, la **couverture** des tests.

2.9. Couverture de code

La couverture de code est une mesure qui permet d'identifier la proportion du code testé. Elle correspond au ratio de code source qui est exécuté quand une suite de test est lancée. Il existe des utilitaires de couverture de code dans pratiquement tous les langages (par exemple Jacoco en Java).

2.10. Doit-on écrire des tests pour tout ?

Non, uniquement pour les choses qui peuvent raisonnablement être source de bug. On n'écrit pas de test pour des instructions qui peuvent être vérifiées par l'OS, l'environnement d'exécution ou le compilateur. Prenons l'exemple de la classe `AClass` suivante :

```
public class AClass {
  int _x;
  public AClass(int x) { _x = x; }
  int getX() { return _x; }
  void setX(int x) { _x = x; }
}
```

On en teste pas `getX()` et `setX()`. Tester `getX(setX(y))==y` revient à tester _x=y, c'est-à-dire à tester le compilateur.

2.11. À quelle fréquence dois-je exécuter mes tests ?

Exécutez le test unitaire aussi souvent que possible, idéalement à chaque changement dans le code.

2.12. Limite des tests

Mais, 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 est faux, mais les tests aussi !

- Question : Doit-on faire des tests de tests ?

3. Test unitaire

Le test unitaire vise deux objectifs de vérification d'une unité en isolation.

1. Vérifier le bon fonctionnement d'une unité du logiciel.
2. Se prémunir contre la régression de code dans l'unité.

En programmation orientée objet, l'**unité** est la **classe**. À chaque classe du code source, on associe une autre classe qui la teste.
Les tests unitaires testent une seule classe et sont indépendants les uns des autres (test en isolation). Cela permet de s'assurer que si un test échoue, c'est la classe testée qui est fautive.
Par exemple, on teste les méthodes publiques et protégées de la classe TelecommandeTV en isolation. Si on teste la classe TelecommandeTV en utilisant une télévision dans les tests, c'est du **test d'intégration**.

3.1. Qualités d'un test unitaire : FIRST

Un test doit impérativement posséder les qualités suivantes :

- **[F]**ast (Rapide) : plusieurs centaines de tests par seconde.
- **[I]**solated (Isolé) : le test ne doit dépendre d'aucun autre test. Les tests peuvent être exécutés dans

n'importe quel ordre.

- **[R]**epeatable (Répétable) : chaque exécution doit produire le même résultat quel que soit le moment ou l'environnement d'exécution. Cela interdit par exemple d'utiliser des valeurs aléatoires non reproductibles.
- **[S]**elf-validating (Auto-évaluable) : pas de recours à un utilisateur pour l'oracle.
- **[T]**imely (Juste à temps) : programmé dès que l'on a la connaissance sur la fonctionnalité.

3.2. Testabilité d'un code

On n'écrit pas du code testable comme du code classique.
Prenons l'exemple d'une alarme qui se déclenche à une heure butoir dans un agenda.

- On part du très mauvais code suivant :

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

Voyez-vous pourquoi il est mauvais ? Ce code est mauvais parce qu'il n'est pas testable automatiquement.

3.2.1. Contrôler une entrée indirecte

La méthode `check()` a deux données de test :

- la date à laquelle l'alarme se déclenche,
- la date courante.

Mais, dans `check()` la date courante est nécessairement fournie par `System`. Mais, `System` étant incontrôlable**, ce code est mauvais parce qu'il n'est pas testable**. La date courante est une entrée indirecte de `check()` qu'il faut pouvoir contrôler.

3.2.2. Observer une sortie indirecte

L'objectif du test est :

- Vérifier que l'alarme se déclenche si la date courante dépasse à la date butoir.

Quel est l'oracle ?

- Écouter si on entend ou pas quelque chose. Mais ce n'est pas auto-évaluable.
- Observer si la méthode `ring()` de Bell a été appelée. Mais l'objet `Bell` est créé dans `check()`, il n'est pas observable.

Dans les deux cas, **ce code est mauvais parce qu'il n'est pas testable**. L'interaction avec `Bell` est une sortie indirecte de `check()`, qu'il faut pouvoir observer.

3.2.3. Code testable

Solution : Encapsuler et externaliser des dépendances pour rendre ce code testable.

```
public final class Agenda {
  public Agenda(int limit, Clock clock, Bell bell) {
    _clock = clock; _bell = bell; _limit = limit;
  }
```

```
  public Agenda() {
    _clock = System; _bell = new Bell(); _limit = 100;
  }
  public void check() {
    if (_clock.getTimeInMillis() > _limit) {
      _bell.ring();
    }
  }
}
```

Il y a deux constructeurs. Le constructeur par défaut est pour l'exécution normale et le constructeur avec paramètres contrôlables est pour les tests.

4. Frameworks de test : JUnit

JUnit est un exemple de framework de test (peut être le plus connu). Il permet de faciliter l'écriture de test pour le langage Java. Ce qu'il offre :

- des assertions expressives pour automatiser le verdict,
- la visualisation du verdict,
- la possibilité de lancer facilement les tests.

4.1. Organisation des codes

Les tests ne doivent pas être dans la source de l'applicatif. Une organisation classique contient les dossiers suivants :

- **bin** (ou **build**) pour les exécutables.
- **src** pour le code source applicatif.
- **test** pour le code source des tests.

L'organisation en paquets des tests suit celle des sources.
Par exemple, pour tester la classe `fr.ensicaen.ecole.projet.MaClasse` du dossier `src`, on trouve la classe `fr.ensicaen.ecole.projet.MaClasseTest` dans le dossier `test`.
Avantages :

- Pas de pollution des sources par les tests.
- Permet de livrer l'applicatif avec ou sans les tests.

4.2. Les assertions de base de JUnit

Les assertions servent à décrire l'oracle d'un cas de test. Ce sont des variations autour du `assert` :

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
- `assertEquals(Object expected, Object actual)`
- `assertEquals(double expected, double actual, double delta)`
- `assert[Not]Same(Object expected, Object actual)`
- `assert[Not]Null(Object actual)`
- `assertArrayEquals([] expecteds, [] actuals)`
- `fail()`…

4.3. Méthode de test JUnit : @Test

L'écriture d'une méthode de test dans une classe de test représentant un cas de test, doit être :

- annotée par `@Test`.
- publique, sans paramètre et de type de retour `void`.

Exemple reprenant le cas d'étude `Human` :

```
@Test
public void test_is_an_adult_when_age_greater_than_18() {
  Human h = new Human();
  h.setAgeLimit(150);
  h.setAge(35);
  assertTrue(h.isAdult());
}
```

4.4. Tester la levée d'exception

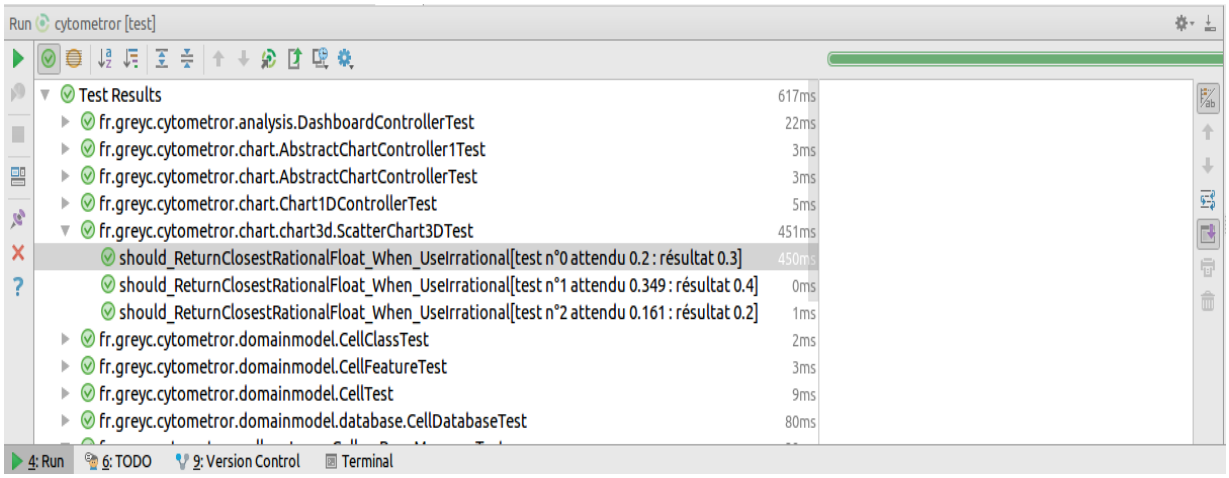
Exemple : vérifier que l'appel de la méthode `setAge()` lève une quand le paramètre est hors limite. Si l'exception est levée alors le verdict est positif.

```
@Test
public void test_throw_exception_if_age_is_greater_than_the_maximum()
  Assertions.assertThrows(OutOfLevelException.class,
                          () -> {
                              Human h = new Human();
                              h.setAgeLimit(150);
                              h.setAge(151);
                          });
}
```

4.5. Visualisation du verdict de JUnit

Le verdict est représenté graphiquement dans un IDE :

- Le test passe : barre verte.
- Le test échoue : barre rouge.
- Levée d'une exception attendue : barre verte.
- Levée d'une exception inattendue : barre rouge.



4.6. Fixture JUnit : @BeforeEach

Préambule des tests : une méthode annotée par `@BeforeEach`.

- Elle est appelée avant chaque méthode de test de la classe.
- Elle sert à factoriser la fixture des tests si elle est commune à tous les tests de la classe.

```
public final class test_level_management {
  private Human _human;
  @BeforeEach
  public void setUp() throws Exception {
    _human = new human();
    _human.setAgeLimit(150);
  }
  @Test
  public void test_age_greater_than_max(){
    Assertions.assertThrows(OutOfLevelException.class,
                            () -> {
                                h.setAge(151);
                            });
  }
}
```

5. Frameworks de test : Mockito

Mockito est le plus connu des frameworks qui permettent l'écriture de « doublures » pour le langage Java.

5.1. La doublure pour les tests

Une doublure remplit deux rôles :

1. Le substitut (*fake*) : une classe qui est une implémentation partielle et qui retourne toujours les mêmes réponses selon les paramètres fournis sans fournir de code pour cela (donc pas de bug inséré).
2. L'espion (*spy*) : une classe qui vérifie l'utilisation d'une classe doublée après son exécution.

Remarque : Les doublures peuvent aussi être employées dans le code fonctionnel pour remplacer une classe non encore développée. Dans ce cas, on parle de « bouchon » (*mock* en anglais).

5.2. Pourquoi des doublures ?

Les doublures offrent une solution pour développer des tests qui nécessitent :

- un composant dont le code n'est pas encore disponible.
  - p. ex : persistance des données.
- un composant difficile à mettre en place.
  - p. ex : une base de données.
- un comportement exceptionnel d'une classe.
  - p. ex : déconnexion dans un réseau.
- un composant dont le code est lent (*empêche le F de FAST*).
  - p. ex : construction d'un maillage.
- une fonction qui a un comportement non-déterministe.
  - p. ex : réseau.

5.3. Exemple d'utilisation du framework Mockito

On suppose la classe `MaClasse` à doubler :

1. D'abord on importe Mockito dans le fichier test :
   - `import static org.mockito. Mockito.*;`
2. On crée l'objet de la classe à tester en utilisant les doublures à la place des objets réels avec la méthode `mock()` :
   - `MaClasse mc = mock(MaClasse.class);`
3. On décrit le comportement attendu de la doublure :
   - `when(mc.maMethode()).thenReturn(56);`
   - `when(mc.maMethode()).thenThrow(new Exception());`
4. On vérifie que l'interaction avec les doublures est correcte :
   - `mc.verify()` avec les bons tests.

5.4. Exemple d'utilisation pour le substitut pour une classe « ServiceAuthentification »

Le test suivant vérifie que la méthode `verifie()` de la classe `MessagerieUtilisateur` fonctionne correctement quand on donne un mot de passe correct ou non. Le problème, c'est que la méthode fait appel à une classe tierce ServiceAuthentification. Il est donc nécessaire d'utiliser une doublure pour garantir la qualité « isolée » du test.

```
@Test
public void testLireMessagesAvecBonMotDePasse() {
    ServiceAuthentification sa = mock(ServiceAuthentification.class);
    when(sa.verifie("toto", "mdp")).thenReturn(true);
    when(sa.verifie("toto", "mauvais_mdp")).thenReturn(false);
    // On introduit la doublure dans la classe à tester.
    MessagerieUtilisateur msg = new MessagerieUtilisateur(sa);
    // Oracle
    assertTrue(msg.lireMessages("toto", "mdp"));
    assertFalse(msg.lireMessages("toto", "mauvais_mdp"));
}
```

5.5. Exemple d'utilisation pour l'espionnage pour une classe « ServiceAuthentification »

Le test suivant vérifie que la méthode `verifie()` de la classe `MessagerieUtilisateur` fait bien appel à la méthode `verifier()` de la classe tierce `ServiceAuthentification` exactement une fois pour s'exécuter.

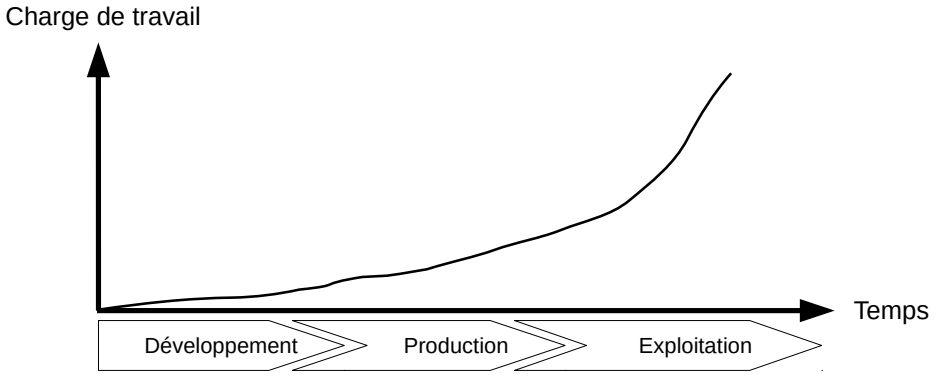
```
@Test
public void testLireMessages() {
    ServiceAuthetification sa = mock(ServiceAuthentification.class);
    Mockito.when(sa.verifier("toto", "mdp")).thenReturn(true);
    Mockito.when(sa.verifier("toto", "mauvais_mdp")).thenReturn(false);
    // étape 1 : on introduit la doublure
    MessagerieUtilisateur msg = new MessagerieUtilisateur(sa);
    // étape 2 : on lance le traitement
    msg.lireMessages("toto", "mdp");
    msg.lireMessages("toto", "mauvais_mot_de_passe");
    // étape 3 : on vérifie que la méthode verifier() a bien été
```

```
    // appelée exactement une fois avec ces paramètres.
    Mockito.verify(sa, times(1)).verifier("toto", "mdp");
}
```

6. Développement dirigé par les tests

6.1. Quand tester ?

Le coût de correction d'un bug est exponentiel avec l'avancement dans le cycle de vie du projet. Il faut donc tester son logiciel le plus tôt possible.



Coût d'un bug dans le cycle de vie d'un logiciel.

6.2. TDD : Test Driven Development

Et si, les tests étaient programmés avant la fonctionnalité ? On aboutit à une nouvelle façon de développer : le développement dirigé par les tests (TDD). On écrit le code du test juste avant d'écrire le code de la fonctionnalité à tester.
La règle d'or du TDD : « Ne jamais écrire une ligne de code fonctionnel sans qu'une ligne de code de test ne l'exige. »

6.3. En pratique

- **Le mantra du TDD : Toujours garder la barre verte pour garder le code propre.**
  - Red → Green → Refactor → Green
- Plus précisément :
  1. Écrire un test pour une fonctionnalité à développer.
  2. Exécuter et constater que la barre est **rouge**.
  3. Écrire le code qui permet de faire passer le test (et rien que ce code).
  4. Lancer le test et vérifier qu'il passe : barre **verte**.
  5. Remanier le code : garder la barre **verte**.
  6. Relancer tous les tests précédents : garder la barre **verte**.

6.4. Pourquoi écrire les tests avant le code ?

6.4.1. Avantages psychologiques

- Travailler plus sereinement.
  - Si un bug apparaît, il sera détecté très tôt par les tests.

- Grâce au code testé :
  - On est serein.
  - On planifie mieux son travail.
  - On évite les paniques de dernière minute.
- Rester focalisé sur la tâche.
  - Oblige à réfléchir à ce que doit faire le code avant de l'écrire.

6.4.2. Avantages techniques

- Garder un temps de développement constant. Tout au long du développement, il y aura le même temps consacré au test et le même temps consacré au développement. On passe beaucoup moins de temps à déboguer.
- On a toujours quelque chose à montrer au client, dont des tests. Il est impossible de livrer du code non testé. Les tests de **non-régression** sont directement inclus.
- Quand on écrit le test, on ne se réfère qu'à la spécification et pas à l'implémentation (test boîte noire). Il y a moins de biais et moins de dépendance au code.
- Le code de test sert aussi de documentation du code.

6.5. Pourquoi écrire un seul test à la fois ?

- Développement itératif.
  - Optique du « petit pas ».
  - Le test représente une partie du contrat total de la fonctionnalité testée.
  - On ne passe au développement de la fonctionnalité suivante que lorsque la précédente a été validée.
- Pourquoi ne pas écrire tout le code fonctionnel d'un coup ?
  - On n'écrit que le code qui a besoin d'être validé par un test sinon on risque d'introduire du code applicatif non testé ou d'ajouter des fonctionnalités inutiles.

6.6. Pourquoi commencer par la barre rouge ?

Il s'agit d'éviter les « happy tests » : des tests qui s'exécutent avec succès (barre verte) alors qu'ils ne devraient pas (barre rouge).
Les causes possibles des happy test sont :

- Le test est construit par copier-coller d'un autre test sans modification.
- Le test est construit sans l'annotation @Test en JUnit.
- Le test ne teste pas réellement la méthode cible.

7. Correction de bug

La correction de bugs occupe une part importante de la charge d'un développeur.

7.1. Mauvaise pratique

- Pratique classique (parfois enseignée)
  1. Truffer le code d'appels d'affichage (*printf*) pour localiser la partie du code fautive.
  2. Lancer le programme pour espérer isoler les lignes de bugs.
  3. Corriger le bug en modifiant et relançant le programme autant de fois que nécessaire.

4. Une fois le bug corrigé, effacer les appels d'affichage.

- Une amélioration sera d'utiliser le dévermineur (p. ex. gdb en C).
  1. Exécuter le programme sous dévermineur.
  2. Ajouter des points d'arrêt pour examiner les valeurs des variables.
  3. Corriger le bug une fois localisé.
- Mais même si la deuxième méthode est plus efficace que la première, elles sont toutes les deux inefficaces :
  - Plus le code est développé, plus le temps de recherche de bug est important.
  - Il ne prémunit pas contre une régression future (n'empêche pas le bug de se reproduire).
  - L'introduction de lignes de code peut changer le comportement du programme (cf. Heisenbug).

7.2. Bonne pratique

1. Écrire des tests pour détecter le bug tel qu'il est décrit.
2. Corriger le code applicatif afin de passer les tests.

Ainsi, les tests permettent aussi d'éviter que le bug ne se reproduise plus tard.

8. Que retenir de ce chapitre ?

- Les tests sont une obligation. Ils visent deux objectifs :
  - chasser les bugs,
  - mettre des garde-fous contre la régression. La régression de code est inhérente à l'activité de développement selon les méthodes agiles puisqu'elle inclut de la refonte de code.
- Les tests doivent forcément accompagner le code d'un logiciel. Un code sans test est inutile. En empruntant la métaphore du génie civil, les tests sont les armatures du béton. On sait en génie civil, qu'il est impossible de faire une tour de 200 m sans béton armé.
- Il y a trois types de test selon le niveau considéré.
  - Unitaire : test en isolation (utilisation de bouchons si nécessaire pour isoler). Ce sont le plus nombreux.
  - Intégration.
  - Système.
- Les tests doivent être écrits au même moment que l'écriture de la fonctionnalité, que ce soit juste avant ou juste après. Personnellement, j'utilise le TDD quand la fonctionnalité est bien définie et les tests après le codage sinon.
- Le code des tests étant du code, il doit donc être propre au même titre que le code source applicatif.