Introduction aux tests unitaires et TDD *(dernière MAJ : 07/02/2009)*

Table des matières

Tester avant d'implémenter

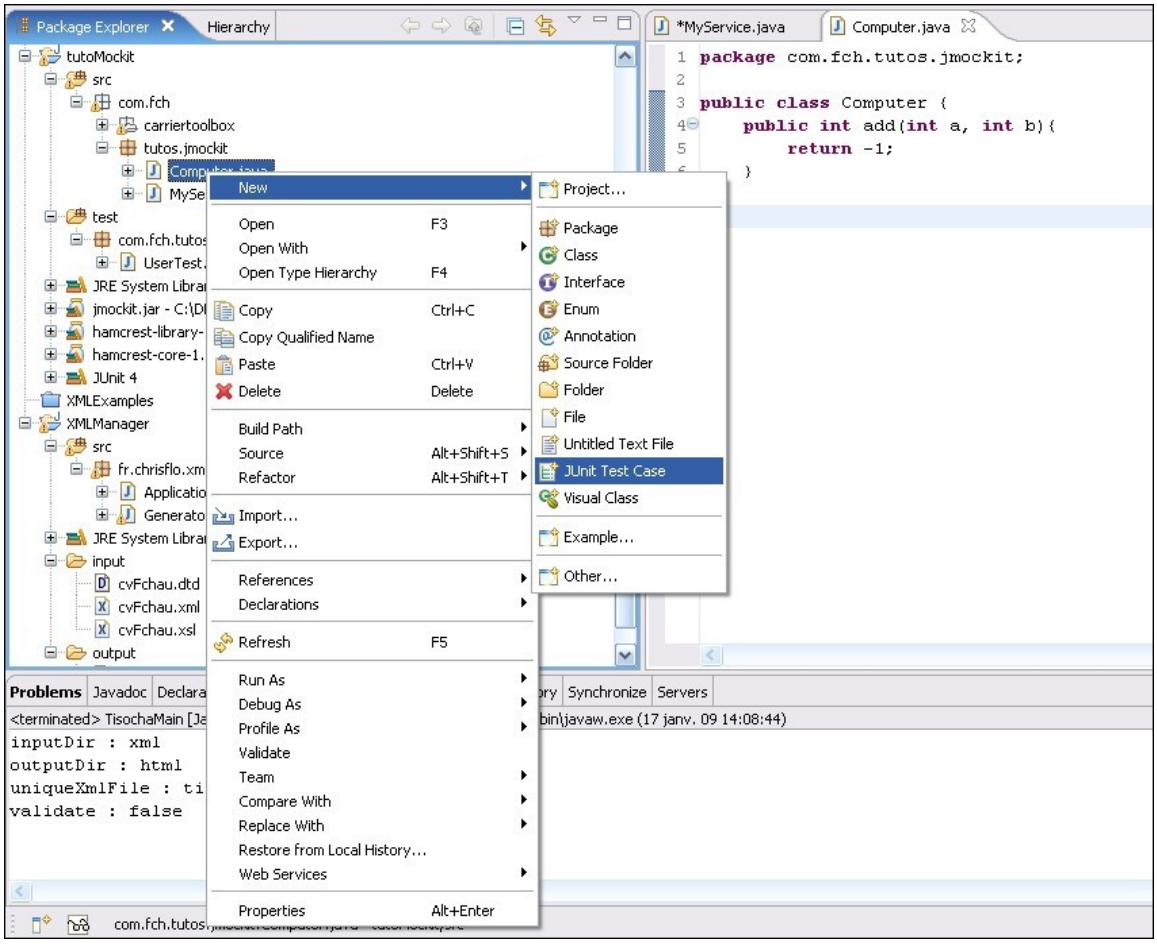
Ces dernières années, le *test driven development* est de plus en plus populaire : les tests sont écrits *avant* d'implémenter la méthode. Cette pratique comporte plusieurs avantages, entre autres :

1. En commençant par les tests, nous ne pouvons pas oublier des les faire.
2. Les projets étant plus souvent en retard qu'en avance, prévoir les tests à la fin présente le risque qu'ils ne soient jamais écrits, faute de temps.
3. Elle permet de s'assurer que le test teste réellement quelque chose, dans la mesure où au début le test ne passe pas, et à la fin de l'implémentation il passe. Il arrive plus souvent que l'on ne croit qu'un test ne soit pas « activé ».

Concrètement, la méthode est créée vide, avec le scrict nécessaire pour que le code compile.

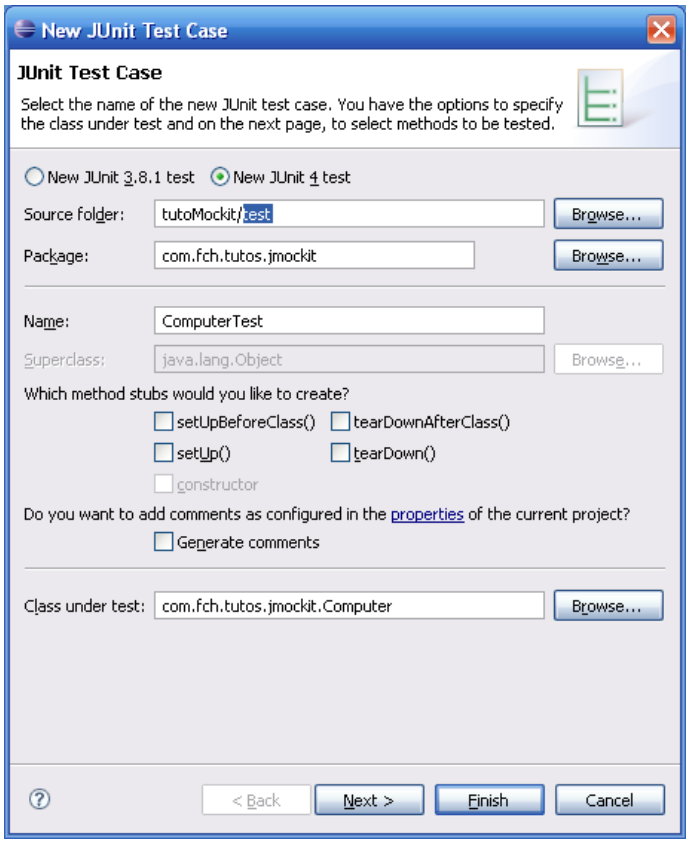
```
public class Computer {
      public int add(int a, int b){
            return -1;
      }
}
```

Avec Eclipse, le plugin Junit permet de créer facilement un test : il suffit de sélectionner la classe à tester dans l'explorateur et sélectionner « new Junit Test Case ».



Pensez à changer la source folder (remplacer *src* par *test* par exemple), et cochez les méthodes que vous souhaiter générer automatiquement.

La méthode *TearDown* est appelée après chaque test unitaire, *setUp* avant chaque test.

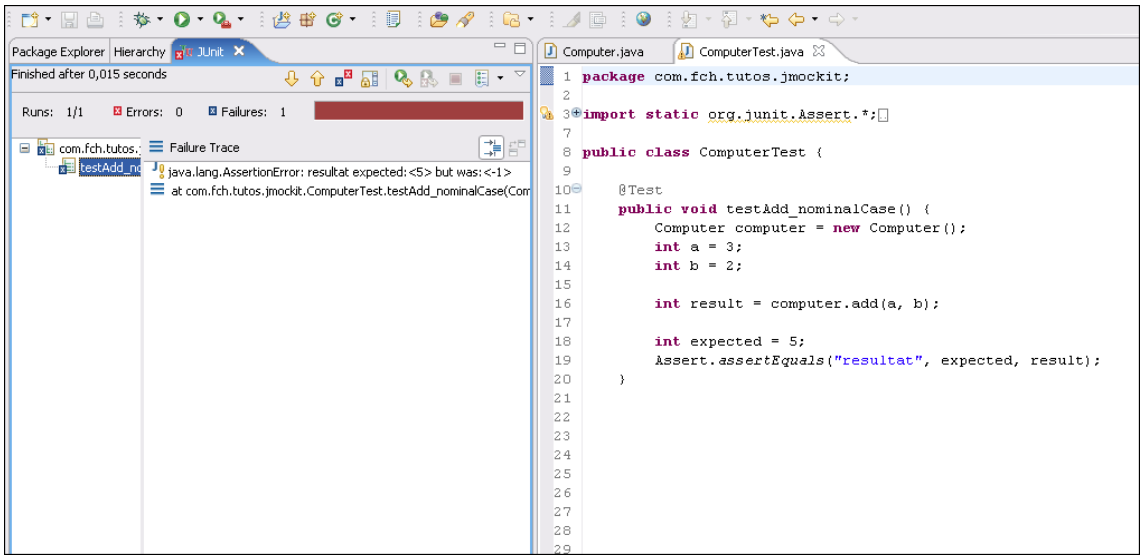


L'assistant génère le code suivant :

```
public class ComputerTest {

	@Test
	public void testAdd() {
		fail("Not yet implemented");
	}

}
```

L'annotation *@Test* indique que la méthode va être appelée en tant que test unitaire. Généralement, *une* méthode représente *un* cas de test : le cas où il y a des paramètres valides, le cas où le paramètre est mauvais, etc.

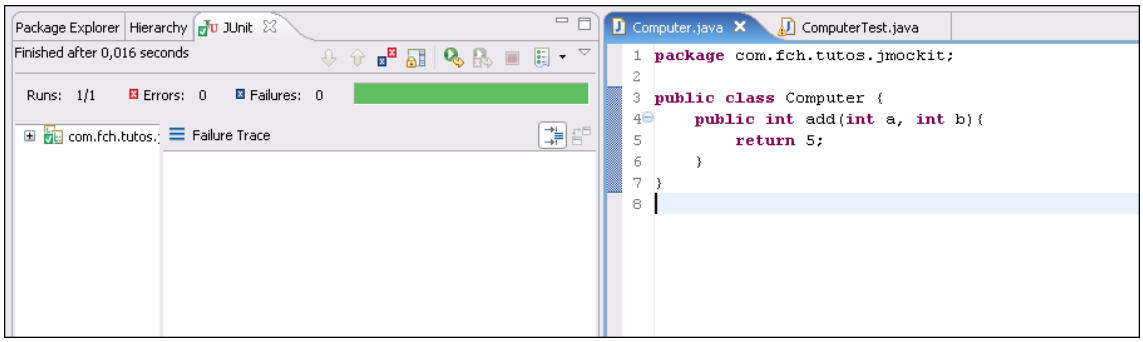
Nous commençons par implémenter le cas le plus courant, le cas « normal » et lançons le test.



Il échoue comme prévu. Dans l'assertion du test (*assertEquals*), mettre un message en premier paramètre et la valeur attendue en deuxième permet d'avoir un message d'erreur beaucoup plus clair « *java.lang.AssertionError: resultat expected:<5> but was:<-1>* ».

De nombreuses assertions sont fournies par Junit : *assertTrue*, *assertNotNull*, *assertArrayEquals*, etc... Les messages sont optionels pour toutes mais en mettre améliore la compréhension du test qui échoue.

Nous pouvons modifier l'implémentation pour le faire passer, en retournant 5 par exemple ;-)



C'est vert ! Un test « nominal » supplémentaire peut être ajouté pour s'assurer que ce n'est pas un hasard, en passant les paramètres 3 et -4 par exemple, et en s'assurant que -1 est retourné.

Nous procédons de la même façon pour écrire tous les tests : écriture du test, lancement du test qui échoue, correction en implémentant la méthode.

Entre deux tests qui passent, un commit peut être fait sur votre gestionnaire de version (SVN ou CVS) afin de sauvegarder les données.

Tester les cas inattendus

Les tests doivent couvrir les cas « normaux » mais il faut aussi songer au cas anormaux, et ce qui est attendu à ce moment là.

Imaginons que nous ayons besoin d'une autre méthode *add()* avec des chaines de caractères en paramètres. Nous créons la méthode :

```
public String add(String a, String b){
      return null;
```

```
}
```

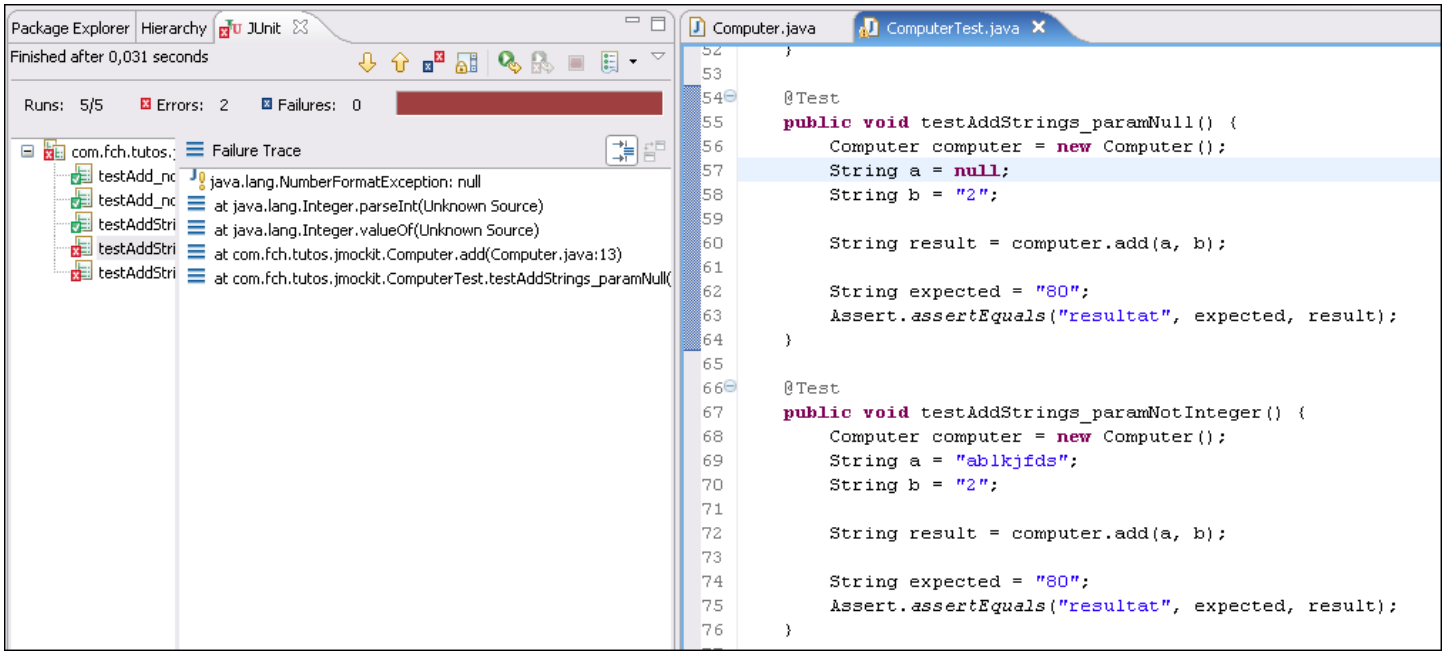
Nous ajoutons le test du cas nominal. Quelques commentaires sont ajoutés dans la classe de test pour plus de lisibilité :

```
// ================================================================================
// Tests for add(String,String)
// ================================================================================

@Test
public void testAddStrings_nominalCase3() {
	Computer computer = new Computer();
	String a = "78";
	String b = "2";

	String result = computer.add(a, b);

	String expected = "80";
	Assert.assertEquals("resultat", expected, result);
}
```

Nous implémentons la méthode afin de faire passer le test :

```
public String add(String a, String b){
	return String.valueOf(Integer.valueOf(a) + Integer.valueOf(b));
}
```

Que se passe-t-il si *a* est *null*? Si *a* n'est pas un entier? Nous écrivons les tests correspondant :



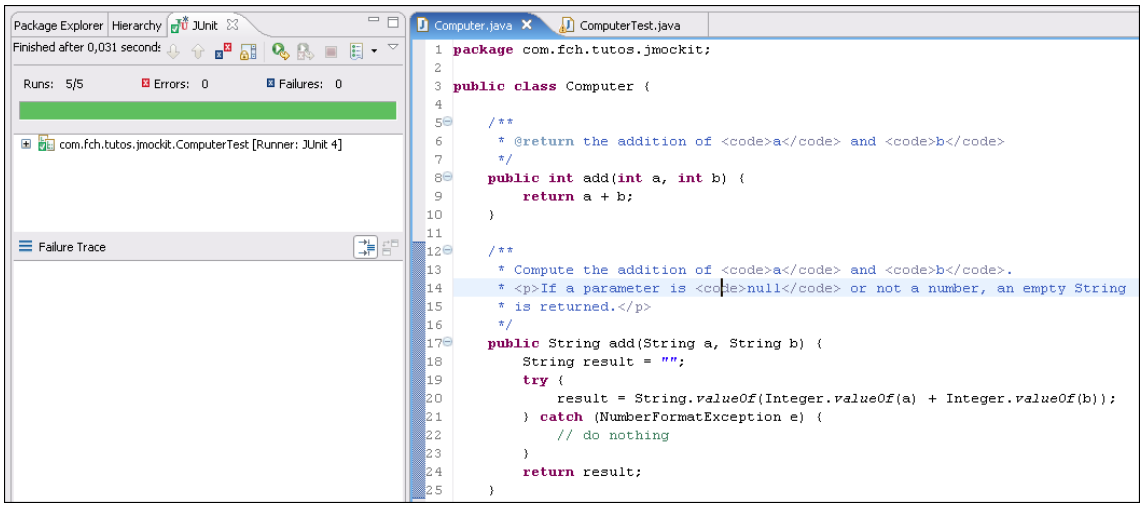
Des exceptions *NumberFormatException* sont lancées. Si c'était ce que nous voulions, nous le spécifierions dans le test :

```
@Test(expected=NumberFormatException.class)
public void testAddStrings_paramNull()
```

Comme nous souhaitons en réalité retourner une chaine vide dans ces cas là, nous changeons simplement la valeur de *expected* :

```
String expected = StringUtils.EMPTY;
```

Le test ne passe pas. Nous implémentons la méthode en conséquence et ajoutons ce cas particulier dans la *javadoc*. L'utilisateur de la méthode ne sera ainsi pas surpris de ce comportement.



Tester les appels de méthodes

Le client souhaite proposer un historique de tous les calculs possibles. Les tests prennent en compte cette demande :

```
	// ================================================================================
	// Tests for add(int,int)
	// ================================================================================

	@Test
	public void testAdd_nominalCase() {
		Computer computer = new Computer();
		int a = 3;
		int b = 2;

		int result = computer.add(a, b);

		int expectedResult = 5;
		Assert.assertEquals("resultat", expectedResult, result);
		String expectedHistory = "3 + 2\n";
		Assert.assertEquals("history",expectedHistory,
computer.getHistory().toString());
	}

	// ================================================================================
	// Tests for add(String,String)
	// ================================================================================

	@Test
	public void testAddStrings_nominalCase() {
		Computer computer = new Computer();
		String a = "3";
		String b = "2";

		String result = computer.add(a, b);

		String expectedResult = "5";
		Assert.assertEquals("resultat", expectedResult, result);
		String expectedHistory = "3 + 2\n";
```

```
		Assert.assertEquals("history", expectedHistory, computer.getHistory()
				.toString());
	}
```

Les deux méthodes *add* effectuant des calculs, nous avons dû ajouter le traitement de l'historique dans les deux tests.

D'une certaine façon, nous testons deux fois la même chose. Si finalement l'historique doit comprendre la date où l'opération a été effectuée, tous ces tests devront être modifiés. Souvent, nous avons tendance à moins factoriser les tests que le code métier alors qu'ils le méritent tout autant.

Pour éviter ces duplications, la fonctionnalité d'historisation pourrait etre externalisée dans une autre méthode, voire une autre classe, *HistoryManager*. C'est cette méthode et elle seule qui doit etre testée en tant qu'entrée et sortie.

```
	@Test
	public void testLogAddition() {
		HistoryManager manager = new HistoryManager();

		manager.logAddition(3, 2);

		String expectedHistory = "3 + 2\n";
		Assert.assertEquals("history", expectedHistory, manager.getHistory());
	}
```

Les méthodes de *Computer* n'ont plus qu'à tester ***l'appel de méthode***, et vérifier que les paramètres passés aux méthodes *add()* sont transmis à *logAddition*.

Computer contient désormais une interface *IHistoryManager*, implémentée par *HistoryManager*.

```
public class Computer {
	private IHistoryManager historyManager;

	public IHistoryManager getHistoryManager() {
		return historyManager;
	}

	public void setHistoryManager(IHistoryManager historyManager) {
		this.historyManager = historyManager;
	}
```

Nous vérifions que la méthode *add* de *Computer* appelle la méthode *logAddition* de son *IHistoryManager* :

```
	@Test
	public void testAdd_nominalCase() {
		Computer computer = new Computer();
		final int a = 3;
		final int b = 2;
		Mockery context = new Mockery();
		final IHistoryManager historyManager = context.mock(IHistoryManager.class);
		computer.setHistoryManager(historyManager);

		context.checking(new Expectations() {

			{
				one(historyManager).logAddition(a, b);
			}
		});

		int result = computer.add(a, b);

		context.assertIsSatisfied();
		int expectedResult = 5;
```

```
        Assert.assertEquals("resultat", expectedResult, result);
    }
```

La librairie JMock permet de créer des implémentations vides d'interface, des « mocks ». Nous pouvons ainsi setter cette coquille vide à *computer*. Ensuite, nous définissons les appels attendus sur ce mock. Ainsi :

```
one(historyManager).logAddition(a, b);
```

déclare que la méthode *logAddition* doit être appelée une et une seule fois, avec les paramètres *a* et *b*. Une fois que ces attentes (« expectations ») sont définies, nous pouvons appeler la méthode testée :

```
int result = computer.add(a, b);
context.assertIsSatisfied();
```

La méthode *assertIsSatisfied()* demande de ***vérifier*** les expectations définies préalablement : une erreur se produira si les conditions ne sont pas remplies.

Framework de mocks

JMock permet de mocker des interfaces, de vérifier les appels (l'ordre peut ou pas compter avec les *séquences*), vérifier les absences d'appels, valider les paramètres, etc. La cheat sheet du site officiel est très bien faite, vous y trouverez toutes les informations nécessaires..

Jmockit est aussi très puissant : il permet de mocker des classes, des initialiseurs static, des constructeurs, des méthodes privées... C'est le sujet du prochain tutorial :-)