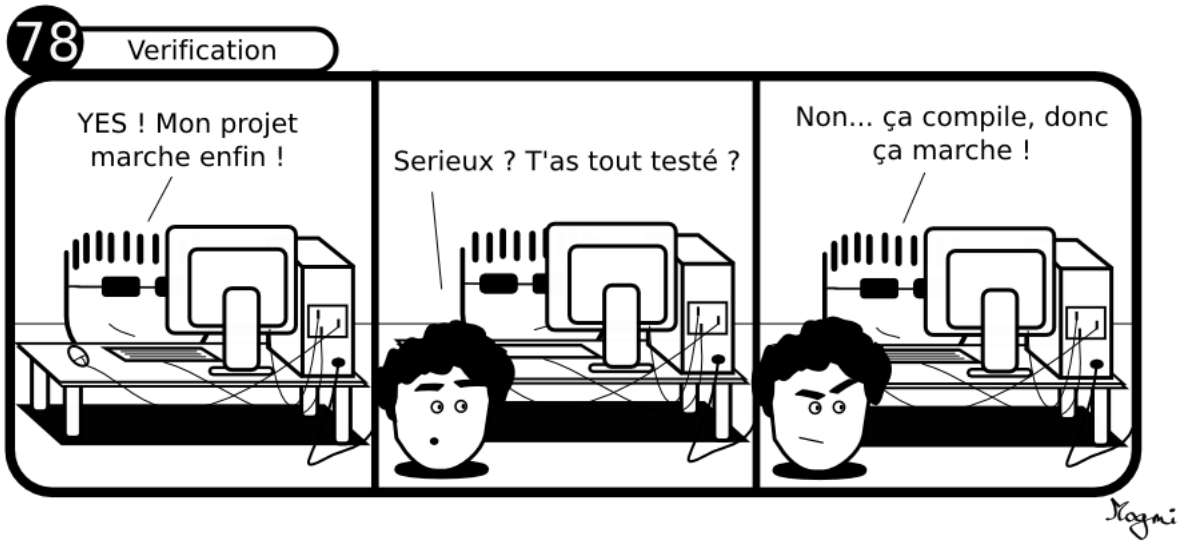
Cours de test et vérification

E.I.S.T.I. ING2 GSI/SIE

Introduction au Test

Introduction au Test



Sommaire

Introduction au Test

- Test: composante essentielle du génie logiciel:
  - Validation: vérifie l'exactitude du logiciel avec exigences du client;
  - Vérification: détermine si le logiciel fonctionne correctement.
- Test dans les faits:
  - À l'origine, la vérification se veut formelle, elle garantie la correction d'un logiciel quelque soit son utilisation (variation des entrées);
  - Dans les faits, en raison de la difficulté, le test consiste simplement à vérifier le comportement du logiciel dans un certain nombre de cas d'utilisation, plus ou moins exhaustifs, visant plus à limiter le nombre de bugs qu'à complètement les éliminer.
- Pourquoi ?
  - Baisser les coûts (oui, oui !)
  - Diminuer les efforts de édveloppement
  - Augmenter la confiance (vis à vis du code des collègues et celle du client !)

Introduction au Test

Historique: les erreurs logiciels (bugs) jalonnent l'histoire de l'informatique et sont autant de d'arguments en faveur du test logiciel.

- calculateurs des voitures (Toyota 2010, Renault 2019);
- logiciels bancaires (banque ZKA 2010, Knight Capital 2012, Louvois (armée) 2013, banque postale 2019);
- logiciels critiques (Ariane 5, Mars Climate Orbiter,...);
- logiciels très critiques (Therac-25, missiles Patriot,...).

Introduction au Test

Citations

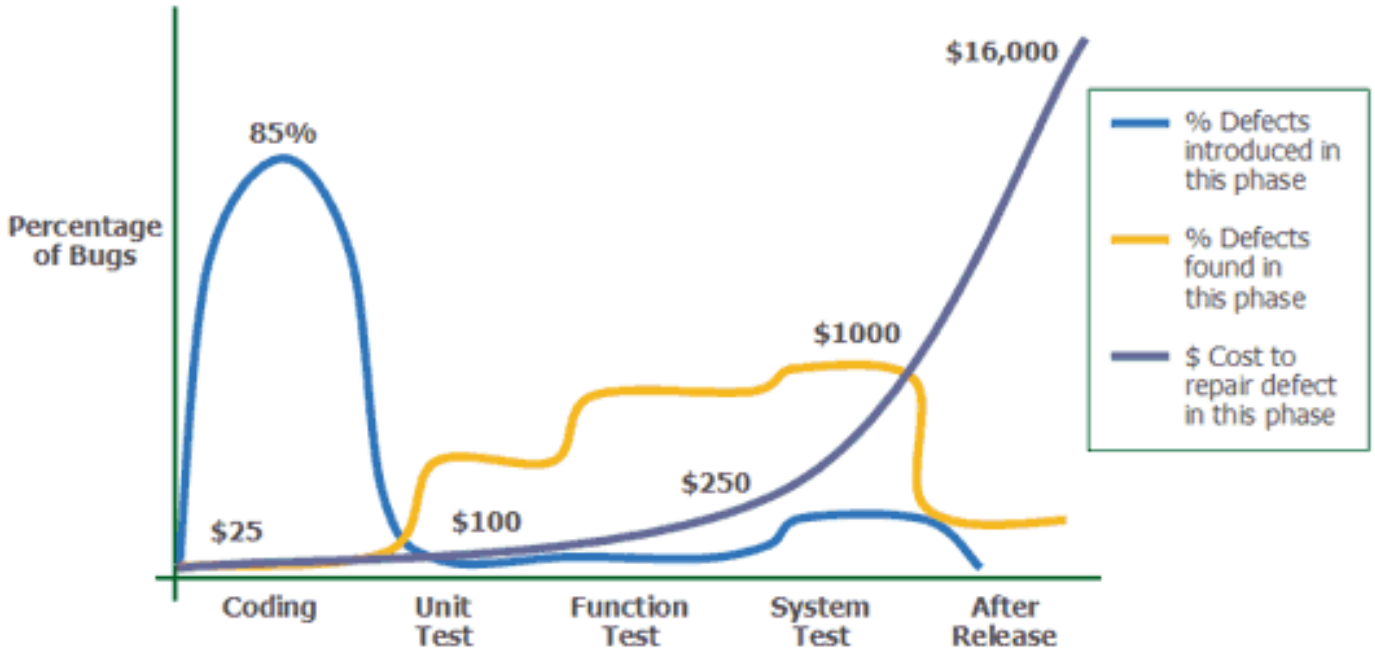
- E.W. Dijkstra (Notes on Structured Programming, 1972) « Testing can only reveal the presence of errors but never their absence. »
- G.J. Myers (The Art of Software Testing, 1979) « Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts. »
- Donald Knuth (1977) « Beware of bugs in the above code ; I have only proved it correct, not tried it. »
- Thomas A. Henzinger (2001) « It has been an exciting twenty years, which has seen the research focus evolve [⋯] from a dream of automatic program verification to a reality of computer-aided [design] debuging. »

Introduction au Test

Quelques chiffres

- Coût des bugs :
  - 64 milliards $/an aux US;
  - impacts humains, environnementaux,...
- Coût du test logiciel :
  - 10 milliards $/an aux US
  - plus de 50 % du développement d'un logiciel critique;
  - en général 30 % du développement d'un logiciel standard.

Coût du Test



source: Applied Software Measurement, Capers Jones, 1996

Introduction au Test

Validation et vérification

- Validation : vérifier que le logiciel est conforme à l'attendu;
- Vérification : prouver que le programme est correct;
- Méthodologie :
  - Tests statiques : revue de code, de spécification
    - But : trouver des défauts de conception, non des défaillances,
    - Outils : Compilateur, Sonar Qube
  - Tests dynamiques : exécution de code sur des scénarios de fonctionnement
    - But : trouver les défaillances par l'exécution du code
    - Outils : Frameworks de test

Introduction au Test

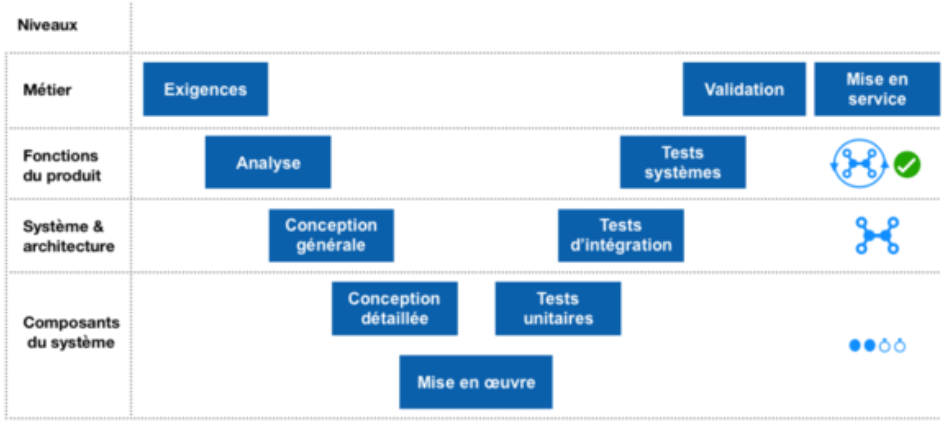
Introduction au Test

Types de tests

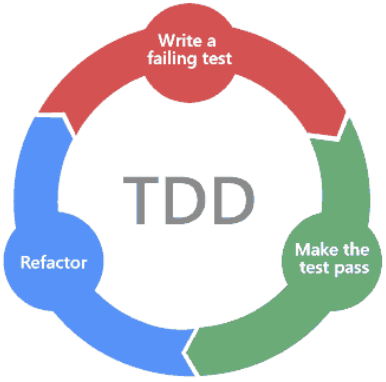
- Tests unitaires : test de chaque module du logiciel indépendamment des autres. La granularité d'un module peut varier selon les cas (fonction, package, classe, ...);
- Tests d'intégration : test des différents modules les uns avec les autres;
- Tests systèmes ou de validation : valider le logiciel par rapport aux spécifications du client en évaluant également des composantes spécifiques telles que la sécurité, la charge ou les performances;
- Tests de régression : vérifier que les mise à jours du logiciel n'ont pas introduits de bugs.

Phases de test

- Cycle en V : après le développement



- Méthodologies Agile : TDD : un test qui échoue appelle un développement



Introduction au Test

Déroulement classique d'un test

1. Choisir un cas de test (CT);
2. Instancier une donnée de test (DT) à partir du CT;
3. Prévoir le résultat attendu à partir de la DT (Oracle);
4. Exécuter le programme sur la DT;
5. Comparer le résultat obtenu avec l'oracle.

Introduction au Test

Familles de test :

- Test en boite blanche (BB) : vous avez accès au code;
- Test en boite noire (BN) : vous n'avez pas accès au code (librairie, code compilé, API...) : documentation !
  On teste les spécifications fonctionnelles selon les cas d'utilisation, les valeurs limites;

Introduction au Test

Sélection des tests en BB :

Déterminer les chemins minimaux entre l'entrée et la sortie du programme afin d'assurer que :

- toutes les conditions d'arrêts des instructions itératives (boucles) ont été testées;
- toutes les branches des instructions conditionnelles (if) ont été testées;
- toutes les structures de données internes (variables locales) ont été testées.

Introduction au Test

Graphe de flot de contrôle (GFC) d'un programme:

- un noeud pour chaque instruction, plus un noeud final de sortie;
- pour chaque instruction du programme, le GFC comporte un arc reliant le noeud de l'instruction au noeud de l'instruction suivante (ou au noeud final si pas de suivant);
- la derniere instruction d'une boucle est reliée au noeud initial de la boucle;
- une conditionnelle est divisée en deux branches qui se relient en sortie de la condition.

Graphe de flot de contrôle

Attention !

- toutes les conditions doivent être décomposées (risque d'explosion cominatoire!);
- toutes les branches doivent être parcourues;
- s'il y a une saisie -> test des valeurs problématiques.

Introduction au Test

Complexité de Mac Cabr

À partir du graphe de Flot:

```
NombreCyclomatique = NbArcs - NbNoeud + 2
```

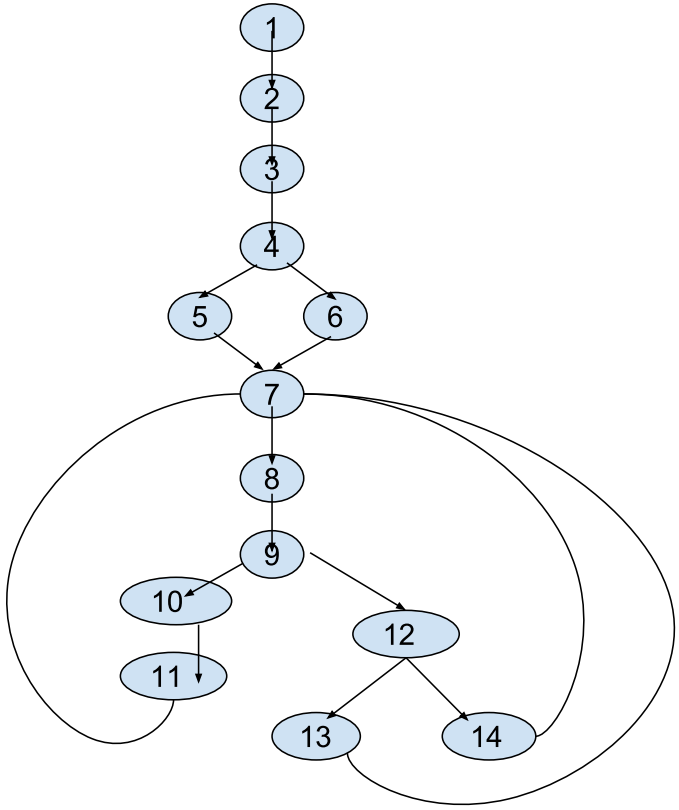
Le nombre cyclomatique détermine le nombre de chemins minimums (égal au nombre de régions du graphe de flot) qu'il va falloir tester (un test par chemin pour tester toutes les possibilités du programme).

Introduction au Test

Exemple

```
void recherche_dico (elem cle, elem t[], int taille, boolean &trouv, int &A) {
    int d, g, m;
1   g = 0;
2   d = taille-1;
3   A = (d+g)/2;
4   if (t[A] == cle)
5       trouv = true;
6   else trouv = false;
7   while (g <= d && !trouv) {
8       m = (d+g)/2;
9       if (t[m] == cle) {
10            trouv = true;
11            A = m; }
12      else if (t[m] > cle)
13               g = m+1;
14           else d = m-1;}}
```

Exemple



Test unitaires avec JUnit

Présentation de JUnit

- Framework d'automatisation des test unitaires;
- Langage JAVA;
- Inventé par Kent Beck et Erich Gamma;
- Version actuelle JUnit5 (utilisation avec Java 8 et plus);
- Version JUnit4 maintenue pour Java 5 et plus (avec les annotations), Junit5 : junit.vintage;
- Version JUnit 3.8 maintenue pour Java 4 et moins (pas d'annotations);
- Utilisable en ligne de commande via une archive jar;
- Intégrable dans les IDE (Eclipse, Intellij, NetBeans, VS Code...) via des plugins.

Test unitaires avec JUnit5

Test unitaires avec JUnit5

Méthodologie des test unitaires

1. Préparation (SetUp) de l'environnement des test (préconditions):
    - initialisations de valeurs, d'objets,
    - activation des logs,
    - ...
2. Execution des cas de tests (CT) prévus lors de l'écriture du programme;
3. Evaluation des résultats ou effets de bords engendrés par l'exécution des CT;
4. Nettoyage (TearDown) de l'environnement si besoin (post-conditions).

Test unitaires avec JUnit5

Implémentation des test unitaires avec JUnit5

L'implémentation se fait au sein d'une classe de test séparée du programme à tester. Les méthodes de cette classe sont décorées par des annotations Java:

- SetUp: annoter les méthodes avec `@BeforeAll`, `@BeforeEach`;
- Tests:
  - annoter les méthodes avec `@Test`,
  - utiliser à l'intérieur des méthodes les assertions,
  - `assertEquals(ValeurAttendue, ValeurCalculee, "texte si failure");`
- Tear Down: annoter les méthodes avec `@AfterAll`, `@AfterEach`.

Test unitaires avec JUnit5

Assertions

Elles permettent de définir les conditions de validité du test (égalité, différence, nullité, valeur booléenne, etc.).

Assomptions

Elles permettent de définir les critères de validité du contexte du test et empécher son exécution s'ils ne sont pas respectés.

Test unitaires avec JUnit5

Gestions des exceptions

- Assertion `assertThrows(Exception.class, fonctionTest);`
- Par exemple :

```
@Test
public void testPlante() {
System.out.println("ça va planter");
int res;
Assertions.assertThrows(java.lang.ArithmeticException.class,
                        () -> {    res = 5 / 0; });
}
```

Test unitaires avec JUnit5

Importation des classes

- Afin de compiler, votre classe de test doit importer les classes JUnit qu'elle utilise;
- Par exemple:

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
```

- Utilisation de la librairie `junit-platform-console-standalone`. Ne pas oublier d'inclure le `.jar` à votre classpath:

```
javac -d out MaClasse.java
javac -d out -cp junit-platform-console-standalone-x.x.x.jar:out MaClasseTest.java
```

Test unitaires avec JUnit5

Exécution des tests et récupération des résultats

- Utilisation de la librairie `junit-platform-console-standalone`
  - exécute l'ensemble des tests;
  - récolte les résultats.
- En ligne de commande:

```
java -jar junit-platform-console-standalone-x.x.x.jar  <options>
```

- Quelques Options marquantes
  - -cp : dossier des classes, jar externes,
  - --scan-classpath : cherche la classe de test dans le classpath,
  - -c : spécifier une classe de test,
  - -t/T : inclure/exclure un Tag,
  - ...
- Par défaut, les résultats sont affichés dans la console.

Test unitaires avec JUnit5

Exemple: division (implémentation)

- Fichier MyClass.java

```
public class MyClass {
    public int divide(int x, int y) {
        return x / y;
    }
}
```

- Fichier MyClassTest.java

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;

public class MyClassTest {
    @Test
    @DisplayName("Mon cas de test identité 10/1=10")
    public void testDivide() {
        MyClass tester = new MyClass();
        Assertions.assertEquals(10, tester.divide(10, 1), "10/1=1 FAILED");
    }
}
```

Test unitaires avec JUnit5

Exemple: division (exécution)

- Compilation:

```
javac -d out MyClass.java
javac -d out -cp junit-platform-console-standalone-x.x.x.jar:out MyClassTest.java
```

- Exécution:

```
java -jar junit-platform-console-standalone-x.x.x.jar -cp out --scan-classpath
```

- Résultat console:

```
.
├── JUnit Jupiter ✔
│   └── TestDiv ✔
│       └── Mon cas de test identité 10/1=10 ✔
└── JUnit Vintage ✔
...
```

Test unitaires avec JUnit5

Exemple: division (exécution erreur)

- Fichier MyClass.java

```
@Test
@DisplayName("Mon cas plante")
public void testBindon() {
    Assertions.assertEquals(2, testeur.divide(2, 5), "2/5=2 FAILED");
}
```

- Résultat d'exécution:

```
├── JUnit Jupiter ✔
│   └── TestDiv ✔
│       └── Mon cas plante ✘ 2/5=2 FAILED ==> expected: <2> but was: <0>
└── JUnit Vintage ✔

Failures (1):
  JUnit Jupiter:TestDiv:Mon cas plante
    MethodSource [className = 'TestDiv', methodName = 'testBindon', methodParamete
    => org.opentest4j.AssertionFailedError: 2/5=2 FAILED ==> expected: <2> but was
       [...]
```

Test unitaires avec JUnit5

Autres fonctionnalités simples

- Désactivation : `@Disabled`;
- Ordre : `@Order(nb)`(Annotation de méthode), `@TestMethodOrder(MethodOrderer.OrderAnnotation.class)` (Annotation de classe);
- Nommage : `@DisplayName("Nom")`;
- Assertions multiples : `assertAll`.

```
public void testPlusieurs() {
    Assertions.assertAll("Divisions classiques",
        () -> Assertions.assertTrue(testeur.divide(10, 5) == 2, "10/5=2 FAILED"),
        () -> Assertions.assertEquals(testeur.divide(10, 3), 3, "10/3=3 FAILED"));
}
```

JUnit5 : Fonctionnalités avancées

Tests paramétrés

Automatiser un ensemble de paramètres pour une même fonctionnalité, provenant de multiples sources.

- Annotation de test : `@ParameterizedTest`;
- Annotations de fournisseur de données : `@ValueSource`, `@MethodSource`, `@CSVSource`, `@CSVSourceFile`, ...

```
public static List<Object[]> provideData() {
    return Arrays.asList(new Object[][]{{7,1,7},{1,7,0},{7,7,1},
                                {7,3,2},{10,5,2},{10,3,3},{0,7,0}});
}

@ParameterizedTest
@DisplayName("Tests paramétriques")
@MethodSource("provideData")
public void testall(int dividende, int diviseur, int result) {
    Assertions.assertEquals(result, testeur.divide(dividende, diviseur));
}
```

JUnit5 : Fonctionnalités avancées

Suites de Tests

Permet de créer des scénarii de test

- Annotations `@Tag`, `@{Include|Exclude}Tags`, `@RunWith`, `@SelectClasses`, `@SelectPackages`, ... ;

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@IncludeClasses(TestDiv.class, TestMult.class)
@IncludeTags({"TestZero"})
public class TestDivSuite {
    // et c'est parti !
}
```

JUnit5 : Fonctionnalités avancées

Répétition de test

Permet de lancer plusieurs répétions d'un même test (code impliquant de l'aléatoire, performances...).

- Annotations : `@RepeatedTest(nb); @RepeatedTest(value = 3, name = RepeatedTest.LONG_DISPLAY_NAME); ....;`

```
@DisplayName("Test répété division")
@RepeatedTest(value = 42, name = RepeatedTest.LONG_DISPLAY_NAME)
void testRepete(RepetitionInfo repInfo) {
  Assertions.assertEquals(2, testeur.divide(7,3),
                  "Test FAILED nb "+repInfo.getCurrentRepetition());
}
```

JUnit5 : Fonctionnalités avancées

Autres fonctionnalités

Junit 5 apporte de nouvelles fonctionnalités que nous n'aborderons pas :

- les tests imbriqués : permet d'inclure une sous classe de test (Nested test, à l'intérieur d'une classe de test);
- les tests dynamiques : tests générés à l'exécution (annotation `@TestFactory`);
- l'injection de dépendances : permet de gérer des paramètres dans les classes de test.

Le java-iste éclairé trouvera des exemples de ces fonctionnalités dans l'EXCELLENT site de Jean-Michel Doudoux

À vous de jouer !