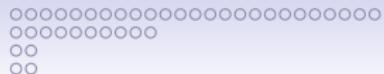


Une initiation à JUnit

Clémentine Nebut

LIRMM / Université de Montpellier 2

Octobre 2012



JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

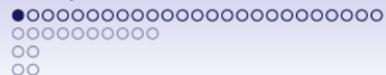
Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

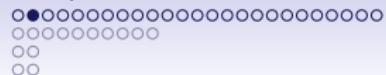
Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



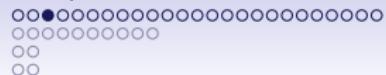
JUnit

- Origine

- Xtreme programming (test-first development), méthodes agiles
- framework de test écrit en Java par E. Gamma et K. Beck
- open source : www.junit.org

- Objectifs

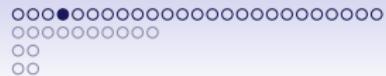
- test d'applications en Java
- faciliter la création des tests
- tests de non régression



Ce que fait JUnit

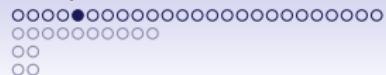
- Enchaîne l'exécution des méthodes de test définies par le testeur
- Facilite la définition des tests grâce à des assertions, des méthodes d'initialisation et de finalisation
- Permet en un seul clic de savoir quels tests ont échoué/planté/reussi

JUnit (et au delà xUnit) est de facto devenu un standard en matière de test



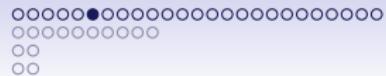
Ce que ne fait pas JUnit

- JUnit n'écrit pas les tests !
- Il ne fait que les lancer.
- JUnit ne propose pas de principes/méthodes pour structurer les tests



JUnit : un framework

- Le framework définit toute l'infrastructure nécessaire pour :
 - écrire des tests
 - définir leurs oracles
 - lancer les tests : principe Hollywood, c'est le code du framework qui appelle celui du reste de l'application
- Utiliser Junit :
 - définir les tests
 - s'en remettre à JUnit pour leur exécution
 - ne pas appeler explicitement les méthodes de test



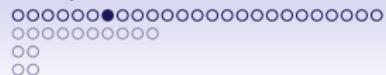
JUnit : versions initiales et versions ≥ 4

Versions initiales

- Paramétrage par spécialisation
- Utilisation de conventions de nommage

Versions ≥ 4

- Utilisation d'annotations
- beaucoup de nouvelles fonctionnalités dans JUnit 4
- attention, la plupart des docs trouvées sur internet se basent sur junit 3
- pas de runner graphique en version 4, laissé au soin des IDEs



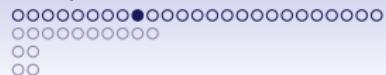
Écriture de test : principe général

- On crée une classe destinée à contenir les tests : la classe de test.
- On y insère des méthodes de test.
- Une méthode de test
 - fait appel à une ou plusieurs méthodes du système à tester,
 - ce qui suppose d'avoir une instance d'une classe du système à tester (la création d'une telle instance peut être placée à plusieurs endroits, voir plus loin),
 - inclut des instructions permettant un verdict automatique : les assertions.



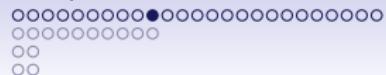
Classe de test

- Contient les méthodes de test
- Est une collection de cas de test (sans ordre)
- peut contenir des méthodes particulières pour positionner l'environnement de test
- En JUnit :
 - JUnit versions <4 : la classe de test hérite de `JUnit.framework.TestCase`
 - JUnit versions ≥ 4 : RAS



Cas de test / méthode de test

- s'intéresse à une seule unité de code/ un seul comportement
- doit rester court
- les cas de test sont indépendants les uns des autres
- Avec Junit, un cas de test \equiv une méthode (méthode de test)
 - JUnit versions <4 : les méthodes de test commencent par le mot test
 - JUnit versions ≥ 4 : annotées @Test
- les méthodes de test seront appelées par Junit, dans un ordre quelconque.



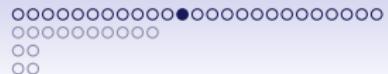
Les méthodes de test

- sont sans paramètres et sans type de retour (logique puisqu'elles vont être appelées automatiquement par JUnit)
- embarquent l'oracle
- i.e. contiennent des assertions
 - à ce stade, x doit valoir 3
 - le résultat de l'appel de cette méthode doit être non nul
 - x doit être plus petit que y



Détails sur les méthodes de test en JUnit ≥ 4

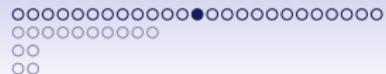
- L'annotation `@Test` peut prendre en paramètre :
 - le type d'exception attendue
`@Test(expected=monexception.class)` Succès si cette exception est lancée.
 - un timeout : `@Test(timeout=10)` (en ms). Fail si la réponse n'arrive pas avant le timeout.
- annotation `@ignore` (paramètre optionnel : du texte) pour ignorer le test



JUnit : un framework

Extraits de code de collecte des méthodes de test, JUnit 3

```
public TestSuite (final Class theClass){  
    ...  
    Method [] = theClass.getDeclaredMethods  
    ...  
}  
private boolean isTestMethod(Method m) {  
    String name= m.getName();  
    Class [] parameters= m.getParameterTypes();  
    Class returnType= m.getReturnType();  
    return parameters.length == 0 && name.startsWith("test")  
        && returnType.equals(Void.TYPE);  
}
```



JUnit : un framework

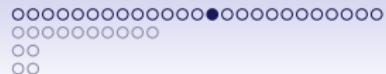
Extraits de code de collecte des méthodes de test, JUnit 3

Dans BlockJUnit4ClassRunner

```
protected List<FrameworkMethod> computeTestMethods() {  
    return getTestClass().getAnnotatedMethods(Test.class);  
}
```

Dans TestClass.java

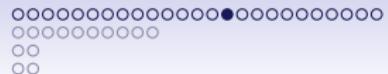
```
public List<FrameworkMethod> getAnnotatedMethods(  
    Class<? extends Annotation> annotationClass) {  
    return getAnnotatedMembers(fMethodsForAnnotations, annotationClass);  
}
```



Les verdicts

Sont définis grâce aux assertions placées dans les cas de test.

- Pass (vert) : pas de faute détectée
- Fail (rouge) : échec, on attendait un résultat, on en a eu un autre
- Error : le test n'a pas pu s'exécuter correctement (exception inattendue, ...)
- En JUnit 4, plus de différence entre fail et error



Exemple en version 4 – classe à tester

<http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial>

```
public class Subscription {  
  
    private int price ; // subscription total price in euro-cent  
    private int length ; // length of subscription in months  
  
    // constructor :  
    public Subscription(int p, int n) {  
        price = p ;  
        length = n ;  
    }  
  
    /**  
     * Calculate the monthly subscription price in euro,  
     * rounded up to the nearest cent.  
     */  
    public double pricePerMonth() {  
        double r = (double) price / (double) length ;  
        return r ;  
    }  
  
    /**  
     * Call this to cancel/nullify this subscription.  
     */  
    public void cancel() { length = 0 ; }  
}
```



Exemple en version 4 – objectif de test

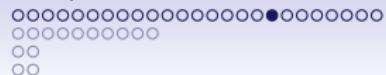
<http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial>

- If we have a subscription of 200 cent for a period of 2 month, its monthly price should be 1 euro, right ?
- The monthly price is supposed to be rounded up to the nearest cent. So, if we have a subscription of 200 cent for a period of 3 month, its monthly price should be 0.67 euro.

Exemple en version 4 – classe de test

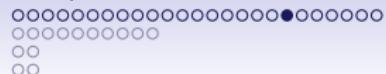
<http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial>

```
import org.junit.* ;  
import static org.junit.Assert.* ;  
  
public class SubscriptionTest {  
  
    @Test  
    public void test_returnEuro() {  
        System.out.println("Test if pricePerMonth returns Euro...") ;  
        Subscription S = new Subscription(200,2) ;  
        assertTrue(S.pricePerMonth() == 1.0) ;  
    }  
  
    @Test  
    public void test_roundUp() {  
        System.out.println("Test if pricePerMonth rounds up correctly...") ;  
        Subscription S = new Subscription(200,3) ;  
        assertTrue(S.pricePerMonth() == 0.67) ;  
    }  
}
```



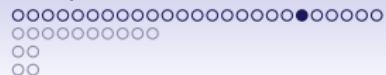
L'environnement de test

- Les méthodes de test ont besoin d'être appelées sur des instances
- Déclaration et création des instances
 - en général, les instances sont déclarées comme membres d'instance de la classe de test
 - la création des instances et plus globalement la mise en place de l'environnement de test est laissé à la charge de méthodes d'initialisation
 - NB : ce n'est pas ce qui a été fait dans l'exemple précédent.



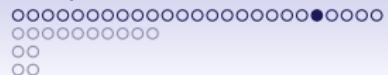
Préambules et postambules

- Méthodes écrites par le testeur pour mettre en place l'environnement de test.
- JUnit 4 : Méthodes avec annotations @Before et @After ; JUnit 3 : Méthodes appelées setUp et tearDown
 - exécutées avant/après chaque méthode de test (l'exécution est pilotée par le framework, et pas le testeur)
 - possibilité d'annoter plusieurs méthodes (ordre d'exécution indéterminé)
 - publiques et non statiques
- Méthodes avec annotations @BeforeClass et @AfterClass (pas en JUnit 3)
 - exécutées avant (resp. après) la première (resp. dernière) méthode de test
 - une seule méthode pour chaque annotation
 - publiques et statiques



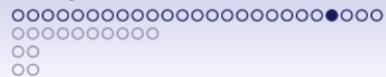
Les assertions (en JUnit 4)

- Permettent d'embarquer et d'automatiser l'oracle dans les cas de test (`adieu`, `println` ...)
- Utilisation de `org.junit.Assert.*`
 - attention, import statique, car les asserts sont des méthodes statiques
 - `import static org.junit.Assert.*;`
- Lancent des exceptions de type `java.lang.AssertionError` (comme les assert java classiques)
- Différentes assertions : comparaison à un delta près, comparaison de tableaux (arrays), ...
- Forte surcharge des méthodes d'assertion.



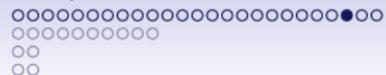
Assert that (nouveauté version 4.4)

- `assertThat([value], [matcher statement]);`
- exemples :
 - `assertThat(x, is(3));`
 - `assertThat(x, is(not(4))));`
 - `assertThat(responseString,
either(containsString("color")).or(containsString("colour")));`
 - `assertThat(myList, hasItem("3"));`
- `not(s), either(s).or(ss), each(s)`
- Messages d'erreur plus clairs
- <http://junit.sourceforge.net/doc/ReleaseNotes4.4.html>



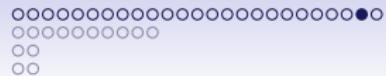
Assumptions (nouveauté version 4.4)

- `AssumeThat(File.separatorChar, is('/'))`
- L'assertion suivante sera ignorée si la supposition n'est pas vérifiée



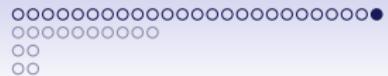
Test paramétré

- Objectif : réutiliser une méthode de test avec des jeux de données de test différents
- Jeux de données de test
 - retournés par une méthode annotée @Parameters
 - cette méthode retourne une collection de tableaux contenant les données et éventuellement le résultat attendu
- La classe de test
 - annotée @RunWith(Parameterized.class)
 - contient des méthodes devant être exécutées avec chacun des jeux de données
- Pour chaque donnée, la classe est instanciée, les méthodes de test sont exécutées



Test paramétré : les besoins

- Un constructeur public qui utilise les paramètres (i.e. un jeu de données quelconque)
- La méthode qui retourne les paramètres (i.e. les jeux de données) doit être statique



Exemple de test paramétré : test de Sum :int sum(int x, int y)

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;
@RunWith(Parameterized.class)
public class TestParamSum {
    private int x;
    private int y;
    private int res;

    public TestParamSum(int x, int y, int res) {
        this.x = x;
        this.y = y;
        this.res = res;}
    @Parameters
    public static Collection testData() {
        return Arrays.asList(new Object[][] {
            { 0, 0, 0 },{ 1, 1, 2 },{ 2, 1, 3 },{10, 9, 19}});}
    @Test
    public void testSum() {
        assertEquals(res,new Sum().sum(x,y));
    }
}
```



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

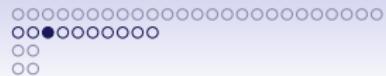
Les types classiques de test

Processus, vocabulaire et difficultés



Classe à tester

```
package calc;
public class Calculator {
    private static int result;// Static variable where the result is stored
    public void add(int n) {
        result = result + n;
    }
    public void subtract(int n) {
        result = result - 1;      //Bug : should be result = result - n
    }
    public void multiply(int n) {}           //Not implemented yet
    public void divide(int n) {
        result = result / n;
    }
    public void square(int n) {
        result = n * n;
    }
    public void squareRoot(int n) {
        for (; ; ) ;                  //Bug : loops indefinitely
    }
    public void clear() {                   // Cleans the result
        result = 0;
    }
    public void switchOn() {// Swith on the screen, display "hello", beep
        result = 0;                  // and do other things that calculator do nowadays
    }
    public void switchOff() { } // Display "bye bye", beep, switch off the screen
    public int getResult() {
        return result;
    }
}
```



Une classe de test

```
import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;
public class CalculatorTest {
    private static Calculator calculator = new Calculator();
    @Before
    public void clearCalculator() {
        calculator.clear();
    }
    @Test
    public void add() {
        calculator.add(1);
        calculator.add(1);
        assertEquals(calculator.getResult(), 2);
    }
    @Test
    public void subtract() {
        calculator.add(10);
        calculator.subtract(2);
        assertEquals(calculator.getResult(), 8);
    }
    @Test
    public void divide() {
        calculator.add(8);
        calculator.divide(2);
        assert calculator.getResult() == 5;
    }
}
```

suite

```
@Test(expected = ArithmeticException.class)
public void divideByZero() {
    calculator.divide(0);
}
@Ignore("not ready yet")
@Test
public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
}
```



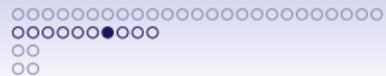
Une autre classe de test

```
import calc.Calculator;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class AdvancedTest extends AbstractParent {
    private static Calculator calculator;
    @BeforeClass
    public static void switchOnCalculator() {
        System.out.println("Switch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
    }
    @AfterClass
    public static void switchOffCalculator() {
        System.out.println("Switch off calculator");
        calculator.switchOff();
        calculator = null;
    }
    @Before
    public void clearCalculator() {
        System.out.println("Clear calculator");
        calculator.clear();
    }
    @Test(timeout = 1000)
    public void squareRoot() {
        calculator.squareRoot(2);
    }
}
```



suite

```
@Test
public void square2() {
    calculator.square(2);
    assertEquals(4, calculator.getResult());
}
@Test
public void square4() {
    calculator.square(4);
    assertEquals(16, calculator.getResult());
}
@Test
public void square5() {
    calculator.square(5);
    assertEquals(25, calculator.getResult());
}
```



suite

```
package junit4;
import org.junit.*;
public abstract class AbstractParent {
    @BeforeClass
    public static void startTestSystem() {
        System.out.println("Start test system");
    }
    @AfterClass
    public static void stopTestSystem() {
        System.out.println("Stop test system");
    }
    @Before
    public void initTestSystem() {
        System.out.println("Initialize test system");
    }
}
```



Avec tests paramétrés

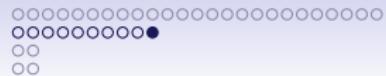
```
import calc.Calculator;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import java.util.Arrays;
import java.util.Collection;
@RunWith(Parameterized.class)
public class SquareTest {
    private static Calculator calculator = new Calculator();
    private int param;
    private int result;
    @Parameters
    public static Collection data() {
        return Arrays.asList(new Object[][]{
            {0, 0}, {1, 1},
            {2, 4}, {4, 16},
            {5, 25},{6, 36}, {7, 49}
        });
    }
    public SquareTest(int param, int result) {
        this.param = param;
        this.result = result;
    }
    @Test
    public void square() {
        calculator.square(param);
        assertEquals(result, calculator.getResult());
    }
}
```



Suite de tests

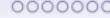
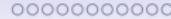
- Rassemble des cas de test pour enchaîner leur exécution

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    CalculatorTest.class ,
    SquareTest.class
})
public class AllCalculatorTests {
```



Conclusion sur JUnit

- Construction rapide de tests
- Exécution rapide
- Très bien adapté pour le test unitaire et test de non régression



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit

xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

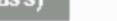
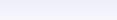
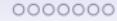
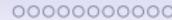
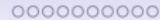
Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

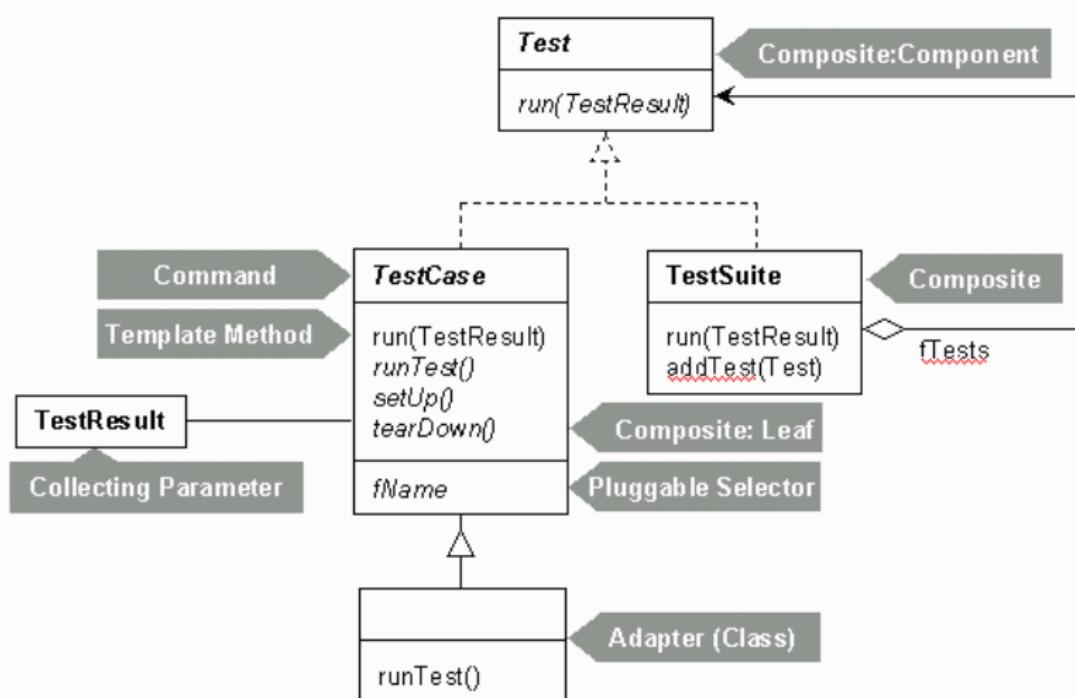
Les techniques

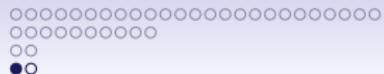
Les types classiques de test

Processus, vocabulaire et difficultés



JUnit 3





Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit

xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

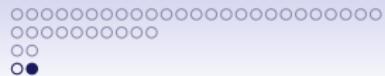
Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



JUnit et les autres

- NUnit -> .net
- PiUnit -> python
- FlexUnit -> Flex
- etc ...



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

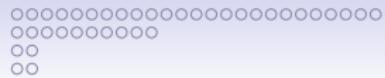
Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

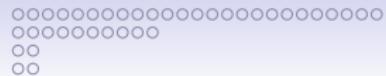
Processus, vocabulaire et difficultés



Le test

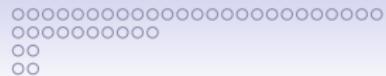
Principe

Essayer pour voir si ça marche ...



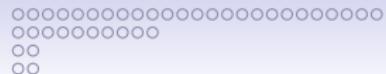
Essayer ...

- Mais au fait comment ça marche ?
 - Démarrage du programme ?
 - Interface graphique ? Textuelle ?
- Quelles entrées ?
 - Données requises ?
- Qu'est-il possible de faire ?
 - Si on veut tout essayer, il faut savoir ce qu'il y a à essayer !
 - Quels enchaînements nécessaires pour essayer une fonctionnalité ?



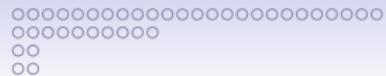
... pour voir ...

- Que peut-on voir ?
 - une couleur dans une interface graphique ?
 - un affichage dans une fenêtre ?
 - la valeur d'une variable ?
 - le résultat d'un calcul intermédiaire ?
- Notion d'observabilité



... si ça marche.

- Comment sait-on que ça marche ?
 - au fait, il doit faire quoi ce programme ?
 - notion de spécifications
 - à partir de ce que l'on peut voir, déterminer si ça marche
 - et si on ne voit pas ce que l'on veut ?
- Et si ça ne marche pas ?
 - Diagnostique
- Et si ça a l'air de marcher ...
 - est-on sûr que ça marche vraiment ?
 - notion de confiance \neq certitude
 - et si c'étaient les tests qui étaient mauvais ou insuffisants ?
 - qualité des tests, critère d'arrêt

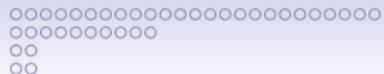


Vers une définition ...

Définition de Myers, 1979

Testing is the process of executing a program with the intent of finding errors. [G. Myers. The Art of Software Testing. 1979]

- Reste à savoir ce qu'on teste et ce qu'est une erreur ...



Qu'est ce qu'on teste ? (quelles propriétés)

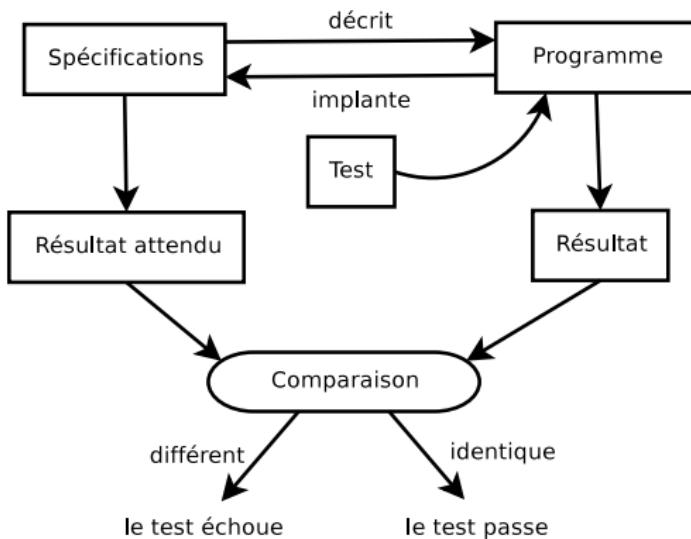
Différentes propriétés à tester

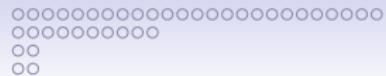
- satisfaction des fonctionnalités requises
- qualité de service (temps de réponse, utilisation mémoire, ...)
- robustesse
- sûreté de fonctionnement
- utilisabilité

On teste vis à vis d'une spécification !

Pour déterminer si on a détecté un problème, toutes les propriétés doivent être spécifiées

Le verdict ...

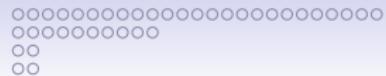




Des alternatives ?

Le test et les méthodes formelles

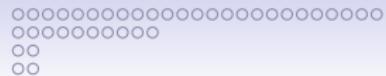
- On fait des spécifications formelles (au fait, c'est quoi ?), puis :
 - on dérive automatiquement le code
 - ou on écrit un code et on le prouve correct



Des alternatives ?

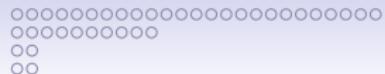
Le test et les méthodes formelles

- Mais des problèmes de fond ...
 - adéquation entre les spécifications et le cahier des charges ?
 - on prouve des propriétés, on ne prouve pas dans l'absolu ... Et si on a oublié des propriétés à prouver ?
- Et des problèmes plus terre à terre ...
 - formation d'experts en méthodes formelles
 - coût de mise en place
 - et quand on modifie le cahier des charges, il faut tout refaire
 - pas de méthodes, d'outils de développement adaptés aux méthodes formelles



Test et V&V

- V&V = Vérification et Validation
- Le test est une technique particulière de V&V
- Si le test est indispensable dans la quasi-totalité des projets, il n'est pas l'unique procédé de V&V à mettre en œuvre. Par exemple :
 - revues techniques
 - analyseurs statiques



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

Le test et les autres procédés de V&V

Quels tests pour quelles erreurs ?

Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés



Différents tests

Plusieurs niveaux (échelles)

- Unitaire
- Intégration
- Système
- Acceptation (ou recette)

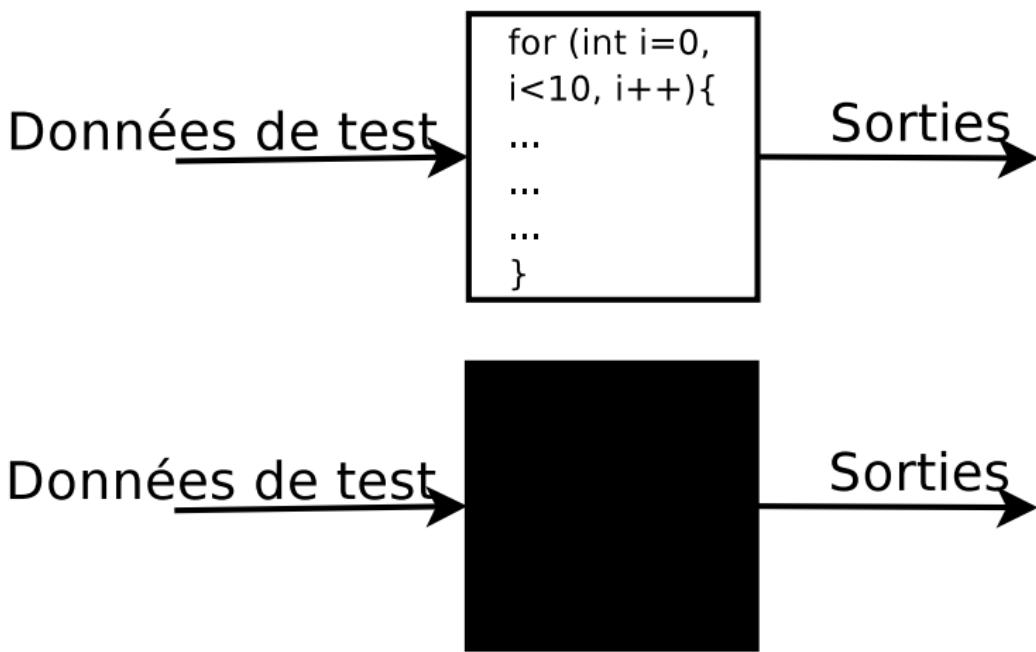
Différents niveaux d'accessibilité

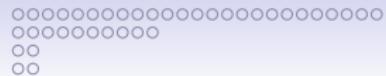
- Test boîte noire (souvent fonctionnel)
- Test boîte blanche (souvent structurel)
- Test boîte grise ?

Plusieurs types classiques

- test fonctionnel
- test de non-régression

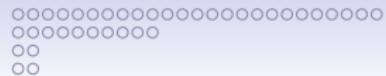
Boîte blanche et boîte noire





Le test fonctionnel

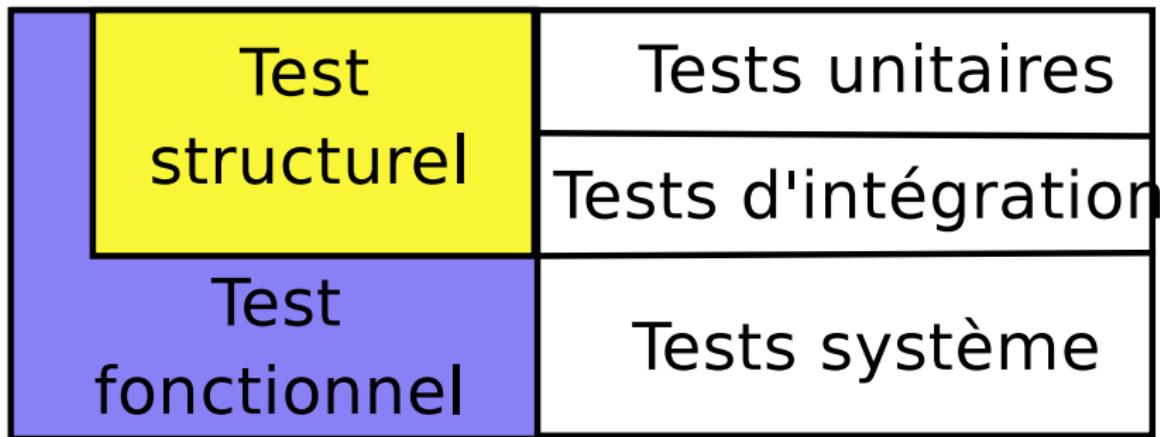
- Technique en général boîte noire
- On se base sur un modèle du programme issu des spécifications
 - informelles (ex. description en langage naturel)
 - semi-formelles (ex. modèles UML)
 - formelles (machines B, IOLTS, ...)



Le test structurel

- Technique boîte blanche
- On se base sur un modèle du code source du programme
 - Le modèle est une représentation de la structure
 - On utilise beaucoup la théorie des graphes pour couvrir le modèle notamment

Techniques et échelles





Complémentarité fonctionnel / structurel

```
function sum (x,y : integer) : integer;  
begin  
  if (x = 600) and (y = 500) then sum := x-y  
  else sum := x+y;  
end
```

- Une technique fonctionnelle a peu de chances de trouver l'erreur
- Une technique structurelle trouvera facilement la donnée de test ($x=600, y=500$)



Complémentarité fonctionnel / structurel

```
prod(int i, inj)
    int k;
    if (i==2){
        k:=i<<1;//décalage à gauche, multiplication par 2
    } else
        faire i fois l'addition de j
    return k;
```

Spécification : renvoie le produit de i par j



Complémentarité fonctionnel / structurel

```
prod(int i, inj)
    int k;
    if (i==2){
        k:=i<<1;//décalage à gauche, multiplication par 2
    else
        faire i fois l'addition de j
    return k;
```

Spécification : renvoie le produit de i par j

Fonctionnel

On choisit ($i=0, j=0$) $\rightarrow 0$ et ($i=10, j=100$) $\rightarrow 1000 \longrightarrow \text{OK}$



Complémentarité fonctionnel / structurel

```
prod(int i, inj)
    int k;
    if (i==2){
        k:=i<<1;//décalage à gauche, multiplication par 2
    else
        faire i fois l'addition de j
    return k;
```

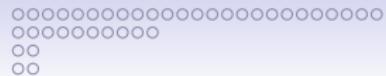
Spécification : renvoie le produit de i par j

Fonctionnel

On choisit ($i=0, j=0$) $\rightarrow 0$ et ($i=10, j=100$) $\rightarrow 1000$ —> OK

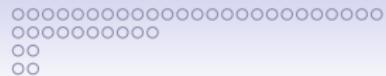
Structural

On choisit au moins une donnée qui passe par le cas $i=2$:
 $(i=2, j=0) \rightarrow 1$ —> NOK



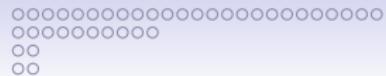
Le test fonctionnel

- On cherche à savoir
 - si toutes les fonctionalités requises sont présentes ...
 - ... et correctes



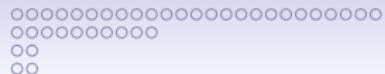
Le test de robustesse

- On cherche à savoir si le système est robuste
- Par exemple,
 - on entre des entrées invalides
 - on ferme violemment le programme
 - on jette l'ordinateur par la fenêtre (test militaire)
- Et on regarde comment le programme se comporte ...



Le test de non-régression

- On cherche à savoir si on n'a pas perdu des propriétés en cours de route ...
 - après un ajout de fonctionnalité
 - après la correction d'une erreur
 - après une optimisation
- En général, on relance les tests qui passaient précédemment
 - d'où l'intérêt de les avoir stockés
 - et d'avoir automatisé l'exécution !



Sommaire

JUnit, un framework de test unitaire pour Java

Les bases

Autre exemple issu de

<http://www.devx.com/Java/Article/31983>

Révisons/Apprenons nos design patterns avec JUnit
xUnit

Pour aller plus loin : le test de logiciels

C'est quoi le test ?

Définition

Le test et les autres procédés de V&V

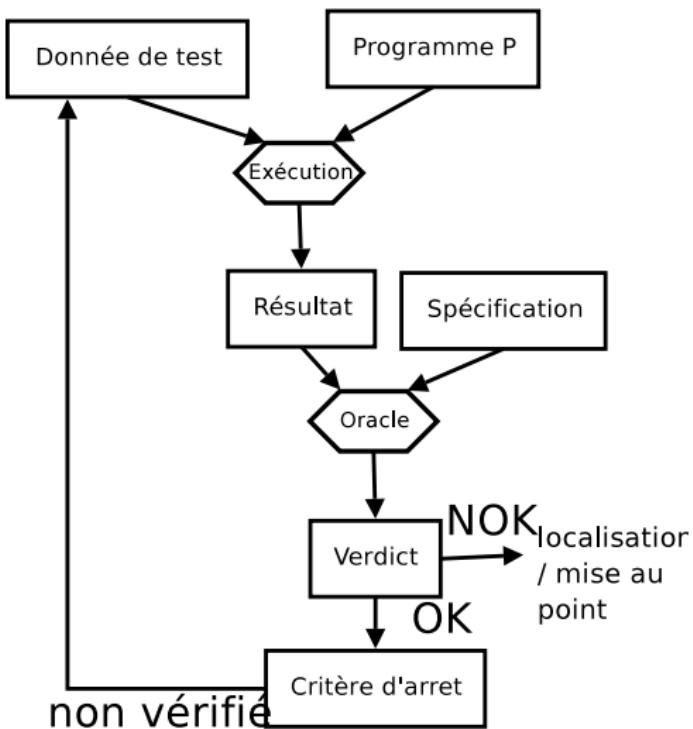
Quels tests pour quelles erreurs ?

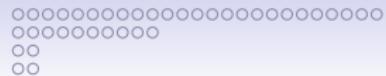
Les techniques

Les types classiques de test

Processus, vocabulaire et difficultés

[label=proc]Processus





Vocabulaire

Oracle

- Aussi appelé fonction d'oracle
- Permet de déterminer si le test a réussi ou échoué
 - ie si le résultat obtenu est celui attendu

Critère d'arrêt

- Permet de déterminer si on a fini de tester



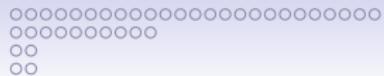
Les difficultés

La génération des données de test

- Comment les choisir ? Sur quels critères ?
- Si on en choisit trop, c'est long / cher !
- Il existe des techniques de génération automatique

L'oracle

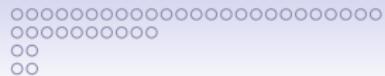
- Comment savoir si ce qu'on a obtenu est correct ?
 - faire le calcul à la main ?
 - utiliser un autre programme ?
 - en théorie : utiliser la spécification ...
 - en pratique : utiliser les propriétés du programme, versions antérieure



Les difficultés

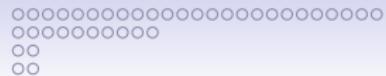
Le critère d'arrêt

- Comment savoir quand il n'est plus nécessaire de tester ?
 - on ne trouve plus d'erreurs depuis 5 minutes ?
 - on n'a plus de temps ?
 - on a passé 10h à tester ?
 - on a exécuté une fois chaque instruction ?
 - on a fait au moins 3 tours dans chacune des boucles ?



Bonnes pratiques

- Conserver les tests, ne jamais les jeter !
- Automatiser au maximum :
 - la génération de test
 - l'exécution
 - l'oracle



Les outils

- Générateurs de test
 - pas toujours avec oracle, à partir de différentes formes de spécification
- Les pilotes de test
 - permettent d'automatiser le lancement des tests, de créer des rapports de test
- Frameworks xUnit
- Outils de mesure de couverture du code
- Outils de monitoring