amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Master informatique
Génie logiciel : Tests
Codes UE : SINA09EL
Tests

Tests

Arnaud Labourel

1 Tests

1.1 Qu'est-ce qu'un test en programmation ?

Il est important de tester le code que l'on écrit afin d'avoir des garanties sur son bon fonctionnement. Un code non testé n'a aucune valeur et par conséquent tout code doit être testé. En effet, même le développeur le plus aguerri peut faire des fautes d'inattention. Même une petite erreur peut faire perdre énormément de temps en débogage. Il est donc important de tester son code au fur et à mesure de sa production afin de trouver au plus tôt de telles erreurs. Il est difficile de donner une définition d'un test, car ceux-ci peuvent prendre tout un tas de forme. Rien qu'exécuter le code qu'on a écrit et vérifier à la main son comportement constitue en soi un test. C'est d'ailleurs généralement la dernière étape du processus de test d'un logiciel. En effet, il est souvent très complexe de tester automatiquement le bon comportement d'une interface utilisateur d'une application. Un autre type de test est le test automatique qui consiste en du code utilisant le code à tester et vérifiant sans contrôle d'une personne le bon fonctionnement du programme. C'est sur ce type de test que nous allons nous concentrer dans cet enseignement. Les tests constituent donc une étape importante du développement logiciel et il existe une nomenclature donnant des noms aux différents types de tests. Les principaux types de test sont les suivants :

— **Tests unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
— **Tests d'intégration** : Tester le bon comportant de partie de programmes formant un tout cohérent appelée module. Cela permet de tester le fonctionnement d'instances issues de classes différentes interagissant ensemble ce qui n'est pas possible avec les tests unitaires.
— **Tests systèmes (anciennement tests fonctionnels)** : Tester que le fonctionnement constaté est identique à celui attendu dans des situations réelles d'utilisation et donc vérifier des scénarios de tests ou des schémas d'utilisation de bout en bout.

Par la suite, nous allons nous concentrer sur les tests unitaires.

1.2 Tests unitaires

Le point important à comprendre sur les tests unitaires est qu'ils testent de petites parties du code (généralement pas plus qu'une méthode appliquée sur un objet). Contrairement aux autres types de tests, les tests unitaires :

— ne communiquent pas avec une base de données ou par le réseau (pas d'action complexe) ;
— ne génèrent ni ne modifient de fichiers (mais ils peuvent en lire) ;
— peuvent être lancés en même temps que d'autres tests unitaires.

L'objectif d'un test unitaire est de permettre au développeur de s'assurer qu'une unité de code ne comporte pas d'erreurs. Dans un test de ce type, une petite partie de code est exécutée. Dans ce cours, on considérera que l'unité de code est la méthode. La plupart des tests consisteront donc à tester que le comportement d'une méthode est bien celui qu'on a défini dans la spécification de la méthode. Tester revient à vérifier que sur certains cas choisis (c'est généralement impossible de tester tous les cas, car ils sont trop nombreux), la méthode produit bien le résultat attendu. Considérons une méthode `void sort()` s'appelant sur une liste d'entiers et qui trie les entiers de la liste par ordre croissant. Tester cette méthode revient à vérifier que pour certaines

listes, la liste après l'appel à la méthode `sort` contient bien les mêmes entiers que la liste de départ dans l'ordre croissant. Il est impossible de tester toutes les listes possibles, car il y en a une infinité (si on ne considère pas les limites de mémoire). Même dans les cas où on aurait un nombre fini de possibilités, il y en a souvent trop pour pouvoir tout tester dans un temps raisonnable.

Vous avez sans doute déjà testé votre code à la main (donc de manière non-automatique) en lançant le code et vérifiant votre code soit via des outils de débogage (avec des points de contrôles pour exécuter le code pas à pas) ou bien directement en vérifiant le bon comportement de votre programme via une utilisation normale de celui-ci. Ici, on peut se concentrer sur des tests automatisés. Un test sera donc un programme qui va appeler une méthode et vérifier que son comportement respecte les spécifications (le contrat de la méthode) via des assertions. Concrètement, un test, c'est du code qui vérifie que des assertions sont vraies. Généralement, un test unitaire consistera an trois étapes :

— créer un objet ;
— appeler une méthode sur cet objet ;
— vérifier que le résultat est bien le résultat attendu.

Bill Wake, auteur de *Refactoring Workbook*, a inventé le terme les trois "A" pour décrire ces étapes : *Arrange, Act, Assert* que l'on pourrait traduire par organiser, agir, vérifier. Se souvenir des trois "A" vous permet de rester concentré sur l'écriture d'un test unitaire efficace. Les tests produits via cette méthode sont reproductibles (pouvant être répétés avec le même résultat) puisqu'ils vérifient un comportement prévisible. De plus, puisque chaque test crée son objet afin de le tester, il est facile de tous les exécuter simultanément, car ils n'y a pas de dépendances entre les tests. Généralement, ce qu'on va faire, c'est regrouper des tests (par exemple les tests de toutes les méthodes d'une classe) en seule classe de test ce qui va nous permettre de les exécuter ensemble facilement. En effet, si l'objet est dans cet état et que je fais cela, alors cela se produira. Une partie du défi de vérifier du code par le biais de tests unitaires consiste à réduire tous les comportements du système à ces cas ciblés et prévisibles. Toute la difficulté de l'utilisation des tests unitaires est de trouver des moyens d'extraire des tests simples et prévisibles à partir de logiciels complexes.

Les règles de base pour l'écriture de test sont les suivantes.

— Écrire au moins une classe de test par classe à tester. Le nom de la classe de test est généralement le nom de la classe testée suivie de `Test`. Par exemple la classe de test d'une classe `Vector` est appelée `VectorTest`.
— Écrire au moins une méthode de test par cas à tester. Le nom de la méthode de test doit indiquer le comportement de l'objet qui est testé. Une manière répandue d'écrire le nom d'une méthode de test est de construire le nom de la méthode de test en mettant `test`, suivi du nom de la méthode, suivi d'un `_` et du comportement testé. Par exemple, un test vérifiant le comportement d'une méthode `withdraw` (retrait) sur un compte en banque à découvert (*overdrawn*) pourra être nommée `testWithdraw_Overdrawn`.
— Pour une méthode à tester, il faut tester :
  — **les cas normaux**, utilisation naturelle de la méthode sur une donnée naturelle, par exemple un retrait d'argent d'un montant strictement positif ;
  — **les cas limites**, utilisation de la méthode sur une donnée "étrange", par exemple par exemple un retrait d'argent d'un montant égal à 0 ;

— **les cas anormaux**, vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés, par exemple un retrait d'argent d'un montant strictement négatif.

1.3 JUnit et assertJ

Afin d'automatiser les tests et de faciliter leur écriture, vous allez utiliser JUnit 5 avec assertJ. Junit est le framework de test unitaire pour Java le plus utilisé alors qu'*AssertJ* est une bibliothèque permettant de faciliter l'écriture d'assertions. Une bonne pratique à respecter est de séparer le code de test du code principal (code du logiciel principal aussi appelé code de production). Comme nous l'avons déjà indiqué précédemment, nous allons travailler avec le moteur de production `Gradle`. Dans un projet `Gradle`, le code de production en Java est dans le répertoire `src/main/java` et le code de test est dans le répertoire `src/test/java`. Il est nécessaire de séparer les tests du code de production car :

— on ne donne pas l'accès au code de test au client par exemple
— les tests ont un rôle spécifique différent du code de production

La commande `gradle test` (ou `./gradlew test` si vous utilisez le *wrapper* de Gradle) lancé à la racine de votre projet permet de lancer tous vos tests.

En JUnit 5, une méthode de test de base :

— a un modificateur d'accès `default` (pas de modificateur d'accès) ;
— est annotée avec `@Test` (à mettre avant la déclaration de la méthode) ;
— ne prend aucun paramètre ;
— ne renvoie rien (retour de type `void`) ;
— contient des assertions `assertThat` qui lèvent une `assertionError` si elles sont fausses (échec du test).

Il existe des méthodes de test plus complexes (comme des méthodes de tests paramétrés) mais on en n'expliquera pas leur syntaxe durant cet enseignement. Le code d'une classe de test avec `JUnit 5` et `AssertJ` aura donc le format suivant :

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class NameTestedClassTest {
  @Test
  void testNameTestedMethod_BehaviorTested(){
    // code containing assertions to test nameTestedMethod
  }
}
```

L'import de la première ligne permet d'utiliser l'annotation `@Test` alors que l'import de la deuxième ligne permet d'avoir accès aux assertions définies par AssertJ. La documentation de ces assertions est disponible à ce [lien][https://www.javadoc.io/doc/org.assertj/assertj-core/latest/org/assertj/core/api/Assertions.html).

Les assertions les plus utilisées d'AssertJ s'appellent de la manière suivante.

— `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
— `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
— `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` égal : `equals` pour les objets et `==` pour les types primitifs.
— `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$
— `assertThat(object).isNull()` : vérifie que la référence est `null`
— `assertThat(object).isNotNull()` : vérifie que la référence **n'**est **pas** `null`
— `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
— `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
— `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
— `fail(message)` : échoue toujours en affichant message.

Il est possible de capturer une exception avec le code `Throwable thrown = catchThrowable(() -> { /* code that can throw an exception */ })` puis de tester des propriétés sur l'exception. Par exemple, on peut tester le contenu du message avec `assertThat(thrown).hasMessageContaining(text)`.

Il est possible de provoquer l'affichage d'un `message` lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat`. Par exemple, l'assertion suivante `assertThat(1+1).as("One plus one should be two").isEqualTo(2)` afficherait `"One plus one should be two"` en cas d'échec du test.

1.4 Exemples de tests unitaires

Afin d'illustrer le fonctionnement des tests unitaires et la manière de tester son code via ceux-ci, le plus simple est de prendre des exemples de classes dont on va tester le comportement.

1.4.1 Exemple de classe à tester : `RationalNumber`

On va commencer par une classe `RationalNumber` qui permet de représenter des nombres rationnels sous la forme de fraction de deux entiers et d'effectuer des opérations d'addition et de multiplications sur ceux-ci. Le code de la classe est le suivant :

```
public class RationalNumber {
  public final int numerator;
  public final int denominator;

  public RationalNumber(int numerator, int denominator) {
    int gcd = gcd(numerator, denominator);
    this.numerator = numerator / gcd;
    this.denominator = denominator / gcd;
  }

  public RationalNumber add(RationalNumber val) {
    int numerator = (this.numerator * val.denominator)
```

```
      + (this.denominator * val.numerator);
    int denominator = this.denominator * val.denominator;
    return new RationalNumber(numerator, denominator);
  }

  public RationalNumber multiply(RationalNumber val) {
    int numerator = this.numerator * val.numerator
    int denominator = this.denominator * val.denominator;
    return new RationalNumber(numerator, denominator);
  }

  private static int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
  }
}
```

Pour ce premier cas assez simple, nous allons nous contenter de tester le bon comportement des deux méthodes `add` et `multiply` qui permettent respectivement d'ajouter et de multiplier deux nombres rationnels. Cela nous donne le code de test suivant dans une classe de test `RationalNumberTest` :

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class RationalNumberTest {
  @Test
  void testAdd(){
    RationalNumber one = new RationalNumber(1, 1);
    RationalNumber onePlusOne = one.add(one);
    assertThat(onePlusOne.numerator)
            .as("Numerator of one plus one is two.")
            .isEqualTo(2);
    assertThat(onePlusOne.denominator)
            .as("Denominator of one plus one is one.")
            .isEqualTo(1);
  }

  @Test
  void testMultiply(){
    RationalNumber twoThirds = new RationalNumber(2, 3);
    RationalNumber twoThirdsTimesTwoThirds = twoThirds.multiply(twoThirds);
    assertThat(twoThirdsTimesTwoThirds.numerator)
            .as("Numerator of twos thirds times two thirds should be four.")
            .isEqualTo(4);
    assertThat(twoThirdsTimesTwoThirds.denominator)
            .as("Denominator of twos thirds times two thirds should be nine.")
```

```
            .isEqualTo(9);
  }
}
```

On peut remarquer qu'on utilise `as` afin de donner des messages explicites en cas d'échec des tests. Ici, nous n'avons testé les méthodes qu'une fois et donc pour une seule paire de valeurs pour chaque opération. Généralement ce n'est pas suffisant, car on peut avoir la malchance que le test passe pour une valeur possible des objets. Une analogie est de considérer une horloge bloquée qui donne la bonne heure deux fois par jour. Même un code complètement faux peut donner la bonne réponse pour une valeur précise des arguments et de l'objet avec lequel elle est appelée. Il est donc important de tester chaque méthode avec au moins deux valeurs possibles des paramètres.

1.4.2 Exemple de classes à tester : `Box` et `Robot`

On va maintenant considérer une classe `Box` permettant de créer une boite avec un certain poids. Le code de la classe avec la documentation est le suivant :

```
public class Box {
  /**
   * Create a box with the specified weight
   * @param weight the weight of the created box
   */
  public Box(int weight) {
    this.weight = weight;
  }

  /**
   * weight of the box
   */
  private int weight;

  /**
   * @return this box's weight
   */
  public int getWeight() {
    return this.weight;
  }
}
```

Il est assez facile de tester le code de la classe `Box` avec le code suivant :

```
import static org.assertj.core.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BoxTest {
  @Test
```

```
  public void testGetWeight() {
    Box someBox = new Box(10);
    assertThat(someBox.getWeight()).isEqualTo(10);
    Box otherBox = new Box(100);
    assertThat(otherBox.getWeight()).isEqualTo(100);
  }
}
```

Maintenant, on souhaite créer une classe `Robot` permettant d'instancier des robots portant des caisses. La spécification informelle de cette classe est la suivante :

> Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse, il ne peut en prendre une autre.

La classe `Robot` définira donc les éléments suivants :

- `Robot(int maxWeight)` : un constructeur qui permet d'instancier un robot pouvant transporter une boite dont le poids est inférieur ou égal au poids spécifié en argument ;
- `boolean isCarryingABox()` une méthode qui renvoie `true` si le robot porte une caisse et `false` sinon ;
- `boolean takeBox(Box box)` une méthode qui fait transporter par le robot la boite spécifiée en argument à deux conditions : le robot ne doit pas déjà transporter une boite et la boite doit peser moins que le poids que peut transporter le robot ;
- `Box getCarriedBox()` une méthode qui renvoie la boite transportée par le robot.

On va inverser le processus classique de développement en écrivant d'abord les tests. C'est une méthode appelée *Test-First Design*. On va donc commencer par appliquer les trois "A" (*Arrange, Act, Assert*) pour concevoir des tests unitaires :

1. **Arrange** : créer la situation initiale et vérifier les « préconditions ». Il est important de vérifier les préconditions, car cela permet de contrôler que l'état de l'objet est correct avant l'appel à la méthode testée. En effet, un test sur une méthode pourrait échouer à cause d'une erreur de code dans le constructeur de l'objet. Il est donc essentiel dans la mesure du possible de vérifier que l'état de l'objet avant l'appel à la méthode testée est bien celui voulu.
2. **Act** : appeler la méthode testée. Si cette méthode renvoie une valeur, il est important de la stocker dans une variable afin de vérifier que sa valeur est correcte.
3. **Assert** : à l'aide d'assertions, vérifier les postconditions, c'est-à-dire la situation attendue après l'exécution de la méthode. Cela peut nécessiter plusieurs assertions.

On va commencer par tester qu'un robot créé ne porte pas de caisse (deuxième phrase de la spécification).

```
public class RobotTest {
  @Test
  public void notCarryingABoxWhenCreated() {
```

```
    Robot robot = new Robot(15);
    // no carried box ?
    assertThat(robot.isCarryingABox()).isFalse();
  }
  }
```

On va continuer par un test de la méthode `takeBox` sur une boite d'un poids inférieur à la capacité de transport du robot (première phrase de la spécification) :

```
public class RobotTest {
  @Test
  public void robotTakeBox_LightEnoughBox() {
    // initial configuration : a robot and a box
    Robot robot = new Robot(15);
    Box box = new Box(10);
    // precondition : the robot does not carry a box
    assertThat(robot.isCarryingABox()).isFalse();
    // execution of the tested method
    boolean boxTaken = robot.takeBox(box);
    // postcondition : the carried box is the box taken
    assertThat(boxTaken).isTrue();
    assertThat(robot.isCarryingABox()).isTrue();
    assertThat(robot.getCarriedBox()).isSameAs(box);
  }
}
```

On continue avec un test de la méthode `takeBox` pour une boite d'un poids strictement supérieur à la capacité de transport du robot (toujours la deuxième phrase de la spécification) :

```
public class RobotTest {
  @Test
  public void robotTakeBox_TooHeavyBox() {
    Robot robot = new Robot(15);
    Box b = new Box(20);
    // precondition : robot does not carry a box
    assertThat(robot.isCarryingABox()).isFalse();
    // execution of the tested method
    boolean boxTaken = robot.takeBox(b);
    // postcondition : no box is carried
    assertThat(boxTaken).isFalse();
    assertThat(robot.isCarryingABox()).isFalse();
  }
}
```

Finalement, on teste l'appel à `takeBox` sur un robot transportant déjà une boite (troisième phrase de la spécification).

```
  @Test
  public void robotCanTakeOnlyOneBox() {
    Robot robot = new Robot(15);
    Box box1 = new Box(10);
    Box box2 = new Box(4);
    robot.takeBox(box1);
    // precondition : the carried box is box1
    assertThat(robot.getCarriedBox()).isSameAs(box1);
    // execution of the tested method
    boolean boxTaken = robot.takeBox(box2);
    // postcondition: the carried box is not box2 and is box1
    assertThat(boxTaken).isTrue();
    assertThat(robot.getCarriedBox()).isNotSameAs(box2)
            .isSameAs(box1);
  }
```

Maintenant que l'on a écrit les tests pour la méthode `takeBox`, on peut écrire le code de production de la classe. Le fait d'avoir défini clairement le comportement voulu de la méthode (sa spécification) et qu'on a testé celui-ci nous permet de coder cette méthode sans crainte.

```
/**
 * A class to instantiate robots that can carry boxes.
 */
public class Robot {
  /**
   * The maximal weight of a box that can be carried by the robot.
   */
  private final int maxWeight;
  /**
   * The box carried by robot.
   */private Box carriedBox;

  /**
   * Creates a robot that carry a box with a weight up to the specified
   * weight.
   *
   * @param maxWeight The maximal weight of a box that can be
   *                  carried by the robot.
   */
  public Robot(int maxWeight){
    this.maxWeight = maxWeight;
  }

  /**
   * Returns {@code true} if this robot carries a box.
   *
```

```
   * @return {@code true} if this robot carries a box
   */
  public boolean isCarryingABox(){
    return carriedBox != null;
  }

  /**
   * Makes the robot carry the specified box if it does not
   * already carry a box and the box has a weight less or
   * equal than the maximal weight that can be carried by
   * the robot.
   *
   * @param box the box to be taken by the robot
   * @return {@code true} if the box was taken by the robot
   */
  public boolean takeBox(Box box){
    if(isCarryingABox() || box.getWeight() > maxWeight)
      return false;
    carriedBox = box;
    return true;
  }

  /**
   * Returns the box carried by the robot.
   * @return the box carried by the robot.
   */
  public Box getCarriedBox() {
    return carriedBox;
  }
}
```

1.4.3 Exemple de classe à tester : `Emails`

Pour donner un dernier exemple, on va donner le code d'une classe `Emails` qui permet d'obtenir les noms d'utilisateurs à partir d'un texte contenant des adresses mails. Le nom d'utilisateur d'une adresse mail est la partie du texte contenu avant l'arobase (par exemple `arnaud.labourel` pour l'adresse arnaud.labourel@univ-amu.fr) et on considère que tout caractère différent d'un lettre, d'un chiffre ou d'un point sépare les adresses.

La classe `Emails` définira donc les méthodes suivantes :

- `Emails(String text)` : un constructeur qui permet d'instancier des emails à partir d'un texte
- `List<String> userNames()` : une méthode qui renvoie la liste des noms d'utilisateurs des emails.

On va commencer par tester la méthode `userNames` sur un cas assez simple qui correspond à une utilisation sur des données normales.

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class EmailsTest{
  @Test
  public void testUserNames_NormalNames() {
    Emails emails =
      new Emails("foo bart@cs.edu xyz marge@ms.com baz");
    assertThat(emails.getUserNames())
      .containsExactly("bart", "marge");
  }
```

Une fois ce premier test écrit, on peut s'attaquer au code de la classe `Emails` :

```
public class Emails {
  private String text;

  public Emails(String text) {
    this.text = text;
  }

  public List<String> userNames() {
    int pos = 0;
    List<String> users = new ArrayList<String>();
    for(;;) {
      int atIndex = text.indexOf('@', pos);
      if (atIndex == -1) break;
      String userName = userName(atIndex);
      if (userName.length() > 0) users.add(userName);
      pos = atIndex + 1;
    }
    return users;
  }

  private String userName(int atIndex) {
    int back = atIndex - 1;
    while (back >= 0 &&
            (Character.isLetterOrDigit(text.charAt(back))
            || text.charAt(back) == '.')) {
      back--;
    }
    return text.substring(back + 1, atIndex);
  }
}
```

Afin de vérifier que notre code est correct, on peut maintenant le tester sur des adresses dont les noms d'utilisateurs

sont étranges (composé que d'une lettre). Ici, c'est assez différent de ce que l'on a fait précédemment avec les tests sur les robots dans le sens où on teste la même méthode avec des cas potentiellement de plus en plus complexes à gérer (et non pas un comportement différent pour une autre situation).

```
public class EmailsTest{
  @Test
  public void testUserNames_NamesWithOneCharacter() {
    Emails emails = new Emails("x y@cs 3@ @z@");
    assertThat(emails.getUserNames())
      .isNotEmpty();
      .containsExactly("y", "3", "z");
  }
}
```

Finalement, on peut aussi tester notre méthode sur des cas limites, c'est-à-dire des adresses emails créées à partir de texte vide ou ne contenant aucune adresse.

```
public class EmailsTest{
  @Test
  public void testUserNames_NullNames() {
    Emails emails = new Emails("no emails here!");
    assertThat(emails.getUserNames()).isEmpty();
    emails = new Emails("@@@");
    assertThat(emails.getUserNames()).isEmpty();
    emails = new Emails("");
    assertThat(emails.getUserNames()).isEmpty();
  }
}
```

1.5 Test unitaires (version courte)

En résumé, ce que vous devez retenir sur les tests sont les points suivants :

- Il est essentiel de tester son code : cela fait gagner énormément de temps en cas d'erreur ;
- principe des trois "A" (*Arrange, Act, Assert* pour concevoir un test :
  1. ***Arrange** : créer la situation initiale et vérifier les « préconditions » ;
  2. ***Act** : appeler la méthode testée ;
  3. ***Assert** :à l'aide d'assertions, vérifier les « postconditions » = situation attendue après l'exécution de la méthode
- plusieurs méthodes de tests peuvent être nécessaires pour tester la correction d'une méthode (une méthode par comportement possible de la méthode en fonction de l'état de l'objet) ;
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important de tester tous les types de cas :
  - cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
  - cas limites (utilisation de la méthode sur une donnée "étrange")

— cas anormaux (vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés)

1.6 La suite : le TDD (Test Driven Development)

1.6.1 Définition du TDD

Le *Test-Driven Development (TDD)*, ou développement piloté par les tests, est une méthode de développement de logiciel qui consiste à concevoir un logiciel par des itérations successives très courtes, telles que chaque itération est accomplie en formulant un sous-problème à résoudre sous forme d'un test avant d'écrire le code source correspondant, et où le code est continuellement remanié dans une volonté de simplification. D'une certaine manière, le TDD est la suite logique *Test-First Design*. La différence est qu'au lieu d'écrire tous les tests pour ensuite écrire tout le code de production, le développeur écrit d'abord un test simple qui échoue (car le code de production correspondant n'existe pas) pour ensuite écrire le code de production qui permet au test de passer et répète ce processus en rajoutant du code de test et de production tant que le code ne satisfait pas à la spécification du logiciel.

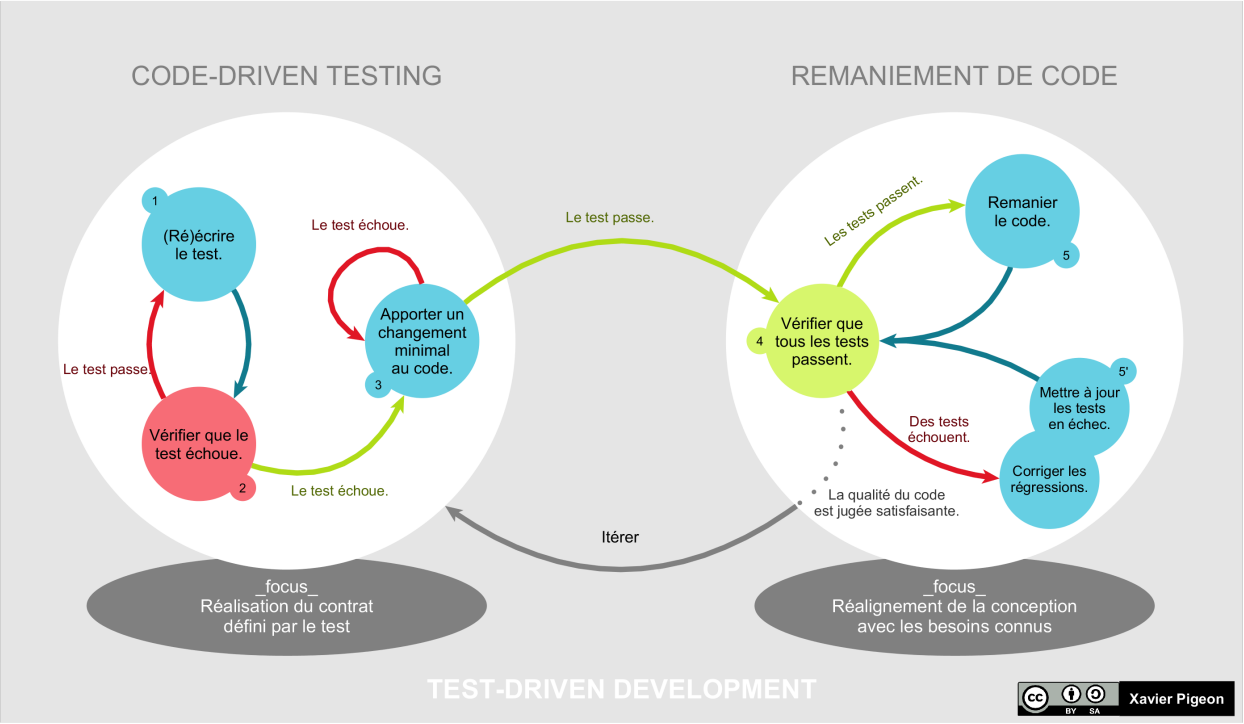
Les trois lois à respecter pour suivre la méthodologie du TDD telle que définies par Robert C. Martin dans « Professionalism and Test-Driven Development » sont les suivantes.

1. Écrivez un test qui échoue avant d'écrire le code de production correspondant.
2. Écrivez une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Écrivez le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Le processus préconisé par la méthodologie TDD comporte cinq étapes.

1. Écrire un seul test qui décrit une partie du problème à résoudre. Il s'agit ici d'écrire un test très simple. La quantité de code de test ajouté doit être minime et le test ne doit vérifier qu'un petit ajout de fonctionnalité dans le code existant. De manière idéale, l'ajout ne devrait consister qu'en une seule ligne de code.
2. Vérifier que le test échoue, autrement dit qu'il est valide, c'est-à-dire que le code se rapportant à ce test n'existe pas encore.
3. Écrire juste assez de code pour que le test réussisse. Là aussi, le but est de rajouter la plus petite quantité de code possible afin de passer le test.
4. Vérifier que le test passe, ainsi que les autres tests existants. Il est important de s'assurer que le code ajouté n'a pas compromis les fonctionnalités existantes.
5. Remanier le code, c'est-à-dire l'améliorer sans en altérer le comportement, qu'il s'agisse du code de production ou du code de test. L'objectif est de simplifier le code le plus possible afin de le rendre le plus lisible possible. Cela passe en autre par la suppression éventuelle de code dupliqué.

Ce processus est répété en plusieurs cycles, jusqu'à résoudre le problème d'origine dans son intégralité. Ces cycles itératifs de développement sont appelés les micro-cycles de TDD. Ce processus est détaillé dans la figure ci-dessous qui a été produite par Xavier Pigeon.



1.6.2 Avantages du TDD

Dans le TDD, contrairement aux autres approches les tests font partie intégrante du processus d'écriture du code du logiciel. Avec le modèle d'organisation appelé cycle en V, les tests sont produits après l'étape d'implémentation et donc après l'écriture du code principal. Le TDD implique non seulement de commencer l'écriture du code par un test, mais impose aussi un aller-retour constant entre code de production et code de test. Les tests dans le processus TDD permettent d'explorer et de préciser le besoin, puis de spécifier le comportement souhaité du logiciel en fonction de son utilisation, avant chaque étape de codage. Le logiciel ainsi produit est tout à la fois pensé pour répondre avec justesse au besoin et conçu pour le faire avec une complexité minimale. On obtient donc un logiciel mieux conçu, mieux testé et plus fiable, autrement dit de meilleure qualité.

Quand les tests sont écrits après l'écriture du code, les choix d'implémentation contraignent leur écriture. Dans certains cas cela peut rendre le code difficile, voire impossible à tester (ou en tout cas trop coûteux en termes de temps de développement). Le processus de TDD qui impose d'écrire les tests d'abord force le développeur à faire des choix d'implémentation facilitant les tests. Cette propriété de testabilité du code favorise une meilleure conception ce qui permet d'éviter des erreurs de conception courantes.

Une autre propriété importante obtenue par le respect du processus de TDD est que chaque petite partie du code est associé à un test. Il est donc normalement facile d'identifier le problème dans le cas d'une régression. Dans le domaine du logiciel, une régression correspond au fait de perdre le bon fonctionnement d'un logiciel après une mise à jour. Ici, ce terme désigne le fait de perdre le comportement attendu du code suite à une réécriture de celui-ci. Dans le cadre du TDD (en supposant que les tests couvrent bien le code), une régression doit entraîner un échec à au moins un test. Si les tests sont bien nommés, il est facile de retrouver la partie du

code principale responsable de la régression surtout que si la méthodologie TDD a été respectée, les ajouts dans le code principal depuis la dernière fois où les tests passaient sont minimaux. C'est en cela que les tests déjà écrits constituent un filet de sécurité contre des accidents de parcours où l'on perdrait le lien entre changement et régression. Ce filet de sécurité permet d'envisager avec sérénité n'importe quelle modification du code, qu'il s'agisse d'une transformation (modification qui affecte le comportement du logiciel) ou d'un remaniement (modification qui n'altère pas le comportement, mais par exemple sa lisibilité).

Pour résumer, le TDD fait gagner en productivité de plusieurs façons.

— Le TDD permet d'éviter des modifications de code sans lien avec le but recherché, car on se focalise à chaque cycle sur la satisfaction d'un besoin précis, en conservant le cap du problème d'ensemble à résoudre. Cela permet d'éviter un écueil classique du développeur qui code des fonctionnalités dont il n'aura jamais besoin : voir principe YAGNI *You Ain't Gonna Need It*.
— Le TDD permet d'éviter les accidents de parcours, où des tests échouent sans qu'on puisse identifier le changement responsable, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.
— Le TDD permet de s'approprier plus facilement n'importe quelle partie du code en vue de le faire évoluer, car chaque test ajouté dans la construction du logiciel explique et documente le comportement du logiciel.

Pour finir, la méthodologie TDD est très puissante, mais peut être compliquée à appliquer si vous n'avez jamais écrit de tests automatiques au préalable. Elle pose néanmoins de bonnes bases sur comment un développeur professionnel devraient produire du code. Le temps qui peut sembler être perdu à écrire une quantité pouvant paraître comme énorme de tests est en fait très souvent du temps gagné par la suite lorsqu'un problème est découvert dans le code.