Errata 1

Les tests

« J'ai fait tourner le programme, il fonctionne bien. »

Tous les programmes contiennent des erreurs, des *bugs*, des défauts... Ces « *bugs* » entrainent, au mieux, un inconfort pour l'utilisateur ou l'utilisatrice. Ils peuvent occasionner une perte de temps, d'argent, de données, de matériel... ou, pire, un danger pour la vie humaine dans un environnement industriel par exemple.

Le travail de développeur ou de la développeuse est de réduire le nombre d'erreurs dans son programme. Pour ce faire, il ou elle le **testera** abondamment. Ce chapitre s'intéresse aux tests.

1.1 Les types d'erreurs et de tests

Les défauts d'un programme peuvent être de différentes sortes :

1. les erreurs de compilation ;
   Ces erreurs sont détectées par le compilateur et rapidement corrigées.
2. les erreurs d'exécution
   ... et le programme s'arrête ;
   ... et le programme ne fournit pas les bonnes valeurs ;
   Ces erreurs sont difficiles à corriger. Elles nécessitent que le programme soit **testé** avec un nombre suffisant de situations différentes. Elles peuvent être dues à des erreurs de programmation mais aussi à des erreurs dans nos algorithmes.
3. les erreurs de programmation qui entrainent une lenteur du programme ou une consommation excessive de mémoire.
   Pour éviter ces erreurs, respecter les bonnes pratiques de développement est une première étape qu'il ne faut pas négliger. Ensuite, il faudra détecter les endroits dans le code où les ralentissements ont lieu ou la consommation de mémoire augmente. Il existe des logiciels dédiés à ce genre de cas.

Pour montrer — garantir, prouver — qu'un programme fonctionne bien, il ne suffit pas de montrer qu'il fournit les bons résultats dans certains cas, il faut convaincre, qu'il fournira les bons résultats dans **tous** les cas :

- ▷ les cas généraux ;
- ▷ les cas particuliers ;
- ▷ lors d'une défaillance de l'environnement ;
  Par exemple, si un capteur de distance est défectueux, il faut que la chaine de production s'arrête.
- ▷ lors d'une utilisation non conforme du programme.

Pour répondre à ces attentes, le développeur ou la développeuse sera attentif · ve à utiliser une méthodologie éprouvée dans toutes les étapes de développement de son projet ; de la phase d'analyse au déploiement. Il mettra en place différents types de tests [1] ; des tests unitaires, d'intégration, fonctionnels, de non-régression... Dans ce premier cours de développement, nous nous intéressons aux **tests unitaires**. Pour ces tests unitaires, le développeur ou la développeuse fournira un ensemble de valeurs à tester représentatives et convaincantes.

Définition

Test unitaire Procédure permettant de tester le bon fonctionnement d'un module, d'une méthode. En fonction des paramètres qu'elle reçoit en entrée, la méthode fournit elle les bons résultats ?

L'idée sous-jacente des tests unitaires est que si chaque partie est correcte, l'ensemble sera correct.

Pour tester régulièrement — très régulièrement — notre programme nous avons besoin d'outils qui vont automatiser ces tests mais commençons par les planifier.

1.2 Planifier les tests

Dans la méthode de résolution de problèmes que nous avons présenté dans le chapitre **??** p.**??** et dans l'exercice résolu de la section **??** p.**??** nous précisions que l'écriture d'une solution complète dans ce premier cours de développement consistait en :

1. spécifier le problème ;
2. fournir des exemples ;
3. écrire un algorithme ;
4. vérifier les exemples (en traçant l'algorithme) ;
5. écrire un programme correspondant à l'algorithme ;
6. tester le programme et constater qu'il fournit bien les résultats attendus.

Les points 2 et 4 sont les prémisses à l'écriture des tests puisque ces exemples vont nous convaincre que l'algorithme — et le programme — fournit bien les « bons »

1. Voir `https://fr.wikipedia.org/wiki/Test_(informatique)`

résultats. Ces exemples constituent le **plan de tests** que nous allons utiliser. Dans ces différents cas, outre les cas généraux, les cas particuliers, les valeurs limites, apparaitront aussi des cas montrant les erreurs de programmation fréquentes. Par exemple :

- ▷ commencer ou arrêter trop tôt ou trop tard une boucle ;
- ▷ ne pas initialiser ou mal initialiser une variable ;
- ▷ confondre $<$ et $\leq$ ou $>$ et $\geq$... voire $<$ et $>$;
- ▷ confondre AND et OR ;
- ▷ mal écrire la négation d'une proposition ;
- ▷ ...

C'est l'expérience qui permettra d'écrire un plan de tests efficace et convaincant.

Remarque Dans certaines méthodologies, la mise en exergue de l'importance des tests se fait en écrivant d'abord les tests avant le code. Voir par exemple Wikipedia [2] ou cette article d'*extreme programming* [3].

Dès lors que le plan de tests est écrit, nous pouvons nous intéresser aux outils qui vont permettre d'automatiser l'exécution des tests et faciliter leur écriture. En effet pour que nos tests soient utiles il faudra tester souvent — idéalement dès qu'une méthode est écrite — et « tout ». Ce sont toutes les méthodes qui sont testées parce que l'écriture d'une méthode pourrait amener une erreur dans une autre méthode. Dans ces conditions, pour que le développeur ou la développeuse teste son programme, ce doit être automatique.

Cette automatisation est fournie par JUnit.

1.3 JUnit

Remarque préalable JUnit n'est pas fourni avec le JDK (*java development kit*), c'est un *framework* de tests indépendant que l'on trouve sur `junit.org`. Il est donc nécessaire de l'installer.

Par contre, il est fourni avec Netbeans.

JUnit est un *framework* simple pour l'écriture de tests répétables.

Pour tester la méthode — `foo` — dans la classe `MyClass`, il faut écrire une classe `MyClassTest` qui contiendra les méthodes de tests et demander à JUnit — dès lors qu'il est installé — d'exécuter cette classe. Un clic dans un IDE ou une commande du style fait l'affaire :

```
$
java org.junit.runner.JUnitCore my.package.MyClassTest
```

terminal

2. `https://fr.wikipedia.org/wiki/Test_driven_development`
3. `http://www.extremeprogramming.org/rules/testfirst.html`

La classe `MyClassTest` contient plusieurs méthodes. Une méthode de test par cas. Une méthode de test :

- ▷ est autonome. Elle ne reçoit pas de paramètre et ne retourne rien ;
- ▷ contient des affirmations qui seront évaluées ;
  La méthode doit me retourner « vrai ». : assertTrue()
  La méthode doit me retourner « faux ». : assertFalse()
  La méthode doit me donner cette valeur si ses paramètres sont ceux-ci. : assertEqauls(<valeur attendue>, <valeur>)
- ▷ est précédée de l'annotation `@Test` permettant de la faire reconnaitre comme un test unitaire ;
- ▷ n'a pas de mot clé static.

Un test aura donc l'allure suivante :

```java
@Test
public void max2_cas1(){
  assertEquals(2, max2(-1, 2));
}
```

java

Après lancement des tests, JUnit fournirait un rapport l'allure suivante :

```
$
JUnit version 4.12
.

Time: 0,012

OK (1 test)
```

terminal

1.4 Exemple

Reprenons l'exemple du calcul du maximum de 2 nombres décrit dans la fiche **??** p.**??**. Voici un plan de tests :

test n°	nb1	nb2	réponse attendue
1	-3	4	4
2	7	4	7
3	4	4	4
4	0	-4	0

Et la classe de tests `MaximumTest` sans les commentaires.

```java
package esi-bru.cours.dev1;

import org.junit.Test;
import static org.junit.Assert.*;

public class MaximumTest {


  @Test
  public void max2_cas1(){
    assertEquals(4, max2(-3, 4));
  }

  @Test
  public void max2_cas2(){
    assertEquals(7, max2(7, 4));
  }

  @Test
  public void max2_cas3(){
    assertEquals(4, max2(4, 4));
  }

  @Test
  public void max2_cas4(){
    assertEquals(0, max2(0, -4));
  }
}
```

java

```
$
JUnit version 4.12
....

Time: 0,012

OK (4 tests)
```

terminal