

FORMATION JAVA FRAMEWORK

JUNIT

Plan

- Principaux types de test
- Principe du test unitaire
- Automatisation des tests unitaires
- Développement conduit par les Test

Différents types de tests (1)

- Les tests unitaires
 - Les tests unitaires consistent à tester individuellement les composants de l'application. On pourra ainsi valider la qualité du code et les performances d'un module.
- Les tests d'intégration
 - Ces tests sont exécutées pour valider l'intégration des différents modules entre eux et dans leur environnement exploitation définitif.
 - Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.
- Les tests fonctionnels
 - Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils sont donc basés sur les spécifications fonctionnelles et techniques.
- Les tests de non-régression
 - Les tests de non-régression permettent de vérifier que des modifications n'ont pas altérées le fonctionnement de l'application.
 - L'utilisation d'outils de tests, dans ce domaine, permet de faciliter la mise en place de ce type de tests.

Différents types de tests (2)

- Les tests IHM
 - Les tests IHM ont pour but de vérifier que la charte graphique a été respectée tout au long du développement.
 - Cela consiste à contrôler :
 - la présentation visuelle : les menus, les paramètres d'affichages, les propriétés des fenêtres, les barres d'icônes, la résolution des écrans, les effets de bord,...
 - la navigation : les moyens de navigations, les raccourcis, le résultat d'un déplacement dans un écran,...
- Les tests de configuration
 - Une application doit pouvoir s'adapter au renouvellement de plus en plus fréquent des ordinateurs. Il s'avère donc indispensable d'étudier l'impact des environnements d'exploitation sur son fonctionnement.
 - Voici quelques sources de problèmes qui peuvent surgir lorsque l'on migre une application vers un environnement différent :
 - l'application développée en 16 bits migre sur un environnement 32 bits,
 - les DLL sont incompatibles,
 - les formats de fichiers sont différents,
 - les drivers de périphériques changent,
 - les interfaces ne sont pas gérées de la même manière...
 - Ainsi, pour faire des tests efficaces dans ce contexte, il est nécessaire de fixer certains paramètres comme par exemple :
 - la même résolution graphique,
 - le même nombre de couleurs à l'écran,
 - une imprimante identique,
 - les mêmes paramètres pour le réseau...

Différents types de tests (3)

- Les tests de performance
 - Le but principal des tests de performance est de valider la capacité qu'ont les serveurs et les réseaux à supporter des charges d'accès importantes.
 - On doit notamment vérifier que les temps de réponse restent raisonnable lorsqu'un nombre important d'utilisateurs sont simultanément connectés à la base de données de l'application.
 - Pour cela, il faut d'abord relever les temps de réponse en utilisation normale, puis les comparer aux résultats obtenus dans des conditions extrêmes d'utilisation.
 - Une solution est de simuler un nombre important d'utilisateur en exécutant l'application à partir d'un même poste et en analysant le trafic généré.
 - Le deuxième objectif de ces tests est de valider le comportement de l'application, toujours dans des conditions extrêmes. Ces tests doivent permettre de définir un environnement matériel minimum pour que l'application fonctionne correctement.
- Les tests d'installation
 - Une fois l'application validée, il est nécessaire de contrôler les aspects liés à la documentation et à l'installation.
 - Les procédures d'installation doivent être testées intégralement car elles garantissent la fiabilité de l'application dans la phase de démarrage.
 - Bien sûr, il faudra aussi vérifier que les supports d'installation ne contiennent pas de virus.

Principe du test unitaire

Un test est unitaire lorsque :

- Il ne communique pas avec la base de données
- Il ne communique pas avec d'autres ressources sur le réseau
- Il ne manipule pas un ou plusieurs fichiers
- Il peut s'exécuter en même temps que les autres tests unitaires
- On ne doit pas faire quelque chose de spécial, comme éditer un fichier de configuration, pour l'exécuter

Et généralement un test unitaire est petit et rapide, il vérifie le traitement d'une méthode de classe et des interactions avec d'autres méthodes de classe. **Ainsi tout ce qui n'est pas un test unitaire constitue alors un test d'intégration.**

Principe du test unitaire

- Maintenant, il se peut que l'on ait à développer une classe dont la responsabilité est d'accéder à la base de données. On dispose alors de plusieurs choix pour tester cette classe :
 - Ecrire un test unitaire en utilisant des objets bouchons (Mock Objects) sur la couche de connexion à la base de donnée
 - Ecrire un test unitaire en bouchonnant la base de données réelle par une base de données mémoire (par exemple HSQLDB)
 - Ecrire un test d'intégration avec le code et la base de données

JUnit

- JUnit fait partie de ces outils qui aident le programmeur dans ce sens. Il est tout d'abord bon de savoir que JUnit n'est pas une usine à gaze qui permet d'automatiser les tests de fonctionnalités macroscopiques (celles fixées par le client). JUnit est quelque chose de très simple, destiné aux personnes qui programment. JUnit est tellement petit et tellement simple que la seule comparaison possible en terme de rapport complexité d'utilisation / utilité est le bon vieux "println" ou les "assertions".



Pourquoi JUnit

- Pourquoi JUnit peut-t-il à ce point révolutionner l'efficacité avec laquelle vous programmez ?
- Un programmeur qui vient de programmer un petit morceau de programme buggé va s'aider d'un débogueur ou de la sortie texte sur la console pour savoir ce que fait son petit morceau de code. Un programmeur qui utilise JUnit va *avant d'écrire le petit morceau de code buggé* d'abord se demander ce que le petit morceau de programme doit faire, puis il va écrire un petit bout de code capable de vérifier si la tâche a bien été réalisée, à l'aide du framework JUnit (le test unitaire). Enfin il va réaliser le petit morceau de code qu'il devait faire, et il va constater que son code est buggé parce que le test unitaire *ne passe pas*. Le test unitaire constate en effet que la tâche n'est pas complétée. Le programmeur qui utilise JUnit va alors déboguer son code jusqu'à ce que la tâche soit correctement réalisée (Rq: Il va éventuellement utiliser un débogueur ou des affichages sur la sortie texte pour y parvenir).

Pourquoi faire tout ça ? (pourquoi travailler plus, je n'ai pas le temps !)

- Parce que pour une raison ou pour une autre, vous devez programmer efficacement. Vous ne pouvez pas vous permettre de sortir des sentiers battus en programmant "hors sujet" ou en oubliant une fonctionnalité. Votre seul objectif en programment est alors de passer les tests unitaires.
- Toujours dans un soucis d'efficacité, vous ne pouvez pas passer non plus votre temps à chercher un bug (ou vous n'aimez pas faire ça...). Que se serait-il passé si le programmeur n'avait pas constaté le bug ? Si vous développez une test unitaire, vous vous ajoutez une marge de sécurité. Vous laissez moins de bugs derrière vous.
- Parce que vous voulez savoir si vous n'êtes pas en train de faire régresser votre code. JUnit vous dira tout de suite si vous avez perdu une fonctionnalité en cours de route, par exemple après avoir supprimé 200 lignes de code en pensant bien faire. Ant peut faire appel à JUnit : vous saurez à chaque compilation s'il y a eu régression. Le refactoring devient moins difficile et aléatoire : vous gagnerez là encore en efficacité. A noter que JUnit est conçu à la base pour éviter ce genre de problème...
- Parce que vous travaillez à plusieurs sur un projet. Vous avez (ou en tout cas vous aurez) une plus grande confiance dans un bout de code fourni avec son test unitaire que dans un bout de code fourni sans. Si voir la barre verte (signe d'un test passé avec succès) n'est pas suffisant pour gagner votre confiance, vous avez alors une seconde possibilité de vérifier le bon fonctionnement, qui est plus rapide que la lecture du code source : la lecture du test. Si le test vous inspire confiance et qu'il passe, alors pourquoi perdre du temps en lisant (pour ne pas dire décrypter !) le code source ? Vous pouvez enfin faire confiance au code des autres, et vous concentrer sur la résolution de vos bugs à vous ! Là encore, vous pouvez gagner en terme d'efficacité.

Junit : Basiques

- Classes de tests “standardisées”
 - Classe qui étend `junit.framework.TestCase`
 - Ne contient pas de `main`
 - Contient des méthodes `testXXXXXX()`
 - Chargées automatiquement
 - Les méthodes doivent contenir des assertions
 - `assertTrue(bool condition)`
 - `assertEquals(int a, int b)`
 - `fail()`

JUnit appliqué à un projet

- Convention de nommage
- Convention pour structurer le projet
- Convention pour réaliser les tests
 - Travail en binome autour d'une spécification
 - Interface puis tests puis implémentation
- Test si possible non triviaux
- Test des méthodes publiques
- Exécuter les tests en même temps que la
- compilation

La définition des cas de tests

- Chaque classe de tests doit avoir obligatoirement au moins une méthode de test sinon une erreur est remontée par JUnit.
 - La découverte des méthodes de tests par JUnit repose sur l'introspection :
 - JUnit recherche les méthodes qui débutent par test, n'ont aucun paramètre et ne retourne aucune valeur.
- Ces méthodes peuvent lever des exceptions qui sont automatiquement capturées par JUnit qui remonte alors une erreur et donc un échec du cas de tests.
- Dès qu'un test échoue, l'exécution de la méthode correspondante est interrompue et JUnit passe à méthode suivante.
- Avec JUnit, la plus petite unité de tests est l'assertion dont le résultat de l'expression booléenne indique un succès ou une erreur.

La définition des cas de tests

- Les cas de tests utilisent des affirmations (assertion en anglais) sous la forme de méthodes nommées `assertXXX()`.
- Il existe de nombreuses méthodes de ce type qui sont héritées de la classe `junit.framework.Assert` :

Méthode	Rôle
<code>assertEquals()</code>	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode <code>equals()</code>). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type <code>Object</code> et pour un objet de type <code>String</code> .
<code>assertFalse()</code>	Vérifier que la valeur fournie en paramètre est fausse.
<code>assertNull()</code>	Vérifier que l'objet fourni en paramètre soit null.
<code>assertNotNull()</code>	Vérifier que l'objet fourni en paramètre ne soit pas null.
<code>assertSame()</code>	Vérifier que les deux objets fournis en paramètre font référence à la même entité. Exemples identiques : <code>assertSame("Les deux objets sont identiques", obj1, obj2) ;</code> <code>assertTrue("Les deux objets sont identiques ", obj1 == obj2) ;</code>
<code>assertNotSame()</code>	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité.
<code>assertTrue()</code>	Vérifier que la valeur fournie en paramètre est vraie.

La définition des cas de tests

- Bien qu'il serait possible de n'utiliser que la méthode assertTrue(), les autres méthodes assertXXX() facilite l'expression des conditions de tests.
- Chacune de ces méthodes possède une version surchargée qui accepte un paramètre supplémentaire sous la forme d'une chaîne de caractères indiquant un message qui sera affiché en cas d'échec du cas de test.
- Le message doit décrire le cas de test évalué à "true".
- L'utilisation de cette version surchargée est recommandée car elle facilite l'exploitation des résultats des cas de tests.
- Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

    public void testCalculer() throws Exception {
        MaClasse2 mc = new MaClasse2(1,1);
        assertEquals(2,mc.calculer());
    }
}
```

La définition des cas de tests

- L'ordre des paramètres contenant la valeur attendue et la valeur obtenue est important pour obtenir un message d'erreur fiable en cas d'échec du cas de test.
- Quelque soit la surcharge utilisée l'ordre des deux valeurs à tester est toujours la même : c'est toujours la valeur attendue qui précède la valeur courante.
- La méthode fail() permet de forcer le cas de test à échouer.
- Une version surchargée permet de préciser un message qui doit être afficher.
- Il est aussi souvent utile lors de la définition des cas de tests de devoir tester si une exception est levée lors de l'exécution.

L'héritage d'une classe de base

- Il est possible de définir une classe de base qui servira de classe mère à d'autres classes de tests notamment en leur fournissant des fonctionnalités communes.
- JUnit n'impose pas qu'une classe de tests dérive directement de la classe `TestCase`.
- Ceci est particulièrement pratique lorsque l'on souhaite que certaines initialisations ou certains traitements soit systématiquement exécutés (exemple chargement d'un fichier de configuration, ...).
- Il est par exemple possible de faire des initialisations dans le constructeur de la classe mère et invoquer ce constructeur dans les constructeurs des classe filles.

JUnit: Mon premier test automatique

Démarche

- Nous allons voir comment écrire notre premier test unitaire pas à pas. Pour ce faire, nous utiliserons le projet de la calculatrice comme exemple.
 - Tout d’abord, nous créerons le projet de calculatrice.
 - Puis nous écrirons la classe d’addition.
 - Ensuite, il faudra créer le répertoire des tests dans lequel seront classés tous les tests. Chaque équipe peut décider de la façon dont elle souhaite organiser son répertoire de code source et en particulier le code de test. Il existe néanmoins une façon classique de procéder qui consiste à créer un dossier de tests symétrique au répertoire des sources pour y classer les tests selon la même organisation de paquet. Cette façon de procéder a l’avantage d’isoler les classes de tests des classes de source tout en permettant aux tests d’accéder à la portée de définition du paquet. Ainsi, lors de la génération de code, il est simple d’isoler les tests du code tout en gardant les classes de tests proches du code source dans une vue par paquet.
 - Enfin, nous pourrions écrire la classe de tests voulue.

GUIDE PAS À PAS (ECLIPSE)

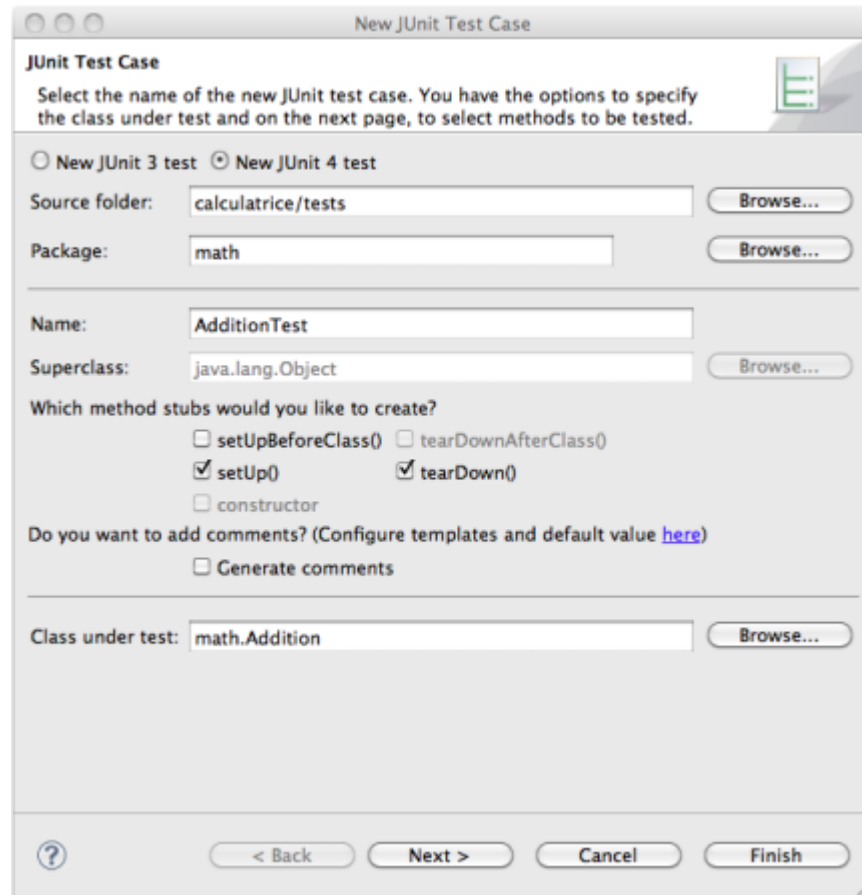
- Cliquez sur File – New – Java Project.
- Nommez votre projet calculatrice en laissant les options par défaut puis cliquez sur OK.
- Ajoutez un package math dans le répertoire src.
- Ajoutez la classe Addition dans le répertoire src.

```
1  package math;
2
3  class Addition {
4      public Long calculer(Long a, Long b) {
5          return a+b;
6      }
7      public Character lireSymbole() {
8          return '-';
9      }
10 }
```

- Ajoutez un nouveau dossier de sources nommé tests au même niveau d'arborescence que src.
- Dans l'explorateur de paquets, faites un clic droit sur la classe Addition.
- Dans le menu contextuel, cliquez sur New – JUnit Test Case.

GUIDE PAS À PAS (ECLIPSE)

- Un panneau s'affiche alors :



The screenshot shows the 'New JUnit Test Case' dialog box in Eclipse. The title bar reads 'New JUnit Test Case'. Inside, the 'JUnit Test Case' section instructs the user to select the name of the new JUnit test case and the class under test. There are two radio buttons: 'New JUnit 3 test' (unselected) and 'New JUnit 4 test' (selected). Below these are fields for 'Source folder' (containing 'calculatrice/tests') and 'Package' (containing 'math'), each with a 'Browse...' button. The 'Name' field contains 'AdditionTest' and the 'Superclass' field contains 'java.lang.Object', also with a 'Browse...' button. A section titled 'Which method stubs would you like to create?' contains four checkboxes: 'setUpBeforeClass()' (unchecked), 'tearDownAfterClass()' (unchecked), 'setUp()' (checked), and 'tearDown()' (checked). There is also an unchecked 'constructor' checkbox. Below this is a question 'Do you want to add comments? (Configure templates and default value [here](#))' with an unchecked 'Generate comments' checkbox. At the bottom, the 'Class under test' field contains 'math.Addition' with a 'Browse...' button. The bottom of the dialog has a help icon, a '< Back' button, a 'Next >' button, a 'Cancel' button, and a 'Finish' button.

New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☒ setUp() ☒ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Class under test:

GUIDE PAS À PAS (ECLIPSE)

- Dans ce panneau :
 - Sélectionnez le bouton radio New JUnit 4 test.
 - Changez le dossier Source folder pour tests.
 - Nommez la classe AdditionTest.
 - Cochez les cases setUp() et tearDown().
 - Dans le champ Class under test, saisissez math.Addition.
 - Enfin cliquez sur Finish.
- Eclipse va remarquer que la bibliothèque de JUnit est absente du projet et vous propose d'ajouter automatiquement cette dernière au projet.
 - Dans le panneau qui apparaît, cliquez sur OK.

GUIDE PAS À PAS (ECLIPSE)

- Eclipse va maintenant créer automatiquement le squelette de la classe de test :

```
1  package math;
2
3  import org.junit.After;
4  import org.junit.Before;
5
6  public class AdditionTest {
7
8      @Before
9      public void setUp() throws Exception {
10     }
11
12     @After
13     public void tearDown() throws Exception {
14     }
15
16 }
```

GUIDE PAS À PAS (ECLIPSE)

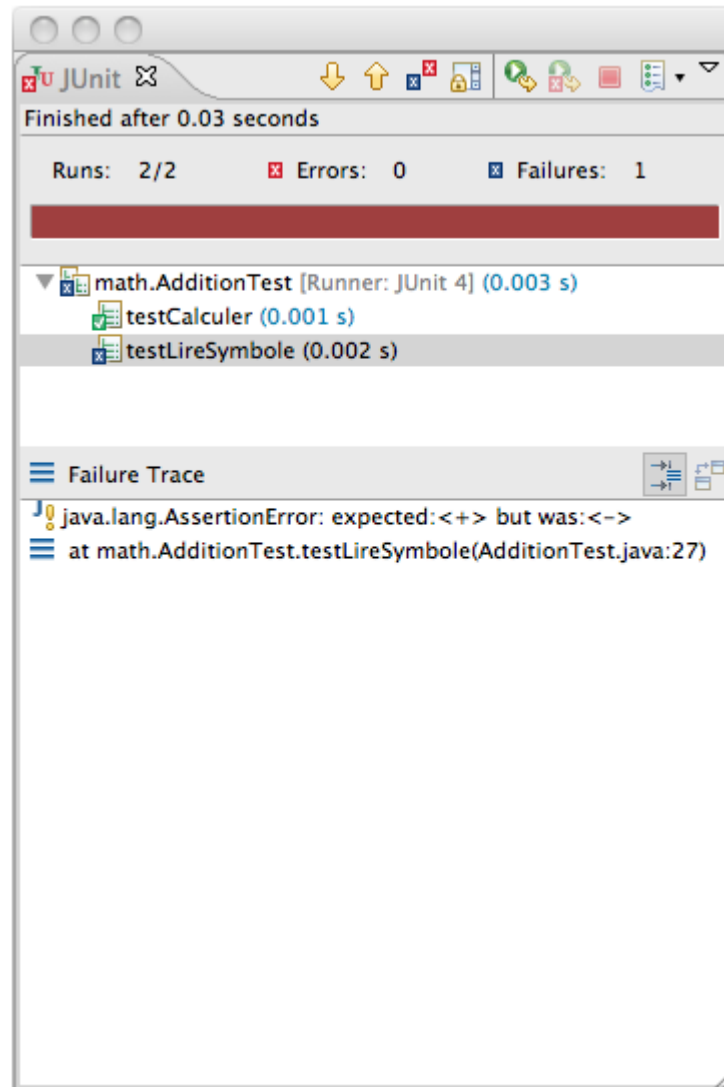
- Il ne reste plus alors qu'à remplir cette dernière.

```
1  package math;
2
3  import org.junit.After;
4  import org.junit.Before;
5  import org.junit.Test;
6  import static org.junit.Assert.*;
7
8  public class AdditionTest {
9      protected Addition op;
10
11      @Before
12      public void setUp() {
13          op = new Addition();
14      }
15
16      @After
17      public void tearDown() {
18      }
19
20      @Test
21      public void testCalculer() throws Exception {
22          assertEquals(new Long(4),
23                      op.calculer(new Long(1), new Long(3)));
24      }
25
26      @Test
27      public void testLireSymbole() throws Exception {
28          assertEquals((Character) '+', op.lireSymbole());
29      }
30  }
```

GUIDE PAS À PAS (ECLIPSE)

- Dans l'explorateur de paquets, faites un clic droit sur la classe AdditionTest.
- Dans le menu contextuel, cliquez sur Run As – JUnit test.
- Enfin, le premier rapport de tests s'affiche !

GUIDE PAS À PAS (ECLIPSE)



GUIDE PAS À PAS (ECLIPSE)

- La barre de progression est rouge, indiquant qu'au moins un test est en échec. Le rapport d'erreur permet de visualiser les tests en échec et d'afficher la cause et l'origine du problème. Dans ce cas, une erreur s'est glissée sur le symbole de l'addition!

L'exécution des tests (1/2)

- JUnit propose trois applications différentes nommées TestRunner pour exécuter les tests en mode ligne de commande ou application graphique :
 - une application console : `junit.textui.TestRunner` qui est très rapide et adaptée à une intégration dans un processus de générations automatiques.
 - une application graphique avec une interface Swing : `junit.swingui.TestRunner`.
 - une application graphique avec une interface AWT : `junit.awtui.TestRunner`.
- Quelque soit l'application utilisée, les entités suivantes doivent être incluses dans le classpath :
 - le fichier `junit.jar`.
 - les classes à tester et les classes des cas de tests.
 - les classes et bibliothèques dont toutes ces classes dépendent.

L'exécution des tests (2/2)

- Suite à l'exécution d'un cas de test, celui ci peut avoir un des trois états suivants :
 - échoué : une exception de type `AssertionFailedError` est levée.
 - en erreur : une exception non émise par le framework et non capturée a été levée dans les traitements.
 - passé avec succès.
- L'échec d'un seul cas de test entraîne l'échec du test complet.
- L'échec d'un cas de test peut avoir plusieurs origines :
 - le cas de test contient un ou plusieurs bugs.
 - le code à tester contient un ou plusieurs bugs.
 - le cas de test est mal défini.
 - une combinaison des cas précédents simultanément.

Les suites de tests (1/2)

- Les suites de tests permettent de regrouper plusieurs tests dans une même classe.
- Ceci permet l'automatisation de l'ensemble des tests inclus dans la suite et de préciser leur ordre d'exécution.
- Pour créer une suite, il suffit de créer une classe de type `TestSuite` et d'appeler la méthode `addTest()` pour chaque classe de tests à ajouter.
- Celle ci attend en paramètre une instance de la classe de tests qui sera ajoutée à la suite.
- L'objet de type `TestSuite` ainsi créé doit être renvoyé par une méthode dont la signature doit obligatoirement être `public static Test suite()`.

Les suites de tests (2/2)

- Celle ci sera appelée par introspection par le TestRunner.
- Il peut être pratique de définir une méthode main() dans la classe qui encapsule la suite de tests pour pouvoir exécuter le TestRunner de la console en exécutant directement la méthode statique Run().
- Ceci évite de lancer JUnit sur la ligne de commandes.
- Deux versions surchargées des constructeurs permettent de donner un nom à la suite de tests.
- Un constructeur de la classe TestSuite permet de créer automatiquement par introspection une suite de tests contenant tous les tests de la classe fournie en paramètre.
- La méthode addTestSuite() permet d'ajouter à une suite une autre suite.

Guide pas à pas

- Dans cette partie, on créera une suite de tests avec exécution sous la console de la jvm
- Etapes :
 - Création d'un dossier **JUNIT_WORKSPACE**
 - Création de la classe MessageUtil.java
 - Création des Classes Test Case
 - Création de la classe Test Suite
 - Création de la classe Test Runner
 - Compilation de tous les éléments avec javac
 - Exécution de TestRunner sous la console

Guide pas à pas

- Création de la classe MessageUtil.java

```
/*
 * This class prints the given message on console.
 */
public class MessageUtil {

    private String message;

    //Constructor
    //@param message to be printed
    public MessageUtil(String message){
        this.message = message;
    }

    // prints the message
    public String printMessage(){
        System.out.println(message);
        return message;
    }

    // add "Hi!" to the message
    public String salutationMessage(){
        message = "Hi!" + message;
        System.out.println(message);
        return message;
    }
}
```


Guide pas à pas

- Création du testcase 1

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJUnit1 {

    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        System.out.println("Inside testPrintMessage()");
        assertEquals(message, messageUtil.printMessage());
    }
}
```

- Création du testcase 2:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJUnit2 {

    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testSalutationMessage() {
        System.out.println("Inside testSalutationMessage()");
        message = "Hi!" + "Robert";
        assertEquals(message, messageUtil.salutationMessage());
    }
}
```

Guide pas à pas

- Création du testcase 1

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestJUnit1.class,
    TestJUnit2.class
})
public class JunitTestSuite {
}
```

- Création du testcase 2:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(JunitTestSuite.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

Guide pas à pas

- Compilation

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit1.java  
TestJunit2.java JunitTestSuite.java TestRunner.java
```

- Exécution

```
C:\JUNIT_WORKSPACE>java TestRunner
```

- Sortie console :

```
Inside testPrintMessage()  
Robert  
Inside testSalutationMessage()  
Hi Robert  
true
```

Intégration de JUnit dans Maven

- Organisation

```
src
+ main
| + java
| | + com
| | | + monprojet
| | | | + Classe.java
+ test
| + java
| | + com
| | | + monprojet
| | | | + ClasseTest.java
```

Guide pas à pas

- Taper la commande :
`mvn archetype:create -DgroupId=exemple -DartifactId=calculatrice -Dversion=1.0`

Maven va créer un dossier calculatrice qui contient un fichier pom.xml.
Éditez ce fichier.

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Changez simplement le numéro de version pour 4.7.

Guide pas à pas

- Ajoutez également la section suivante pour indiquer que nous souhaitons compiler en Java 1.x

```
<build>
  <finalName>Calculatrice</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Guide pas à pas

- Créez le répertoire src/main/java/math.
- Sauvegardez-y le fichier Addition.java.

```
package math;

class Addition {
    public Long calculer(Long a, Long b) {
        return a+b;
    }
    public Character lireSymbole() {
        return '-';
    }
}
```

Guide pas à pas

- Créez le répertoire
src/test/java/math.
- Sauvegardez-y le fichier
AdditionTest.java.

```
1 package math;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6 import static org.junit.Assert.*;
7
8 public class AdditionTest {
9     protected Addition op;
10
11     @Before
12     public void setUp() {
13         op = new Addition();
14     }
15
16     @After
17     public void tearDown() {
18     }
19
20     @Test
21     public void testCalculer() throws Exception {
22         assertEquals(new Long(4), op.calculer(new Long(1),
23                                                new
24 Long(3)));
25     }
26
27     @Test
28     public void testLireSymbole() throws Exception {
29         assertEquals((Character) '+', op.lireSymbole());
30     }
31 }
```


Guide pas à pas

- Enfin, lancez la commande suivante : `mvn test`

```
# mvn test
-----
T E S T S
-----

Running math.AdditionTest
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed:
0.03 sec <<< FAILURE!

Results :

Failed tests:
    testLireSymbole(math.AdditionTest)

Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
```

Test driven development

- Le test-driven development (TDD) ou en français développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.
- L'objectif du TDD est de produire du "code propre qui fonctionne". Pour cela, deux principes sont mis en oeuvre :
 - un développeur écrit du code nouveau seulement lorsqu'un test automatisé a échoué
 - toute duplication de code (ou plus généralement d'information, ou de connaissances) doit être éliminée. L'acronyme anglais DRY (Do not Repeat Yourself) peut être utilisé comme moyen mnémotechnique pour cette phase très importante.
- Ces deux principes doivent être strictement respectés, même s'ils paraissent difficiles ou bizarres dans un premier temps. Bien comprendre le TDD suppose en effet de respecter strictement la discipline imposée par le cycle décrit ci-dessous.

Test driven development

- **Bien que simples, ils ont diverses implications :**
 - nous allons concevoir notre code de manière incrémentale, en ayant toujours du code en état de marche, de telle sorte que ce code nous fournisse de l'information pour prendre de nombreuses petites décisions au cours de notre développement.
 - nous devons écrire nos propres tests, parce que nous ne pouvons pas attendre de nombreuses fois par jour qu'une autre personne le fasse
 - notre environnement de développement doit fournir une réponse ultra rapide en cas de petits changements
 - notre code doit être composé d'éléments très cohérents et très peu couplés, afin de rendre le test facile.

Test driven development

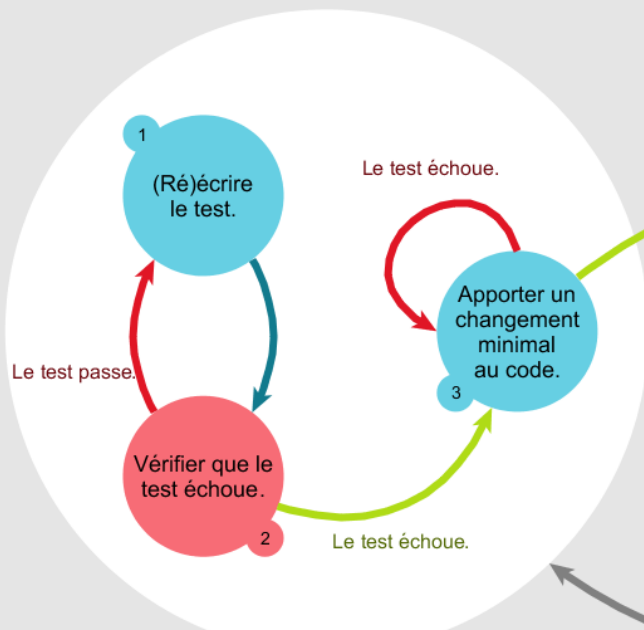
- Le travail se fait en trois phases. Les deux premières sont nommées d'après la couleur de la barre de progrès dans les outils comme Nunit :
 - ROUGE: écrire un petit test qui échoue, voire même ne compile pas dans un 1er temps
 - VERT : faire passer ce test le plus rapidement possible, en s'autorisant si besoin les "pires" solutions : S'il existe une solution propre, simple et immédiate, réalisez-la Si une telle solution prend plus d'une minute, notez la et revenez au problème principal : avoir une barre de progrès verte en quelques secondes
 - REMANIEMENT (refactoring en anglais) : éliminer absolument toute duplication apparue durant les étapes 1 et 2.
- **Pour réaliser l'étape 2 ci-dessus, il y a trois stratégies :**
 - Simulation : retourner une constante, puis remplacer progressivement ces constantes avec des variables afin d'obtenir le code réel
 - Implémentation évidente : taper directement la bonne solution
 - Triangulation : avoir deux exemples du résultat recherché, et généraliser.

Cycle de travail

- **Le cycle de travail en TDD est donc le suivant :**
 - ajouter rapidement un nouveau test
 - Exécuter tous les tests, et constater l'échec du nouveau : ROUGE
 - Faire un petit changement
 - Exécuter tous les tests, et constater qu'ils passent : VERT
 - Remanier le code pour éliminer toute duplication : REMANIEMENT

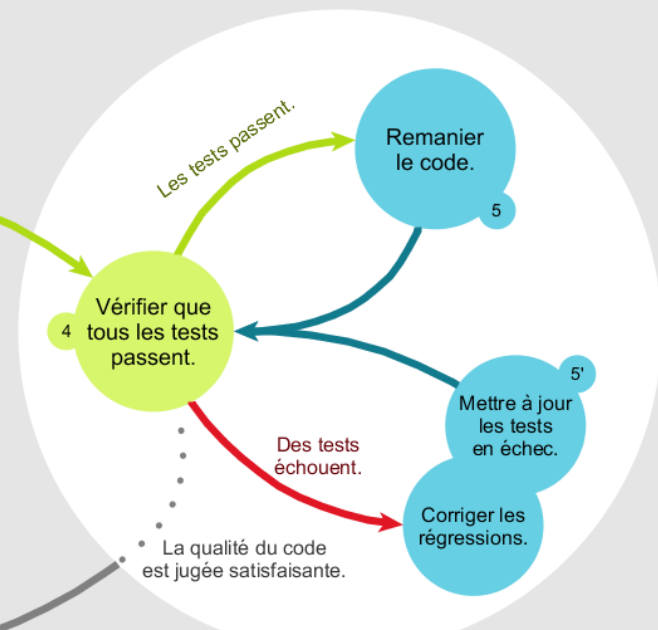
Test driven development

TEST-FIRST DEVELOPMENT



focus
Réalisation du contrat
défini par le test

REFACTORING



focus
Réalignement de la conception
avec les besoins connus

Itérer

Le cycle TDD

