TDD

Test Driven Development

Comment coder en appliquant la méthode TDD ?

Auteur: Frédéric Delorme<[frederic.delorme@gmail.com](mailto:frederic.delorme@gmail.com)>

Créé le: 02/01/2012

Sommaire

[Introduction 2](#_Toc314478736)

[Cahier des charges de l'exercice 3](#_Toc314478737)

[Case de gestions 3](#_Toc314478738)

[BankAccount 3](#_Toc314478739)

[BankUser 3](#_Toc314478740)

[Address 4](#_Toc314478741)

[Part 1, the BankAccount 5](#_Toc314478742)

[Le projet Maven 5](#_Toc314478743)

[Cas de gestion 1 12](#_Toc314478744)

[Cas de gestion 2: Credit() 17](#_Toc314478745)

[Cas de gestion 3: Debit() 21](#_Toc314478746)

[Cas de gestion 4: threshold ! 23](#_Toc314478747)

[BankUser 26](#_Toc314478748)

[TODO 26](#_Toc314478749)

[Address 26](#_Toc314478750)

[TODO 26](#_Toc314478751)

[Part 2 27](#_Toc314478752)

[Nouvelles dépendances 27](#_Toc314478753)

[JPA = Spring+Hibernate 27](#_Toc314478754)

[Références 28](#_Toc314478755)

# Introduction

Nous nous proposons d'expérimenter la méthode TDD pour produire du code pleinement tester et bien facturé en partant de zéro.

Nous avons besoins de quelques outils pour menez à bien cet exercice :

* Eclipse 3.6+ ([http://www.eclipse.org/downloads/](http://www.eclipse.org/downloads/packages/release/helios/sr2)),
* Maven 2+ (<http://maven.apache.org/>),
* Le plugin [m2clipse](http://eclipse.org/m2e/) (<http://eclipse.org/m2e/>) pour intégrer les commandes Maven à Eclipse,
* Et pour nos besoins, le plugin Egit d'Eclipse permettant l'accès aux dépôt GIT (<http://www.git.org/> ) du site github.com de votre serviteur: <https://github.com/mcgivrer/JunitTddTutorial>

# Cahier des charges de l'exercice

Notre client souhaite pouvoir modéliser un compte bancaire. Aussi, nous allons créer une première entité nommée BankAccount en respectant les cas de gestion décrits ci-après.

## Cas de gestions

Les cas proposés ici ne reflètent en rien la réalité et sont seulement présent pour illustrer notre propos concernant l'utilisation du TDD. Concentrons-nous sur les objets BankAccount, BankUser et Address.

### BankAccount

L'entité que nous souhaitons créer doit permettre de modéliser les fluctuations de la valeur d'un compte bancaire. La valeur de ce compte est à ce titre représentée par une valeur décimale permettant de représenter de très grands nombres.

1. Le compte possède un numéro de compte unique sur 23 caractères numériques
2. A sa création, un compte bancaire doit valoir 0.0,
3. Ce compte peut être crédité d'une valeur positive.
   * Dans le cas d'une tentative de créditer le compte d'une valeur négative, une erreur est émise et tracée.
4. Ce compte peut-être débité d'une valeur négative.,
   * Dans le cas d'un débit d'une valeur positive, une erreur est émise et tracée.
5. En cas de passage à une valeur inférieure à 0.0 ou bien inférieure à un seuil minimum:
   * une erreur dépassement de seuil minimum autorisé est émise et tracée.
6. Le compte bancaire peut être lié à une monnaie, monnaie qui doit exister parmi les codes internationaux.
   * Se référer au fichier iso-4217-currency-codes.csv fournissant une telle liste..

### BankUser

Un usager bancaire est identifié par :

1. son nom de type String contenant 50 caractères au maximum,
2. son prénom, de type String contenant 50 caractères au maximum,
3. sa date de naissance de type Date, au format JJ/MM/AAAA,

Quelles informations administratives additionnelles peuvent être fournies :

1. Adresse postale personnelle de type Address,
2. E-mail personnel de type String au format [{nom@domaine.ext}](mailto:%7bnom@domaine.ext%7d),  
   ex : frederic.delorme@capgemini.com
3. Numéro de téléphone (téléphone fixe et/ou téléphone portable) au format international   
   ex : +33 (0) 999 999 999,

Enfin, une règle élémentaire:

1. un usager bancaire(BankUser) peut posséder au moins 1 compte bancaire (BankAccount).

## Address

Un usager bancaire possède une ou plusieurs adresses. L'objet Address permet de renseigner ces informations. Une adresse contient alors les informations suivantes:

* Titre du lieu de type String contenant au maximum 100 caractères,
* Numéro, de type String contenant au maximum 6 caractères numériques,
* Rue, de type String contenant au maximum 100 caractères,
* Code postale, de type String contenant au maximum 6 caractères,
* Ville, de type String, contenant au maximum 60 caractères
* Pays (listé dans la liste de codes internationaux des pays), de type String, sur 3 caractères. La valeur stockée est le code pays, la valeur affichée est le libellé du pays.

# Partie 1, Le compte bancaire.

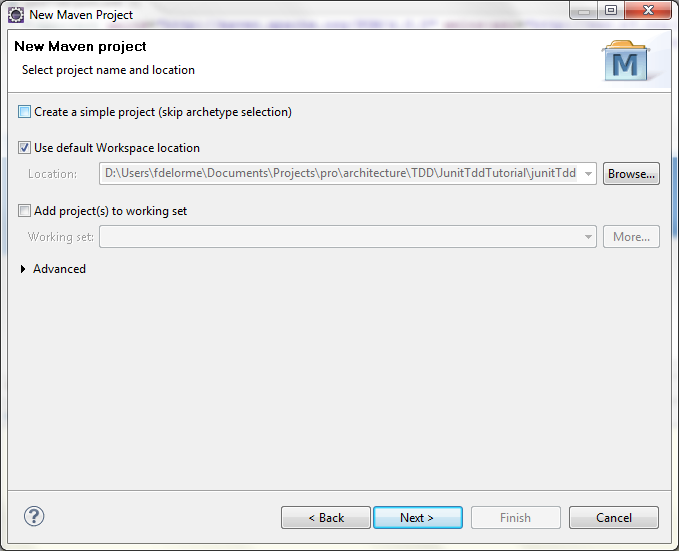
Concentrons nous sur le premier objet qui nous intéresse, a savoir le compte bancaire: BankAccount.

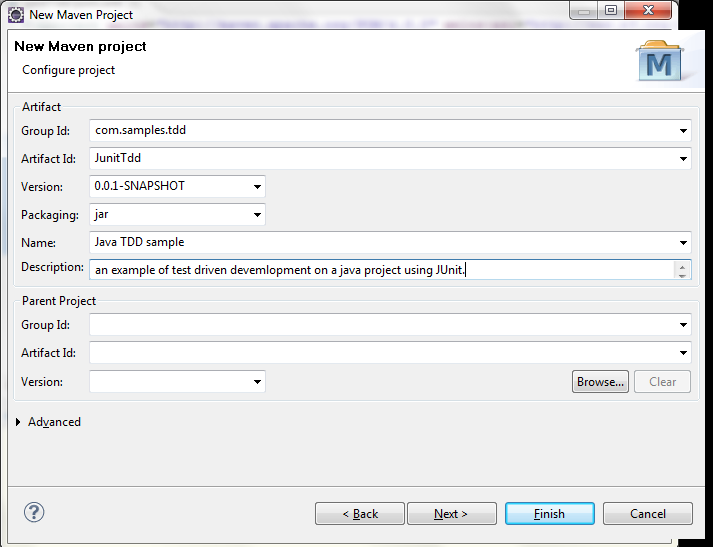
Essayons de le créer en suivant les principes du Test Driven Development.

## Le projet Maven

Comme nous l'avons indiqué en introduction, nous allons utiliser le plugin M2clipse pour exécuter les opérations liées à Maven.

Aussi, en premier lieu, créons le projet maven via l'assistant “New Maven Project” auquel vous accéderez en passant par le menu contextuel de l'explorateur de projet dans eclipse. Sélection New, puis dans la zone de filtre, saisissez “maven”. Vous pouvez alors choisir “New Maven Project”:

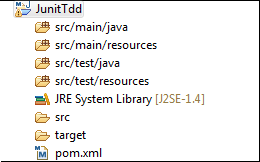
Illustration 1: Create a new Maven Project

Illustration 2: Fill all maven information **GroupId**, **ArtifactId** and **Version.** **Packaging** would be Jar. **Name** and **Description** are not mandatory.

Saisissez les valeurs suivantes :

* GroupId: *com.samples.tdd*
* ArtifactId: *junitTdd*
* version: laissez la valeur proposée par défaut *0.0.1-SNAPSHOT*.

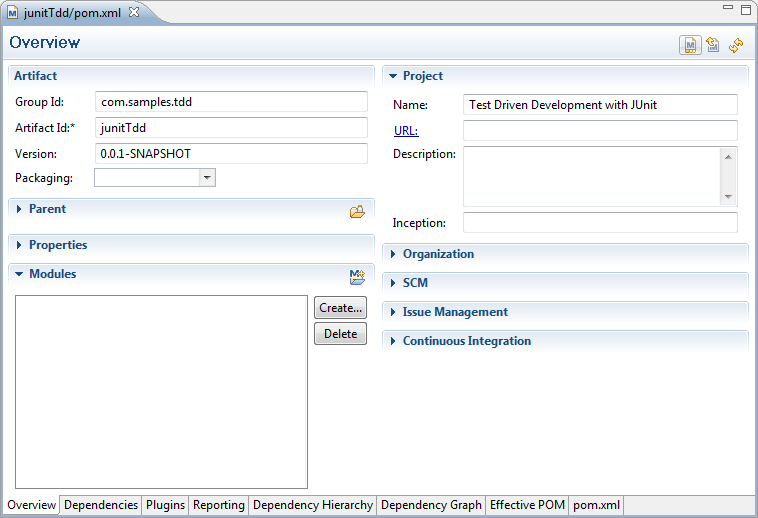
Après quelques secondes vous obtiendrez un projet ayant la structure suivante :

Illustration 3: Structure of a maven project

Nous avons besoin, pour pouvoir coder et exécuter nos tests (,nous les découvrirons ci-après) d'au moins une dépendance: la librairie de test unitaire JUnit !

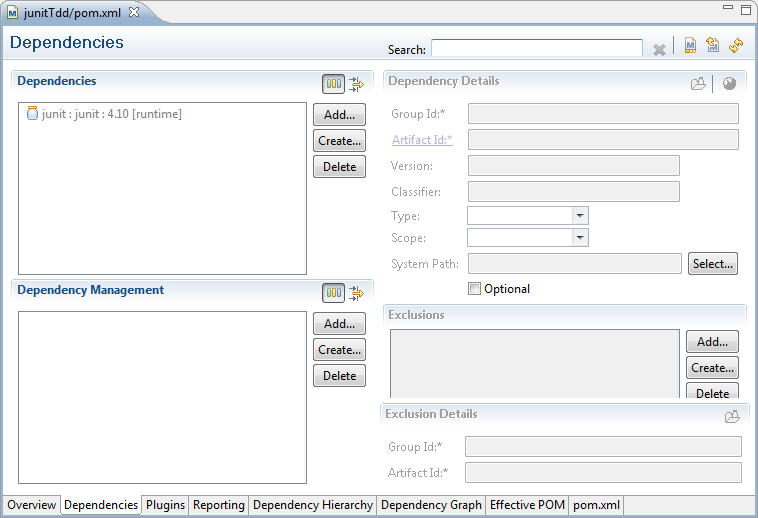
JUnit est une API spécifique permettant de coder et d'exécuter des tests unitaire en Java. (voir <http://www.junit.org/> pour plus d'information).

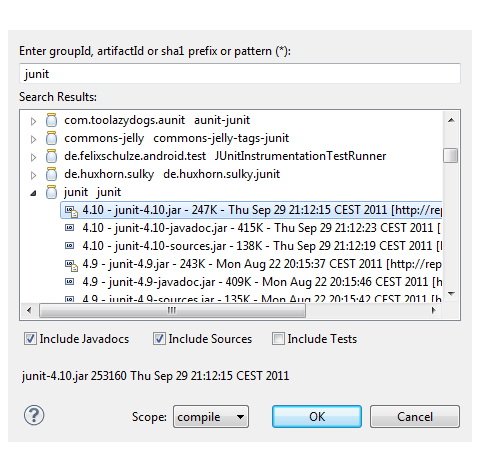
Pour cela, ouvrons notre fichier pom.xml dans notre nouvel éditeur de dépendance Maven fourni par le plugin eclispe pour Maven :

Illustration 4: Default view on the maven configuration file: pom.xml

Cliquez qur l'onglet “Dependencies” (en bas de fenêtre) :

un clic sur le premier bouton “Add” en haut de fenêtre vous permettra de recherche et choisir l'API JUnit :

Illustration 5: Dependencies editor for our JUnitTdd project

Illustration 6: Select the last JUnit version (4.10) to be embbeded in our project

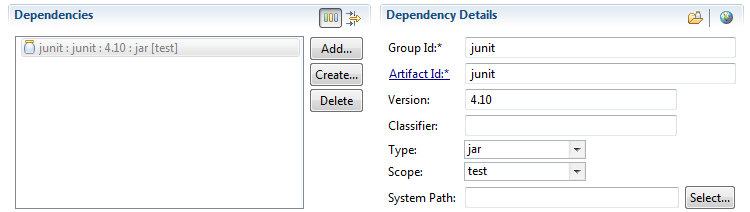
Validez via le bouton “OK”.

Nous avons oublié de sélectionner le scope de notre bibliothèque. Revenons d'ailleurs sur cette notion de Scope.

**Note**

Dans un projet Maven, la notion de scope spécifiée pour chaque dépendance (ou bibliothèque) permet d'indiquer à Maven dans quelle partie du cycle de vie du logicielle cette dépendance sera nécessaire. Dans notre cas, nous souhaiton réaliser des tests. Aussi, choissison “test” comme scope. (pour plus d'information, se référer à la documentation de Maven).

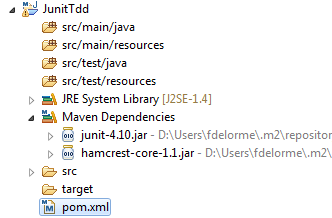
Allons dans l'onglet “Dependencies” et éditons le scope de la bibliothèque JUnit nouvellement insérée, choisissez “test” comme scope:

Illustration 7: Switch scope to "test"

**Note**

Le scope de type “test” indique à Maven qu'au moment du packaging de l'application pour livraison et déploiement, cette bibliothèque ne sera pas livrée car inutile pour l'exécution normale de l'application. Les tests codés n'ont pas lieu d'être déployé en production !

Une nouvelle entrée apparait dans le projet : “Maven Dependencies”

Illustration 8: Maven Dependencies listing all jar needed by your brand new maven project.

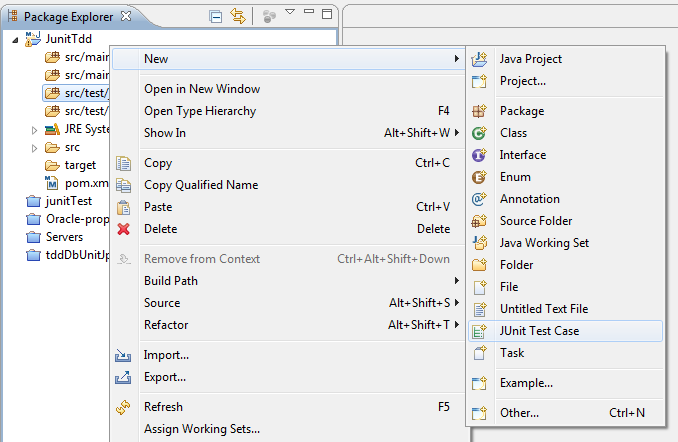
Ensuite, dans le répertoire réservé à la rédaction des tests *src/test/main* créer le package *com.samples.tdd.bank*.

## BankAccount

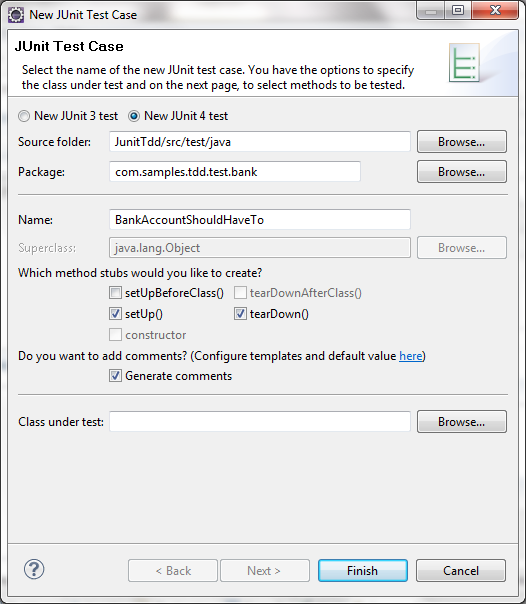
Ajoutez une nouvelle classe Java nommée *BankAccountShouldHaveTo*.

Cette nouvelle classe est une classe de test liée au concept de JUnit et permettant la radaction de test “en ligne”, vous aidant à écrire correctement le code de l'object **BankAccount** pour répondre à l'ensemble des cas de gestions décri dans le chapitre "Cahier des charges”.

Pour créer une telle classe, il est aisé d'utiliser l'assistant réservé à cet effet :

Illustration 9: Create a new Junit Test Case

L'assistant se présente de la manière suivante :

Illustration 10: Perform setting for the BankAccountShouldHaveTo test case.

Et il produit par défaut le code suivant :

/\*\*

\* Test Case implementation for the BankAccount Entity creation.

\* @author Frédéric Delorme<frederic.delorme@gmail.com>

\*

\*/

public class BankAccountShouldHaveTo {

/\*\*

\* @throws java.lang.Exception

\*/

@Before

public void setUp() throws Exception {

}

/\*\*

\* @throws java.lang.Exception

\*/

@After

public void tearDown() throws Exception {

}

@Test

public void test() {

fail("Not yet implemented");

}

}

Allons revoir la liste des cas d'utilisation (ou cas de gestions) proposés pour l'objet BankAccount. Nous devons en premier lieu créer notre objet avec la valeur 0.0 par défaut.

### Cas de gestion 1

Le compte possède un numéro de compte unique codé sur 23 caractères. Il doit être généré à sa création.

Note :

Dans le cadre de l’exercice, nous allons utiliser un générateur aléatoire pour créer ce numéro de compte.

Le code du générateur est fourni dans le code de l’exemple, il suffit de l’appeler comme suit :

String value =

RandomGenerator.getString(23,RandomGenerator.StringGenType.NUMERIC);

Le code généré est alors une chaine de caractères de 23 caractères numériques.

Créez la classe de cas de test dans le même package que la précédente :

### Cas de gestion 2

Créez votre premier test en copiant les lignes suivantes en lieu et place de la fonction de test() proposée par défaut :

@Test

public void beInitializedWithAZeroValue() {

BankAccount ba = new BankAccount();

assertEquals("BankAccount is not initialized “

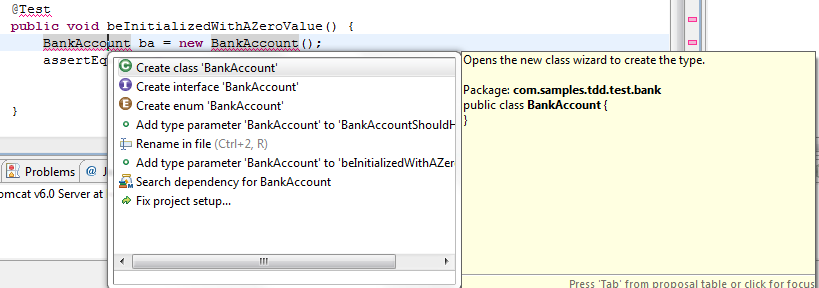
+ ”with a Zero default value !",

Double.valueOf(0.0),ba.getValue());

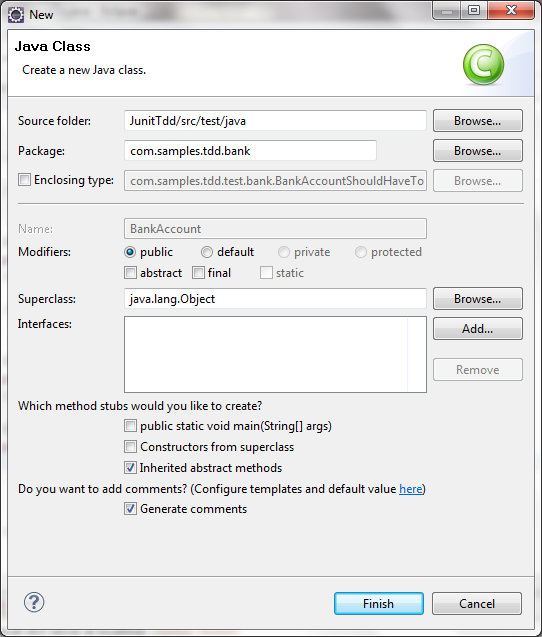
}

Quelques erreurs apparaissent dans l'onglet “Problems” d'Eclipse. Vous devez créer la classer BankAccount.

Pour cela, pressez le raccourci clavier suivant : CTRL+SHIFT+1 et l'assistant de code d'Eclipse vous propose automatiquement de créer cette classe :

Illustration 11: Create the missing class BankAccount with Eclipse wizard !

N'oubliez par lors de sa création de bien spécifier le répertoire source dans lequel cette classe doit être créée (*src/main.java*), choisissez aussi le package de destination en *com.samples.tdd.bank* et cochez “Generate Javadoc comments”.

Illustration 12: Class wizard to perform creation of our BankAccount in the com.samples.tdd.bank package

Vous obtenez alors le code suivant :

package com.samples.tdd.bank;

/\*\*

\* BankAccount Entity definition.

\* @author Frédéric Delorme<frederic.delorme@gmail.com>

\* @see com.samples.tdd.test.balance.BankAccountShouldHaveTo

\*

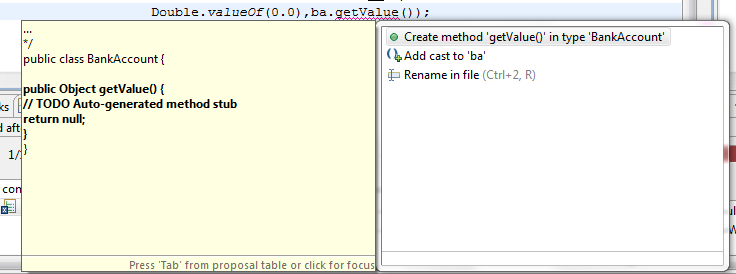
\*/

public class BankAccount {

}

Sauvegarder le fichier ainsi généré et retournez vers la classe de test BankAccountShouldHaveTo.

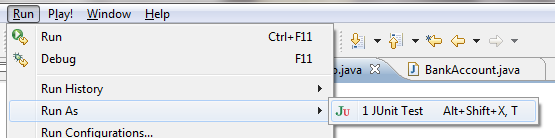
Sautez à la marque d'erreur suivante dan l'éditeur de code et appeler a nouveau l'assistant de code d'Eclipse avec un CTRL+SHIFT+1 pour appeler l'assistant de création de méthode :

Illustration 13: Generate default implementation for getValue().

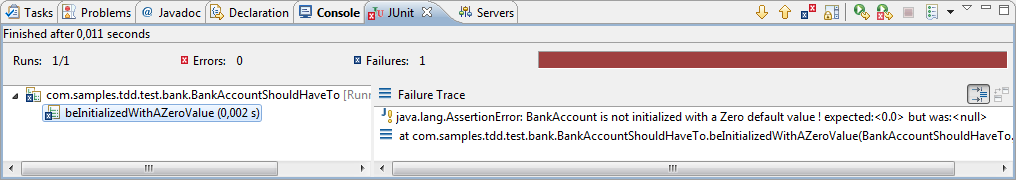
A nouveau, sauver le fichier BankAccount.java ainsi modifié et revenez à la fenêtre de la classe de test.

#### The Red step

Exécutez le test JUnit via le menu Run => run as … => 1 Junit Test , ou pressez les touches magiques suivantes : ALT+SHIFT+X,T :



Le test JUnit est alors exécuté par Eclipse et une jolie couleur rouge apparait alors dans la vue JUnit nouvellement apparue. L'implementation fournie par défaut par l'assistant ne rempli pas le cas de gestion demandé: la méthode getValue() retourne une valeur nulle (Voir le message dans l'onglet Junit).

Illustration 14: Red Step : test are executed with red error

#### The Green Step

La prochaine étape consiste à fournir l'implémentation correcte de notre méthode *getValue():*

Pour cela retournez à l'éditeur de code de notre BankAccount:

public Double getValue() {

return new Double(0.0);

}

De retour vers la classe de test BankAccountShouldHaveTo et executez une réorganisation des imports java présent en début de fichier avec CTRL+CHIFT+O et un CTRL+SHIFT+F. Qui permet de le reformatage complet de votre fichier en respect avec la notation standard Java :

package com.samples.tdd.test.bank;

import static org.junit.Assert.assertEquals;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import com.samples.tdd.bank.BankAccount;

/\*\*

\* Test Case implementation for the BankAccount Entity creation.

\* @author Frédéric Delorme<frederic.delorme@gmail.com>

\*

\*/

public class BankAccountShouldHaveTo {

/\*\*

\* @throws java.lang.Exception

\*/

@Before

public void setUp() throws Exception {

}

/\*\*

\* @throws java.lang.Exception

\*/

@After

public void tearDown() throws Exception {

}

@Test

public void beInitializedWithAZeroValue() {

BankAccount ba = new BankAccount();

assertEquals("BankAccount is not initialized "

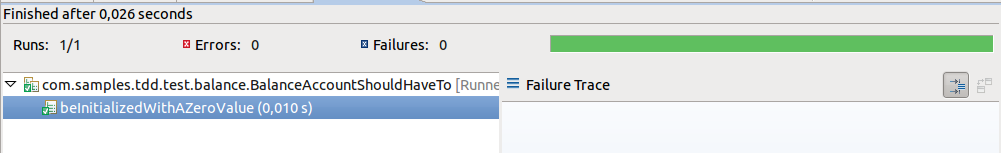
+ "with a Zero default value !",

Double.valueOf(0.0),ba.getValue());

}

}

Comme vous pourrez le noter, cette implémentation est temporaire est n'est présente ici que pour faire passer au vert notre test Junit.

Illustration 15: Green Step, JUnit test case is well executed: first implementation is ok.

#### The Refactor step

Mais cette implementation n'est pas fonctionnellement correcte, la valeur de notre compte doitêtre un attribut de l'objet car elle est appeler à changer dans le cycle de vie de l'objet.

Refactorez donc l'objet pour nous conduire au code suivant dans la classe *BankAccount* :

public class BankAccount {

/\*\*

\* Value for this BankAccount.

\*/

private Double value;

/\*\*

\* Initialization of the instance with default 0.0 value.

\*/

public BankAccount(){

value=0.0;

}

/\*\*

\* Return BankAccount value.

\* @return

\*/

public Double getValue() {

// TODO Auto-generated method stub

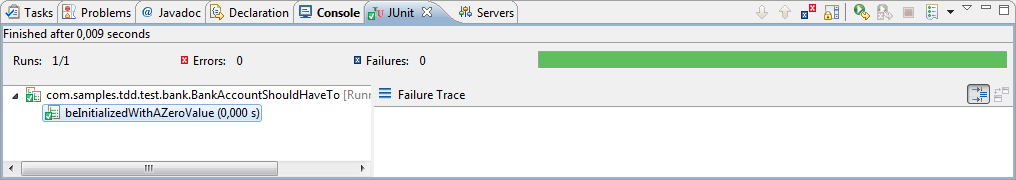
return value;

}

}

Relancez l'exécution du test JUnit avec SHIFT+ALT+X, T et vous obtenez le même résultat qu'auparavant, mais cette fois l'implémentation est correcte en regard du fonctionel demandé :

Passez maintenant au cas de gestion suivant.

Illustration 16: After refactoring, test are green again.

### Cas de gestion 3: Credit()

Comme décrit dans le cas de gestion, une méthode doit permettre de créditer notre compte d'une valeur positive. Nous devons donc commencer par coder le test vérifiant cette méthode. Ajoutez un nouveau test. Pour cela ajoutez les lignes de code suivantes :

@Test

public void beCreditedWithAPositiveValue(){

BankAccount ba = new BankAccount();

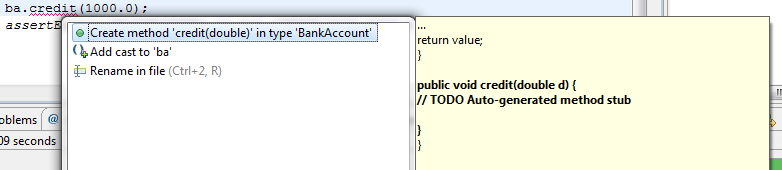
ba.credit(1000.0);

assertEquals("BankAccount is not created with 1000.0 bucks !",

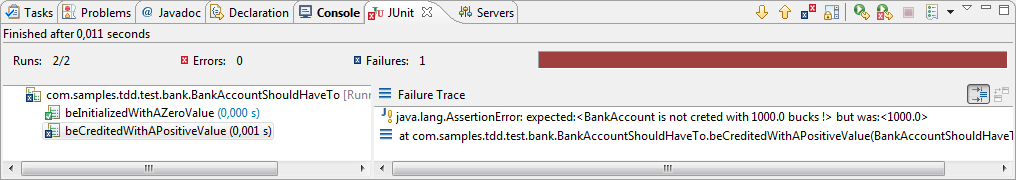
Double.valueOf(1000.0));

}

OK, vous allez évidemment utiliser à nouveau l'assistant de création de méthode pour la méthode permettant l'opération de crédit :

Illustration 17: Implement the credit() method with Method's wizard !

#### The Red step:

Illustration 18: Execute test: the red phase !

Corrigeons l'implémentation par défaut par le code ci-dessous :

/\*\*

\* Credit <code>creditValue</code> bucks to the BankAccount value.

\* @param creditValue

\*/

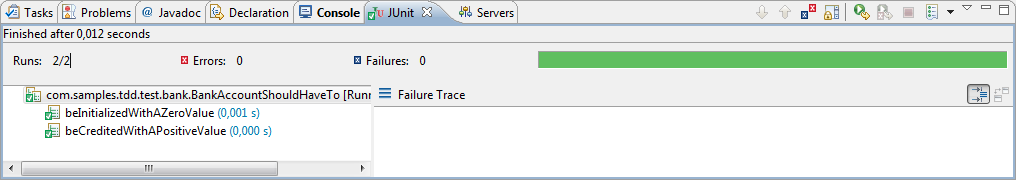
public void credit(double creditValue) {

value+=creditValue;

}

Et relancez à nouveau l'exécution du test JUnit.

#### The Green Step

Illustration 19: Green: implementation is near OK.

#### The Refactor Step

Mais il est clairement indiqué dans notre cas de gestion que l'implémentation doit génerer une erreur dans le cas ou la valeur passée à l'opération de crédit est négative :

Aussi, complétez le test de la manière suivante :

@Test

public void beCreditedWithAPositiveValue(){

BankAccount ba = new BankAccount();

try{

ba.credit(1000.0);

assertEquals("BankAccount is not credited “

+”with 1000.0 bucks !",

Double.valueOf(1000.0), ba.getValue());

ba.credit(-500.0);

assertEquals("BankAccount was credited with “

+”a negative value !",

Double.valueOf(500.0),ba.getValue());

}catch (Exception e ){

if(!(e instanceof BankAccountNegativeCreditException)){

fail("BankAccount throw wrong exception”

+” for negative credit value");

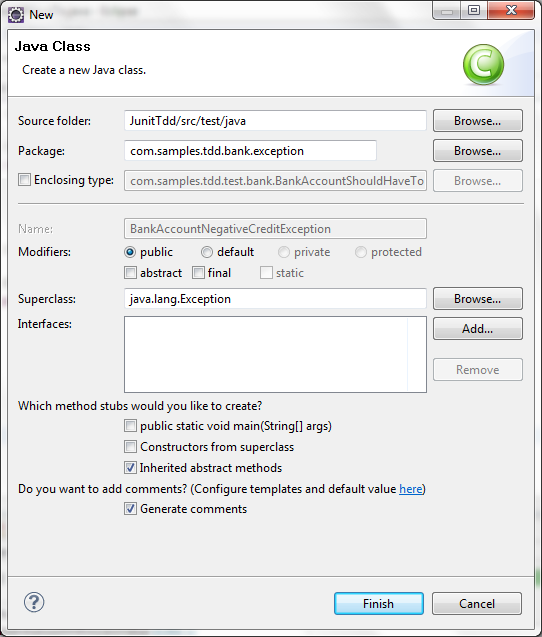
}

}

}

Vous pouvez maintenant implémenter une nouvelle “Exception”.

A l'aide de l'assistant de classe d'Eclipse New... => Class, créer une nouvelle exception dans le package com.samples.tdd.bank.exception héritant de l'objet Exception :



Voici le code obtenu:

/\*\*

\* Exception thrown on Negative value input to BankAccount.credit() method.

\* @author FDELORME

\*

\*/

public class BankAccountNegativeCreditException extends Exception {

/\*\*

\*

\*/

private static final long serialVersionUID = -645567821894120585L;

/\* (non-Javadoc)

\* @see java.lang.Throwable#getMessage()

\*/

@Override

public String getMessage() {

return super.getMessage();

}

}

Et maintenant, implémentez la nouvelle méthode de credit() :

/\*\*

\* Credit <code>creditValue</code> bucks to the BankAccount value.

\* @param creditValue

\* @throws BankAccountNegativeCreditException

\*/

public void credit(double creditValue) throws BankAccountNegativeCreditException {

if(creditValue<0){

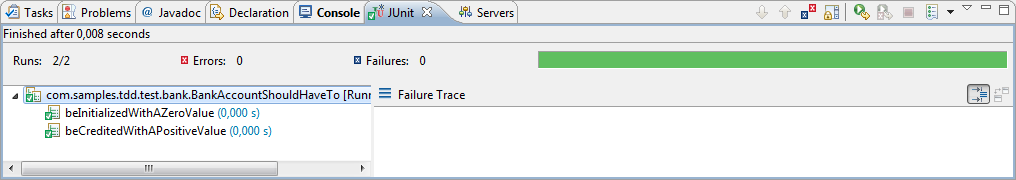
throw new BankAccountNegativeCreditException();

}

value+=creditValue;

}

Vérifiez et confirmez qu'près refactoring, du code, l'exécution du test est de nouveau passée au vert :

Illustration 20: Green state for the test of our new Exception error.

### Cas de gestion 4: Debit()

Dans le cas du 3ème cas de gestion, nous devons vérifier si nous pouvons débiter une valeur négative sur notre BankAccount. Dans le cas d'une valeur passée positive, une erreur doit être émise par l'objet BankAccount :

#### The Red Step

Ci-dessous le nouveau test pour la méthode debit() :

@Test

public void beDebitedWithANegativeValue(){

BankAccount ba = new BankAccount();

try{

ba.debit(-200.0);

assertEquals("BankAccount is not debited with 200.0 bucks !",

Double.valueOf(-200.0), ba.getValue());

ba.debit(200.0);

assertEquals("BankAccount was debited with a positive value !", Double.valueOf(-200.0),ba.getValue());

}catch (Exception e ){

if(!(e instanceof BankAccountPositiveDebitException)){

fail("BankAccount throw wrong exception for positive debit value");

}

}

}

Ainsi, a l'aide de l'assistant de classe, générez une nouvelle exception du nom de BankAccountPositiveDebitException dans le package com.samples.tdd.bank.exception au sein du répertoire source src/main/java :

public class BankAccountPositiveDebitException extends Exception{

/\*\*

\*

\*/

private static final long serialVersionUID = -685229677548773214L;

/\* (non-Javadoc)

\* @see java.lang.Throwable#getMessage()

\*/

@Override

public String getMessage() {

// TODO Auto-generated method stub

return super.getMessage();

}

}

Implémentez la méthode BankAccount.debit(double debitValue) :

/\*\*

\* Debit <code>debitValue</code> bucks from the BankAccount value.

\* **@param** debitValue

\*/

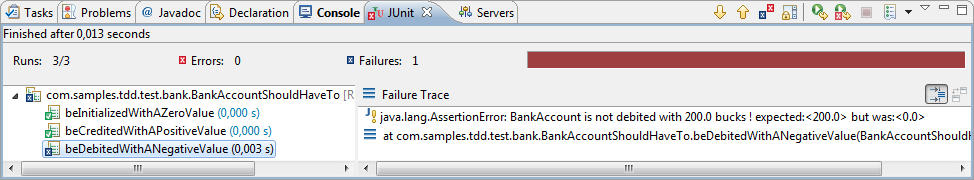
**public** **void** debit(**double** debitValue) {

debit+=debitValue;

}

Et lancez le test JUnit :

#### The Green Step

Illustration 21: Debit implementation is not ok. functionality are not covered.

Modifiez le code pour émettre l'exception comme indiqué :

/\*\*

\* Debit <code>debitValue</code> bucks from the BankAccount value.

\* @param debitValue

\* @throws BankAccountPositiveDebitException

\*/

public void debit(double debitValue) throws BankAccountPositiveDebitException {

if(debitValue>0){

throw new BankAccountPositiveDebitException();

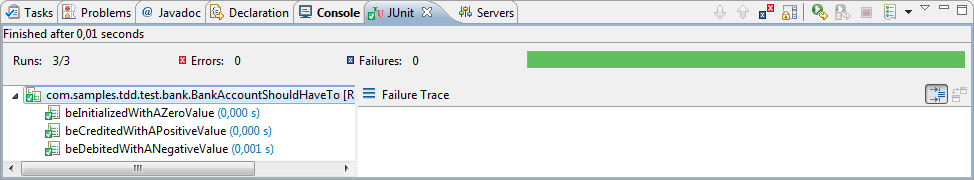
}

value+=debitValue;

}

Si vous lancez maintenant l'exécution du test, vérifier que celui-ci vire au vert :)

#### The Refactor Step

Illustration 22: BankAccount.debit() method well implemented.

Dans ce cas, nous avons bénéficié du step de refactoring précédent pour appliquer directement la création d’un code bon.

### Cas de gestion 5: threshold !

Le cas de gestion suivant décrit un seuil d'autorization de dépassement (en anglais “threshold”) qui doit permettre la levé d'un message d'erreur en cas de dépassement dans l'objet BankAccount.

Par défaut ce seuil est de 0.0, mains une autre valeur négative peut être autorisée.

Commencez par coder le test de vérification lors du débit vis-à-vis du seuil par défaut 0.0 :

@Test

public void beInitializedWithAZeroThreshold(){

BankAccount ba = new BankAccount();

assertEquals("BankAccount default threshold value is not set to 0",Double.valueOf(0.0), ba.getThreshold());

}

Executez le test, vous devriez avoir une “code rouge” sur celui-ci.

Aussi, ajoutons le code d'initialisation du seuil dans le constructeur par défaut de notre classe BankAccount :

public BankAccount(){

value=0.0;

threshold = 0.0;

}

En relançant l'exécution du test, on obtient directement un feu vert ;)

Vous devez maintenant coder le test de l'emission de dépassement du seuil, après la création d'une nouvelle exception du nom du BankAccountThresholdOverflowException

/\*\*

\* Exception emitted on threshold overflow on <code>BankAccount.debit()</code> operation.

\* **@author** FDELORME

\* **@see** com.samples.tdd.bank.BankAccount#debit(double)

\*

\*/

**public** **class** BankAccountTresholdOverflowException **extends** Exception {

/\*\*

\*

\*/

**private** **static** **final** **long** *serialVersionUID* = -3781400530800001356L;

/\* (non-Javadoc)

\* @see java.lang.Throwable#getMessage()

\*/

@Override

**public** String getMessage() {

// **TODO** Auto-generated method stub

**return** **super**.getMessage();

}

}

Et ajoutez le code ci-dessous dans notre test :

@Test

**public** **void** beThresholdOnDebitOperation(){

BankAccount ba = **new** BankAccount(1000.0);

ba.setThreshold(-500.0);

*assertEquals*("BankAccount Threshold was not set to 2000.0 bucks !",

Double.*valueOf*(-500.0),ba.getThreshold());

ba.setValue(1500.0);

**try** {

ba.debit(-2500.0);

*assertEquals*("BankAccount threshold exception was not emitted !",Double.*valueOf*(0.0),ba.getThreshold());

} **catch** (BankAccountPositiveDebitException pde) {

*fail*("BankAccount debit error !");

} **catch** (BankAccountTresholdOverflowException toe) {

**if**(!(toe **instanceof** BankAccountThresholdOverflowException)){

*fail*("BankAccount emit wrong exception on threshold overflow.");

}

}

}

En respectant la méthodologie TDD, vous obtenez une feu rouge surl'executionde ce code.

Implémentez correctement la méthode BankAccount.debit(double d) pour obtenir un feu vert :

/\*\*

\* Debit <code>debitValue</code> bucks from the BankAccount value.

\* @param debitValue

\* @throws BankAccountPositiveDebitException

\* @throws BankAccountTresholdOverflowException

\*/

public void debit(double debitValue)

throws BankAccountPositiveDebitException,

BankAccountTresholdOverflowException {

if(debitValue>0){

throw new BankAccountPositiveDebitException();

}else if((value+debitValue)<threshold){

throw new BankAccountTresholdOverflowException();

}else{

value+=debitValue;

}

}

Attention, cependant, nous avons oublié de mettre en oeuvre un test, le cas ou le seuil n'est pas atteint lors d'un débit :

@Test

public void notBeThresholdOnAuthorizedDebitOperation(){

BankAccount ba = new BankAccount(1000.0);

try{

ba.debit(-200.0);

assertEquals("BankAccount threshold exception was not emitted !",Double.valueOf(0.0),ba.getThreshold());

}catch(Exception e){

if(!(e instanceof BankAccountTresholdOverflowException)){

fail("BankAccount emit wrong exception on threshold overflow.");

}

}

}

Nous avons maintenant terminé de coder et partiellement refactoré notre classe de BankAccount, nous pouvons effectuer le dernier point, a savoir un refactoring de la totalité de l'objet pour éviter la duplication de code.

Vous pouvez noter que depuis le début, nous n'avons pas modifié les méthodes générées au tout début, à setUp() et tearDown().

Nous y voilà !

Vous avez très certainement remarqué que pour chaque test, nous avons en premier lieu instancié un nouvel objet BankAccount pour l'exécution du test:

BankAccount ba = new BankAccount(1000.0);

Dans la documentation de JUnit, vous découvrirez que la méthode setUp() précédée de l'annotation @Before demandant au moteur JUnit d'être exécutée avant toute autre méthode, setUp est exécutée avant chaque méthode de test !

Refactorons notre code pour prendre en compte cette précieuse information, et nous allons allèger notre code de toutes les instanciations de BankAccount :

/\*\*

\* BankAccount instance used throught mainly all test.

\*/

private BankAccount ba;

/\*\*

\* @throws java.lang.Exception

\*/

@Before

public void setUp() throws Exception {

ba = new BankAccount();

}

Maintenant, nettoyons toutes les méthodes précédées du @Test pour supprimer quand cela est possible l'instanciation de l'objet BankAccount.

## BankUser

TODO

## Address

TODO

# Part 2

*TODO*

Nous devons ajouter de nouvelles dépendances car nous souhaitons procéder au stockage en base de données de nos informations. De plus des relations seront mises en place entre BankAccount et BankUser.

## Nouvelles dépendances

Ajoutons ces nouevlles entrées dans le fichier pom.xml, toujours via l'onglet « Dependencies » de l'éditeur Maven :

* DBunit 2.4.8
* spring-core
* hibernate-entitymanager

Modifions le projet pour intégrer les configurations nécessaires à Spring et à Hibernate pour permettre la mise en œuvre d'une persistance respectant la spécification JPA (Java Persistance Api).

### JPA = Spring+Hibernate

JPA est une spécification de Sun (enfin, Oracle maintenant) qui préconise comment la persistance des entités manipulées par vos applications doit être mise en œuvre dans la pile Java EE (voir <http://docs.oracle.com/javaee/6/api/javax/persistence/package-summary.html>) où dans la documentation <http://en.wikipedia.org/wiki/Java_Persistence_API>.

#### Spring

TODO

#### Hibernate

TODO

# Références

* Junit : [www.junit.org](http://www.junit.org/)
* DbUnit: [www.dbunit.org](http://www.dbunit.org/)
* CobolUnit : <https://sites.google.com/site/cobolunit/>