Eclipse et JBoss

Développement d'applications J2EE professionnelles, de la conception au déploiement

Karim Djaafar

avec la contribution de

Olivier Salvatori

ISBN : 2-212-11406-0

EYROLLES

6

Any program feature without
an automated test simply doesn't exist.
Kent Beck

Tests unitaires et in-container avec JUnit et Cactus

La problématique de l'automatisation des tests unitaires est un ingrédient essentiel dans le cycle de développement. Comme tout développeur le sait d'expérience, l'écriture des tests est une tâche rébarbative. Nous souhaitons bien sûr les écrire, mais, pour une raison ou une autre, nous ne le faisons pas. Il en résulte un cercle vicieux : plus nous sommes sous pression, moins nous faisons de test, et le code devient de moins en moins stable.

Les tests permettent de réduire les bogues, de fournir une documentation appropriée pour les API du code testé et d'améliorer le design de l'application. Un des piliers de la démarche XP (eXtreme Programming) est l'intégration en continu *(continuous integration),* consistant à coder, tester et intégrer en même temps. Ce chapitre montre d'abord comment réaliser des tests à l'aide du framework JUnit, qui définit comment structurer les cas de tests et fournit les outils pour les lancer. Il se poursuit par la présentation du framework Cactus, qui permet une intégration en continu et peut être couplé avec l'outil Ant pour la construction des livrables.

Les tests unitaires

L'objectif premier d'un test unitaire, généralement effectué par un développeur de l'équipe J2EE, est d'extraire une portion de code isolée du système global en cours de développement et de vérifier que le résultat est celui attendu.

La taille de cette portion, ou « unité », de code, qui comprend des composants individuels, comme des classes Java, des servlets, voire des EJB, varie en grande partie selon la taille des fonctionnalités testées, qui peuvent aller d'une classe à un package.

Cette tâche d'extraction d'une unité de code doit bien sûr être effectuée avant son intégration au reste du système. Le coût de recherche d'un bogue et de sa correction après déploiement de l'application dépasse largement celui des tests qui auraient permis de le détecter en amont du développement. Les tests unitaires sont écrits et exécutés par le développeur d'un pan de l'application. Ils sont généralement de type « boîte noire », c'est-à-dire qu'ils sont conçus en utilisant les détails nécessaires à l'implémentation de telle ou telle fonctionnalité, que seul le développeur possède, et qui font partie de ce que l'on appelle communément la mémoire du projet.

Il est préférable d'écrire les tests unitaires d'un composant avant d'entamer son développement, en s'appuyant sur les spécifications du composant. Mieux le test unitaire est construit, plus le développeur est confiant dans la fiabilité de son code. Idéalement, le test anticipe et intègre les modifications intervenues dans les fonctionnalités du composant avant toute modification de son code. Bien entendu, le test unitaire échoue jusqu'à ce que le code associé au composant soit modifié.

Le support et l'encouragement au développement de tests unitaires réguliers conduit à d'importants gains de productivité dans l'équipe et de qualité du code produit.

Même si la démarche de test est souvent ressentie par les développeurs comme fastidieuse, voire inutile, il est recommandé de prendre le temps de définir une stratégie de test pour un projet de développement. L'existence d'un framework de test peut être d'un secours appréciable. Pour en savoir plus sur les tests fonctionnels de type *black-box,* voir l'ouvrage de référence sur le sujet, *Test Driven Development: By Example,* de Kent Beck, qui décrit dans le détail cette approche des tests et leur conception avant tout codage.

Le framework JUnit

JUnit *(www.junit.org)* est un framework Open Source permettant de développer et d'exécuter des tests unitaires en Java. Il a été conçu par Erich Gamma et Kent Beck, deux membres du célèbre Gang of Four (bande des quatre), auxquels on doit l'ouvrage classique *Design Patterns* et qui comptent parmi les promoteurs de la méthodologie de développement logiciel XP (eXtreme Programming).

Validés dans le référentiel de l'équipe en même temps que le code qu'ils sont censés tester, les tests unitaires forment la pierre angulaire de l'eXtreme Programming. Sans entrer dans le détail de cette méthodologie *(voir http://www.extremeprogramming.org),* toutes les

classes du système doivent être testées, et aucun code ne peut être livré sans les tests JUnit correspondants. Pour de plus amples précisions sur cette méthode de développement, reportez-vous à l'ouvrage *L'eXtreme Programming,* paru sur ce thème aux éditions Eyrolles.

Un test unitaire est donc constitué d'une collection de tests destinés à vérifier le comportement d'une unité de code au sein d'une classe. Une portion de code ne peut être testée isolément mais doit faire partie d'un environnement d'exécution.

Il se révèle pratique d'automatiser la série de tests unitaires de manière à vérifier que tout fonctionne normalement. JUnit testant votre classe par le biais d'un scénario, vous devez créer un scénario de test tenant compte des étapes suivantes :

1. Instanciation de l'objet.
2. Invocation des méthodes.
3. Vérification des assertions (méthode `assertFonctions`).

Tout échec dans l'exécution d'un test du scénario délivre une documentation adéquate pour le diagnostic.

Le développeur se charge de l'écriture du test proprement dit en sous-classant la classe `TestCase` (cas de test), qui est au cœur du framework JUnit. Ce dernier fournit de son côté la structure du cas de test et l'environnement permettant son exécution.

Dans JUnit, chaque élément de test doit être implémenté sous la forme d'une méthode déclarée *public void* et ne prenant aucun paramètre. Tous les cas de test doivent posséder un constructeur simple, avec un paramètre de type string utilisé comme nom du cas de test affiché dans la classe de log. Comme vous le verrez, Eclipse rend la création et l'utilisation des tests relativement triviales, même si vous devez toujours décider quoi tester et comment le tester.

L'intégration complète de JUnit comme brique logicielle à l'IDE Eclipse ajoute un argument de poids, s'il en fallait, au choix de cette plate-forme de développement.

Exemple de cas de test

L'extrait suivant est le cas de test JUnit le plus simple. Il fait partie des tests inclus dans la classe JUnit `TestCase`, mais vous pouvez en ajouter autant que vous le souhaitez. Vous le rencontrerez tout au long de ce chapitre pour illustrer les concepts à l'œuvre dans les tests avec JUnit :

```
import junit.framework.*; ←❶

Public class SimpleCasTest extends TestCase {←❷
// Constructeur de Test
    Public SimpleCasTest (String nom) {←❸
        Super (nom);
    }
    public void testSimpleCasTest () {←❹
        // Code de test à écrire
```

```
        int reponse = 2;
        assertEquals((1+1), reponse); ← ❺

    }
}
```

Le cœur du framework JUnit est la classe `junit.framework.TestCase` (repère ❶). Toutes les classes de test doivent hériter de cette classe :

```
import junit.framework.*;
```

La bibliothèque du framework JUnit (repère ❷) est indispensable pour utiliser les mécanismes associés :

```
Public class SimpleCasTest extends TestCase {
```

La classe `SimpleCasTest` (repère ❸) nécessite ses propres méthodes de test et doit dériver de la classe `TestCase` :

```
Public SimpleCasTest (String nom) {
       Super (nom);
}
```

À chaque test est associé un nom (repère ❹) de manière à distinguer le test générateur de l'erreur si le test échoue :

```
public void test SimpleCasTest () {
    // Code de test à écrire
    int reponse = 2;
            assertEquals((1+1), reponse);
}
```

La fonction `assertEquals` (repère ❺) est une des nombreuses assertions fournies avec l'interface JUnit `Assert`. Il existe de nombreuses variantes de la méthode `assertEquals` *(voir l'API javadoc associée),* qui traitent avec de nombreux types de paramètres, y compris les types de base, conteneur et chaîne. Son rôle est avant tout de comparer différents types de paramètres et de générer une exception interne, qui est utilisée par le framework pour reporter une erreur si l'un des deux paramètres est différent ou, en cas d'égalité, de retourner la fonction appelée.

Ce simple exemple a pour rôle de tester si l'expression 1 + 1 est égale à la valeur de réponse, mais cela peut aller jusqu'au test du résultat provenant d'une autre fonction.

Cas de test *(TestCase)*

Dans JUnit, chaque cas de test est implémenté sous la forme d'une méthode de type *public void* sans paramètre. Cette méthode est invoquée à partir du moteur d'exécution de JUnit défini dans le package **junit.textui.TestRunner.** Si le nom de la méthode commence par `test`, le moteur la récupère automatiquement et l'exécute. Cette fonctionnalité se révèle éminemment pratique lorsque vous possédez de nombreux cas de test possédant plusieurs méthodes de test.

Vous pouvez automatiser une série de tests répétitifs en utilisant l'objet JUnit `TestSuite` et en combinant et exécutant en même temps plusieurs tests.

Pour exécuter une suite de deux cas de test unitaire et les exécuter simultanément, utilisez les instructions suivantes :

```
TestSuite suite = new TestSuite ();
suite.addTest (new SimpleTest("ajouter") ) ;
suite.addTest (new SimpleTest("testDivideByZero"));
testResult result = suite.run ();
```

Les cas de test JUnit (`TestCase`) permettent l'utilisation de méthodes `setUp` pour initialiser les éventuelles variables ou objets. Cette méthode est en effet appelée avant chaque évaluation du test. La méthode `setUp` est souvent associée à `tearDown`. Cette dernière est appelée après l'évaluation de chaque test et peut être utilisée pour déréférencer les variables ou autres ressources du test, de manière que chaque test soit exécuté sans effet de bord sur les autres tests.

Dans l'exemple suivant, mis courtoisement à notre disposition par Mike Clark, l'un des auteurs de l'outil Open Source JUnitPerf, dédié à l'automatisation des tests de performance, remarquez l'utilisation des méthodes `setUp()` et `tearDown()` appelées avant et après chaque méthode `testXXX()` qui s'exécute :

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;
    private Product _defaultBook;

    /**
     * Construit un objet ShoppingCartTest avec le nom spécifié
     *
     * @param nom nom du cas de test
     */
    public ShoppingCartTest(String nom) {
        super(name);
    }

    /**
     * Configuration des tests.
     * Appelé avant chaque méthode de test
     *
     */
    protected void setUp() {

        _bookCart = new ShoppingCart();

        _defaultBook = new Product("Extreme Programming", 23.95);
```

```
        _bookCart.addItem(_defaultBook);
    }

    /**
     * Détruit la configuration du test
     *
     * Appelé après chaque méthode de test
     */
    protected void tearDown() {
        _bookCart = null;
    }

    /**
     * Test d'ajout de produit dans le caddy
     */
    public void testProductAdd() {

        Product newBook = new Product("Refactoring", 53.95);
        _bookCart.addItem(newBook);

        double expectedBalance = _defaultBook.getPrice() + newBook.getPrice();

        assertEquals(expectedBalance, _bookCart.getBalance(), 0.0);

        assertEquals(2, _bookCart.getItemCount());
    }

    /**
     * Teste le vidage du caddy
     */
    public void testEmpty() {

        _bookCart.empty();

        assertTrue(_bookCart.isEmpty());
    }

    /**
     * Teste la suppression d'un produit du caddy
     *
     * @throws ProductNotFoundException si le produit n'est pas dans le caddy
     */
    public void testProductRemove() throws ProductNotFoundException {

        _bookCart.removeItem(_defaultBook);

        assertEquals(0, _bookCart.getItemCount());

        assertEquals(0.0, _bookCart.getBalance(), 0.0);
    }
```

```
    /**
     * Teste la suppression d'un produit inconnu du caddy
     *
     * Ce test est un succès
     * si l'exception ProductNotFoundException est levée
     */
    public void testProductNotFound() {

        try {

            Product book = new Product("Ender's Game", 4.95);
            _bookCart.removeItem(book);

            fail("Should raise a ProductNotFoundException");

        } catch(ProductNotFoundException success) {
            // successful test
        }
    }

    /**
     *
     * Assemble et retourne une suite de tests
      *  pour toutes les méthodes de test de ce cas de test
     *
     * @return A non-null test suite.
     */
    public static Test suite() {


        TestSuite suite = new TestSuite();

       // A lancer alternativement.


        suite.addTest(new ShoppingCartTest("testEmpty"));
        suite.addTest(new ShoppingCartTest("testProductAdd"));
        suite.addTest(new ShoppingCartTest("testProductRemove"));
        suite.addTest(new ShoppingCartTest("testProductNotFound"));

        return suite;
    }

    /**
     * Lance le cas de test
     */
    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Installation et mise en route de JUnit

Après cette présentation des concepts, passons au volet pratique de l'installation de JUnit (version 3.8.1) et de l'exécution du test précédent. Vous verrez à la section suivante comment intégrer et utiliser JUnit dans Eclipse.

1. À partir du site *www.junit.org,* téléchargez la distribution de JUnit sous forme de fichier ZIP, et décompressez-la sous un de vos lecteurs, **d:\junit** par exemple.
2. Ajoutez **junit.jar** à votre variable `classpath` à partir du panneau de configuration Windows :

   ```
   set classpath=%classpath%;INSTALL_DIR\junit3\junit.jar
   ```

   où `INSTALL_DIR` désigne le répertoire d'installation de JUnit.
3. Copiez l'exemple dans votre éditeur Java favori (EditPlus ou emacs), et sauvegardez-le sous le nom **SimpleCasTest.java.**

À présent, vous allez générer les affichages correspondant à l'exécution de ce cas de test. Il existe trois sortes d'interfaces pour le TestRunner JUnit, qui est l'interface d'exécution des tests et de collecte des résultats correspondants de JUnit :

- `TextUI`, qui fournit une interface de sortie texte (console de sortie).
- `AwtUI`, qui fournit une interface de sortie utilisant la bibliothèque Java AWT (Abstract Windows Toolkit).
- `SwingUI`, qui fournit une interface de sortie utilisant la bibliothèque Java Swing.

La version graphique de l'outil TestRuner JUnit se présente sous la forme illustrée à la figure 6.1, avec les champs suivants :

- Un champ de saisie contenant le nom de la classe avec une méthode `Suite`.
- Un bouton d'exécution pour lancer le test.
- Une barre de progression, qui passe du vert en cas de succès au rouge en cas d'échec.
- Une liste des tests qui ont échoué.

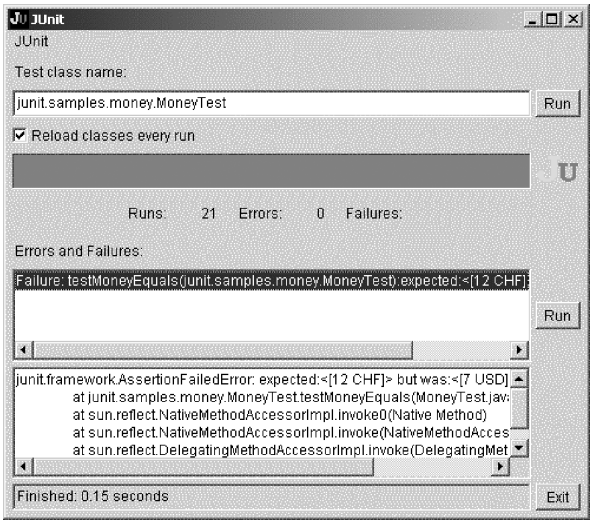
Dans l'exemple **SimpleCasTest.java,** vous utilisez la sortie `TextUI` sous DOS. Vous utiliserez le type de sortie `Swing` lors de la configuration dans Eclipse.

Les lignes suivantes permettent de compiler et exécuter la classe de test :

```
D:\junit3.8.1\samples>java junit.textui.TestRunner SimpleCasTest
.
Time: 0

OK (1 test)
```

JUnit TestRunner affiche le résultat en gras *(voir figure 6.1)* pour indiquer l'exécution correcte du test.

Figure 6.1
La version graphique du TestRuner de JUnit



L'échec du test s'affiche dans la console DOS de la façon suivante après compilation et modification du test `assertEquals` :

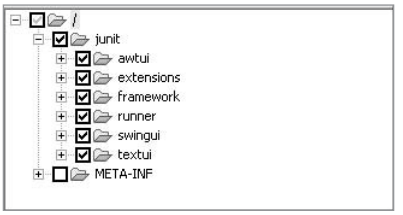
```
D:\junit3.8.1\samples>java junit.textui.TestRunner SimpleCasTest
.F
Time: 0,01
There was 1 failure:
1) testSimpleCasTest(SimpleCasTest)junit.framework.AssertionFailedError: expected
d:<2> but was:<3>
        at SimpleCasTest.testSimpleCasTest(SimpleCasTest.java:11)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Configuration de JUnit dans Eclipse

Vous allez à présent configurer et démarrer un projet JUnit dans Eclipse (nous supposons évidemment que vous disposez d'une installation JUnit proprement installée et configurée) :

1. Créez un nouveau projet Java appelé **junit,** *via* Fichier, Nouveau Projet et Projet Java.
2. Sélectionnez Fichier puis Importer et Fichier zip.
3. Cliquez sur Suivant.
4. Cliquez sur le bouton Parcourir, et ouvrez le fichier **src.jar** qui se trouve sous l'installation de JUnit pour importer les ressources dans votre espace local.
5. Sélectionnez les ressources comme illustré à la figure 6.2.

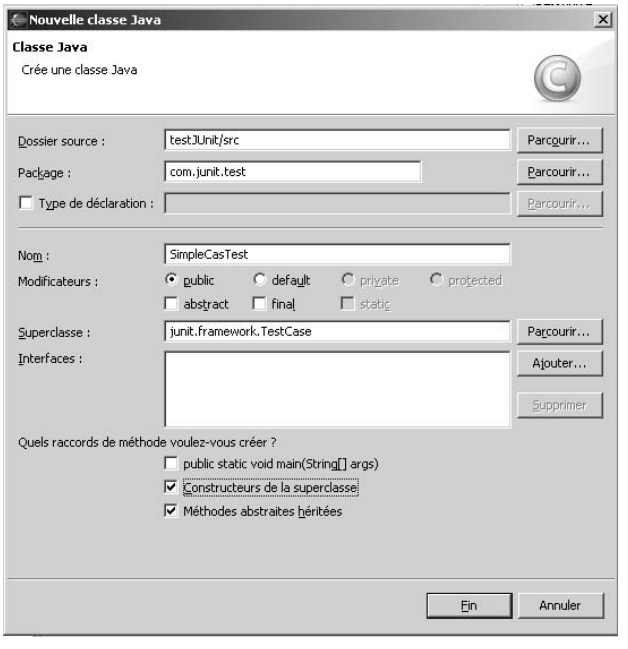
Figure 6.2
Ressources JUnit à intégrer dans l'espace local du projet



6. Sélectionnez la destination des ressources importées dans le dossier **junit/src** (bouton Parcourir).
7. Cliquez sur Fin. Les ressources du framework JUnit sont désormais disponibles dans Eclipse.
8. Si sous souhaitez disposer de la documentation javadoc des API JUnit, vérifiez dans le menu Préférences d'Eclipse que l'emplacement de la commande **javadoc** est bien spécifié (**d:\jdk1.3.1_04\bin\javadoc.exe,** par exemple), puis, dans le menu contextuel du projet (Propriétés de JUnit puis Emplacement javadoc), sélectionnez l'emplacement **file:/d:/junit3.8.1/javadoc/.**

   Vous allez à présent créer un nouveau projet Java contenant l'exemple de la section précédente, appelé, par exemple, **TestJunit.**
9. Ajoutez la référence au précédent projet JUnit dans l'option Propriétés du projet **TestJunit** (onglet Projets).

Figure 6.3
Création de la classe de test



10. Créez une nouvelle classe dans ce projet (Nouveau, Classes), la classe `SimpleCasTest`, selon les paramètres illustrés à la figure 6.30.

11. Cliquez sur Fin. Le code ci-dessous est généré (nous y avons ajouté le corps de la méthode `SimpleCasTest` de la section précédente) :

```
package com.junit.test;

import junit.framework.TestCase;

/**
 * @author user
 *
 * To change this generated comment edit the template variable "typecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class SimpleCasTest extends TestCase {

/**
 * Constructor for SimpleCasTest.
 */
public SimpleCasTest() {
    super();
}

/**
 * Constructor for SimpleCasTest.
 * @param name
 */
public SimpleCasTest(String name) {
    super(name);
}
    public void testSimpleCasTest () {
        // Code de test à écrire
        int reponse = 2;
        assertEquals((1+1), reponse);

    }

}
```

12. Dans la vue Package, recherchez **junit.textui.TestRunner.java,** et double-cliquez pour l'ouvrir dans un éditeur (le code source de la classe principale d'exécution JUnit est alors disponible).

13. Dans la vue Structure, notez que la classe `TestRunner` comporte une icône signalant qu'elle définit une méthode `main`, comme illustré à la figure 6.4.

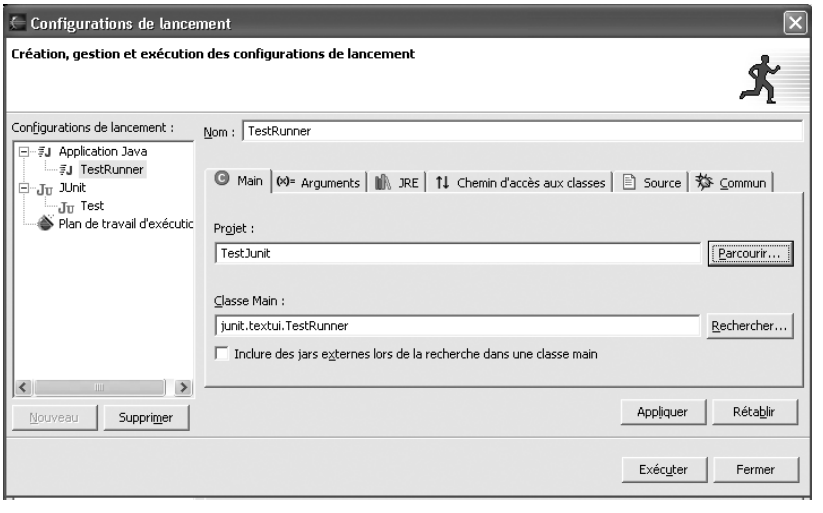
Figure 6.4
Vue Structure de la classe TestRunner

14. À l'aide du bouton Exécuter de la barre d'outils, sélectionnez Application Java dans le sous-menu Exécuter en tant que. Cette opération lance la classe dans l'éditeur actif ou la classe sélectionnée dans le navigateur sous la forme d'une application Java.
15. Le programme s'exécute, et le message illustré à la figure 6.5 apparaît dans la vue Console pour signaler que vous devez indiquer un argument d'exécution.

Figure 6.5
La vue Console après exécution de la classe TestRunner

16. Dans le menu Exécuter de la barre d'outils, sélectionnez Exécuter.
17. La boîte de dialogue Configurations de lancement s'affiche, comme illustré à la figure 6.6, avec la configuration de TestRunner sélectionnée. Lorsque vous avez exécuté le programme à l'aide du raccourci Application Java, une configuration de lancement a été automatiquement créée avec les paramètres par défaut pour lancer la classe sélectionnée.

Figure 6.6
Configuration de lancement

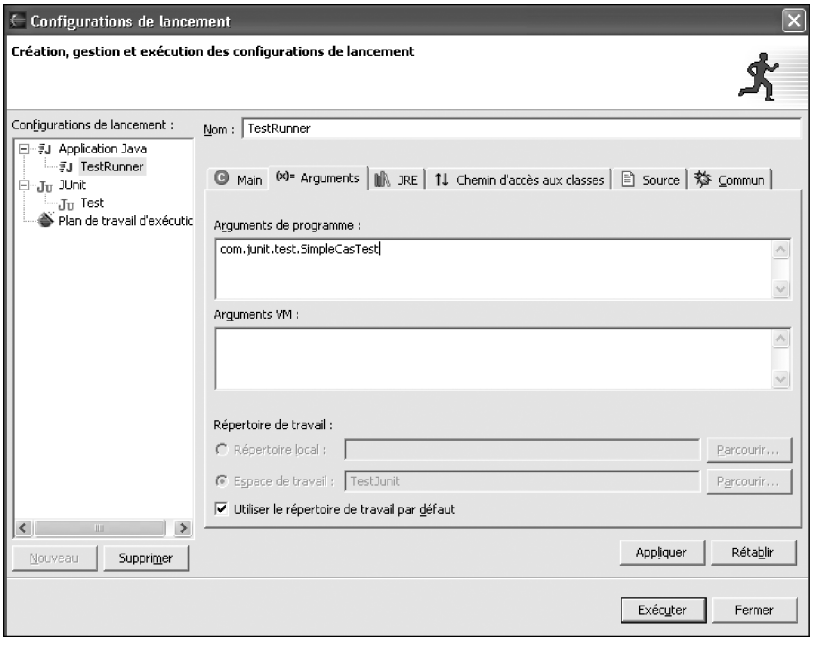


18. Cliquez sur l'icône ci-contre, puis sélectionnez Exécuter en tant que puis Test JUnit *(voir figure 6.7)*.

Figure 6.7
Configuration de lancement du test avec JUnit (1/2)

19. Sélectionnez l'onglet Arguments, et entrez **com.junit.test.SimpleCasTest** dans la zone d'arguments du programme illustrée à la figure 6.8.

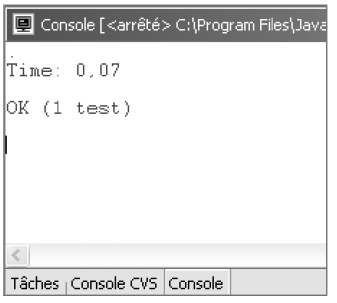
Figure 6.8
Configuration de lancement du test avec JUnit (2/2)



20. Cliquez sur Exécuter. Cette fois, le programme s'exécute correctement et indique le nombre de tests effectués *(voir figure 6.9)*.

21. Reprenez le même exemple avec la version graphique de l'outil JUnit en sélectionnant Test JUnit à partir du menu Exécuter en tant que. Vous devez obtenir le résultat illustré à la figure 6.10.

Figure 6.9
Résultat de la console



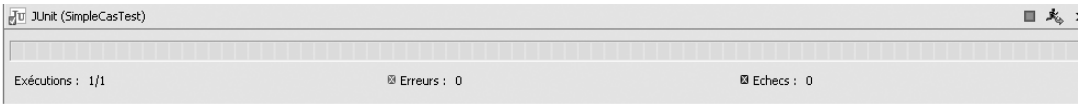


Figure 6.10
Résultat de l'exécution du test avec la version graphique intégrée

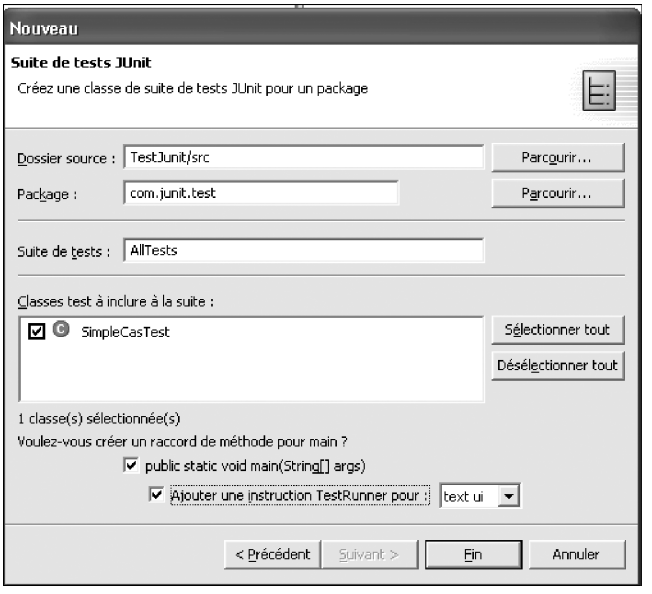
Création et mise à jour de suites de tests JUnit

Dans Eclipse, vous pouvez créer une suite de tests JUnit.

Procédez de la façon suivante :

1. Sélectionnez un package contenant des tests pour lesquels vous voulez créer une suite de tests.
2. Dans la barre de menus, sélectionnez Fichier, puis Nouveau et Autre.
3. Cliquez sur le bouton Ouvrir l'assistant Nouveau de la barre d'outils.
4. Sélectionnez Java puis JUnit dans la sous-fenêtre de gauche et Suite de tests dans celle de droite.
5. Cliquez sur Suivant. L'assistant de création apparaît, comme illustré à la figure 6.11.
6. Dans la zone Dossier source, entrez le nom du dossier source dans lequel la suite de tests doit être créée.
7. Dans la zone Package, entrez le nom du package dans lequel la suite de tests doit être créée. Si vous n'indiquez pas de valeur, c'est le package par défaut qui est utilisé.
8. Dans la zone Suite de tests, entrez le nom de la classe de la suite de tests (`AllTests` par défaut).
9. Dans la liste de classes qui s'affiche, sélectionnez les classes de test pour lesquelles vous voulez créer la suite de tests (`SimpleCasTest`).
10. Cochez l'option autorisant le raccourci de la méthode `main` *(voir figure 6.11)* utile pour avoir un point d'entrée `main` dans le programme de lancement, et sélectionnez le mode de représentation `text ui`, ou affichage texte.

Figure 6.11
Création d'une suite de tests avec l'assistant de création JUnit



11. Cliquez sur Fin. L'assistant génère le fichier suivant, qui représente la suite de tests à dérouler. Il insère deux marqueurs spéciaux (`//$JUnit-BEGIN$` et `//$JUnit-END$`) dans la classe de la suite de tests pour permettre à l'assistant de mettre à jour les classes existantes. Il est recommandé de ne pas modifier le code placé entre les marqueurs :

```
package com.junit.test;

import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * @author user
 *
 * To change this generated comment edit the template variable "typecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class AllTests {

public static void main(String[] args) {
    junit.textui.TestRunner.run(AllTests.class);← ❶
}

public static Test suite() {
    TestSuite suite = new TestSuite("Test for com.junit.test");
    //$JUnit-BEGIN$
    suite.addTest(new TestSuite(SimpleCasTest.class));
```

```
        //$JUnit-END$
        return suite;
    }
}
```

12. Remplacez la ligne de code du repère ❶ par la ligne suivante :

```
junit.textui.TestRunner.run(suite());
```

13. Relancez Exécuter en tant que, puis choisissez Application Java.

La vue console affiche le résultat du test.

Intégration de JUnit avec Ant

JUnit et Ant forment un couple complémentaire. Ant automatise le processus de construction et le déploiement, et JUnit automatise les tests. Ant peut ainsi automatiser la construction, le déploiement et le processus de test. Il dispose d'ailleurs d'un certain nombre de balises dédiées pour supporter JUnit. La balise `<junitreport>`, par exemple, permet la génération de rapports de test au format HTML ainsi qu'une personnalisation très riche des documents *via* XSLT.

Le célèbre parseur Xalan de la fondation Apache *(http://xml.apache.org/xalan-j/index.html)* permet de transformer des documents XML en HTML (le JDK 1.4 contient une version de Xalan-J 2.*x* et la version 1.5 une version compatible XSLTC).

Pour utiliser JUnit, vous pouvez vous aider des bibliothèques JAR du framework JUnit. Pour positionner ces dernières dans le CLASSPATH de Ant, le moyen le plus simple est de les copier directement dans le répertoire **ANT_HOME/lib, ANT_HOME** désignant le répertoire d'installation de Ant.

Une fois les fichiers JAR requis installés dans Ant, en particulier le fichier **junit.jar,** vous pouvez construire et tester le programme selon le canevas suivant :

```
<project name= "junitTest" default="test">
        <target name="init">
                <property name="outdir" value="/tmp/junitTest" />
        </target>

        <target name="prepare" depends="init">
                <mkdir dir="${outdir}"/>
        </target>

        <target name="compile" depends="prepare">
             <javac srcdir="." Destdir="${outdir}"
                        classpath="junit.jar" />
        </target>

        <target name="test" depends="compile">
            <junit printsummary="true" >

                 <test name="com.junit.test.AllTests" />
```

```
                <classpath>
                    <pathelement location="${outdir}" />
                </classpath>
            </junit>
        </target>
</project>
```

Dans ce fichier de construction **build.xml,** le nom du projet est **junitTest.** Il possède les cibles habituelles `init`, `prepare`, `compile` et `test`, cette dernière cible étant spécifiquement dédiée aux tests qui dépendent de la cible `compile`. La cible `init` crée une propriété `outdir` qui contient l'emplacement du répertoire de sortie. La cible `prepare` crée les répertoires de sortie, et la cible `compile`, comme à son habitude, compile le code source de l'application sur le répertoire de sortie `outdir`.

Le point intéressant à analyser est la cible `test`. Elle utilise la tâche JUnit pour exécuter un test créé avec ce framework pour exécuter `com.junit.test.AllTests` décrit à la section précédente. La tâche JUnit possède un sous-élément `test` (en gras dans l'extrait). Cet élément est utilisé pour positionner le nom de la classe du cas de test à exécuter (attribut `name`). Remarquez l'utilisation de la balise `printsummary`, qui affiche une ligne de statistique pour chaque cas de test exécuté.

Voici le résultat de l'exécution :

```
C:\test>ant
Buildfile: build.xml

init:

prepare:

compile:

test:
    [junit] Running junit.samples.AllTests
    [junit] Tests run: 1, Failures: 0, Errors: 1, Time elapsed: 0 sec

    [junit] TEST junit.samples.AllTests FAILED

BUILD SUCCESSFUL
Total time: 2 seconds
C:\test>
```

Ce rapport de test est pour le moins concis. Vous souhaiteriez sans doute en savoir plus sur le test qui a échoué. Vous disposez heureusement d'une sous-balise `<formater>`, accompagnée de formateurs prédéfinis (attribut `type`) :

- `xml` : affiche le résultat au format XML.
- `plain` : affiche le résultat en format plein texte.
- `brief` : affiche le contenu détaillé du test qui a échoué.

Modifiez en conséquence le fichier **build.xml,** en précisant que le résultat est affiché *via* la sortie standard et non dans un fichier (booléen `usefile`) :

```
<project name= "junitTest" default="test">
        <target name="init">
                <property name="outdir" value="/tmp/junitTest" />
        </target>

        <target name="prepare" depends="init">
                <mkdir dir="${outdir}"/>
        </target>

        <target name="compile" depends="prepare">
             <javac srcdir="." Destdir="${outdir}"
                         classpath="junit.jar" />
        </target>

        <target name="test" depends="compile">
            <junit printsummary="true" >

          <formatter type="brief" usefile="false"/>

          <test name="com.junit.test.AllTest" />

          <classpath>
             <pathelement location="${outdir}" />
          </classpath>

              </junit>
         </target>
</project>
```

Voici le résultat de l'exécution du test, cette fois un peu plus loquace sur la raison de l'échec :

```
C:\test>ant
Buildfile: build.xml

init:

prepare:

compile:
    [javac] Compiling 2 source files to \tmp\junitTest

test:
    [junit] Running com.junit.test.AllTest
    [junit] Tests run: 1, Failures: 0, Errors: 1, Time elapsed: 0 sec

    [junit] Testsuite: com.junit.test.AllTest
```

```
    [junit] Tests run: 1, Failures: 0, Errors: 1, Time elapsed: 0 sec

    [junit] Null Test:  Caused an ERROR
    [junit] com.junit.test.AllTest
    [junit] java.lang.ClassNotFoundException: com.junit.test.AllTest
    [junit]     at java.net.URLClassLoader$1.run(URLClassLoader.java:199)
    [junit]     at java.security.AccessController.doPrivileged(Native Method)
    [junit]     at java.net.URLClassLoader.findClass(URLClassLoader.java:187)
    [junit]     at java.lang.ClassLoader.loadClass(ClassLoader.java:289)
    [junit]     at java.lang.ClassLoader.loadClass(ClassLoader.java:235)
    [junit]     at java.lang.ClassLoader.loadClass(ClassLoader.java:235)
    [junit]     at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:302)

    [junit]     at java.lang.Class.forName0(Native Method)
    [junit]     at java.lang.Class.forName(Class.java:219)


    [junit] TEST com.junit.test.AllTest FAILED

BUILD SUCCESSFUL
Total time: 3 seconds
```

À titre d'exemple, voici le même code, dont le test a été un succès, appliqué à l'exemple **VectorTest.java** fourni avec la distribution JUnit (modifiez en conséquence la balise `<test name="junit.samples.VectorTest" />` :

```
Buildfile: build.xml

init:

prepare:

compile:
    [javac] Compiling 42 source files to \tmp\junitTest

test:
    [junit] Running junit.samples.VectorTest
    [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,01 sec

    [junit] Testsuite: junit.samples.VectorTest
    [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0,01 sec

    [junit] Testcase: testCapacity took 0,01 sec
    [junit] Testcase: testClone took 0 sec
    [junit] Testcase: testContains took 0 sec
    [junit] Testcase: testElementAt took 0 sec
    [junit] Testcase: testRemoveAll took 0 sec
    [junit] Testcase: testRemoveElement took 0 sec


BUILD SUCCESSFUL
Total time: 13 seconds
```

Il est possible d'afficher le résultat au format HTML, beaucoup plus facile à échanger. Il suffit de générer un fichier de rapport de test au format XML, *via* l'attribut `type`, puis d'utiliser la tâche Ant prédéfinie `junitreport`. Ajoutez à la suite de la balise `junit` le code suivant :

```
<junitreport todir="./reports">

    <fileset dir=".">
      <include name="TEST-*.xml"/>
    </fileset>

    <report format="frames" todir="./report/html"/>

    </junitreport>
```

Le résultat des tests est cette fois formaté sous une forme beaucoup plus sympathique, comme l'illustre la figure 6.12.

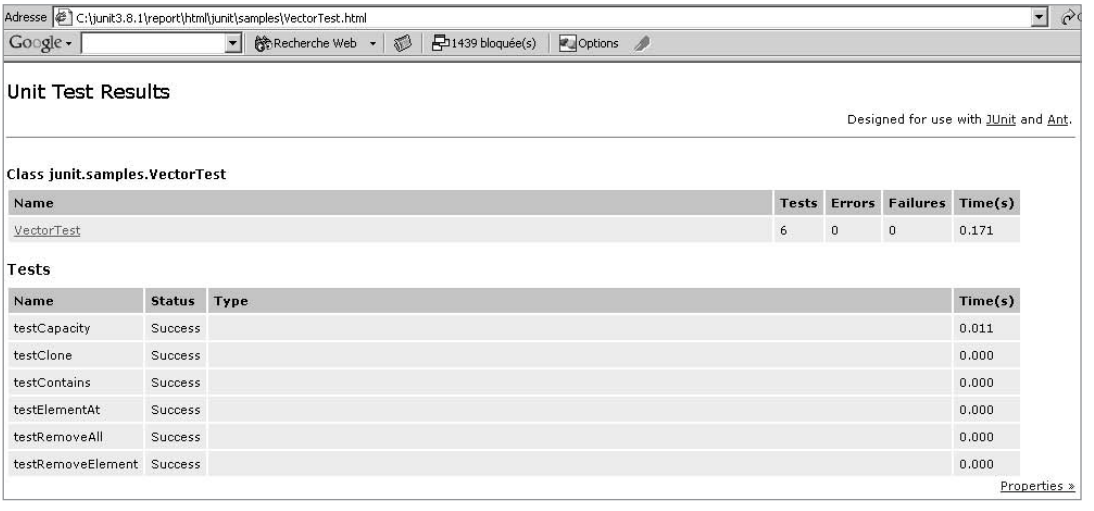


Figure 6.12
Résultat du rapport de test au format HTML

Extensions à JUnit

Un certain nombre de frameworks de test et de packages additionnels ont fait leur apparition sur le marché de l'Open Source. Ils permettent de couvrir l'ensemble des technologies Java, notamment celles dédiées aux tests des composants de la partie serveur *(server-side),* avec les JSP, les servlets et les EJB, composants que le framework JUnit peut difficilement tester seul.

Typologie des frameworks de test unitaire

Les frameworks de test unitaire peuvent être classés selon trois catégories de tests : les tests de la logique du code, les tests unitaires et d'intégration et les tests fonctionnels :

- **Framework de test de la logique du code.** La meilleure stratégie pour ces tests est d'utiliser des objets *mock*, aussi appelés objets bouchons. Un objet mock est un objet de simulation qui retourne des valeurs prévisibles vers l'objet testé en fonction de la connaissance que l'on a du comportement de l'objet. L'objectif principal des objets mock est de tester unitairement une méthode, isolée du domaine fonctionnel, en utilisant une copie des objets simulés plutôt que les objets réels. L'approche de test MO (Mock Object) est exhaustive et très générale. Elle s'adapte à tout type de test unitaire (test unitaire de servlet, test JDBC, test Struts, etc.). Les tests MO sont à distinguer de la stratégie *in-container,* qui repose sur le test du conteur réel et est adoptée par certains frameworks, comme Cactus. Pour plus de détails sur l'approche MO, reportez-vous au site dédié *http://www.mockobjects.com.*

- **Framework de test unitaire et d'intégration.** Le framework Cactus entre dans cette catégorie en simulant l'interaction avec le conteneur des objets testés (servlets, Beans d'entreprise, etc.).

- **Framework de test fonctionnel.** Permet de tester les valeurs retournées par le serveur.

Pour une couverture exhaustive des tests de votre application (test unitaire, test de recette applicative, test d'intégration), vous devriez utiliser idéalement les trois types de frameworks. Le framework Cactus a été conçu pour couvrir les besoins de test du deuxième type. En combinaison avec JUnit, il constitue cependant un bon compromis pour les deux autres catégories.

Voici quelques exemples d'outils et de frameworks Open Source de test J2EE parmi les plus populaires :

- **Cactus.** Framework *server-side* (côté serveur) pour le test des servlets, des EJB et des autres technologies serveur (bibliothèques de tags). Il est disponible sur le site de la fondation Apache *(http://jakarta.apache.org/cactus/).*

- **HttpUnit/HtmlUnit/jWebUnit.** Framework pour le test des applications Web mais pour la partie fonctionnelle, côté utilisateur. Le processus consiste à choisir un cas d'utilisation et d'écrire le test HttpUnit/HtmlUnit/jWebUnit. Le test doit prouver que le scénario associé au cas d'utilisation fonctionne selon les interactions de l'utilisateur, qu'il réagit au système et que ce dernier fonctionne de manière prédictive lorsqu'un scénario associé à un cas d'utilisation échoue.

- **JUnitPerf.** Framework agissant comme « décorateur », ou wrapper, des cas de test JUnit, en vérifiant que ces derniers respectent les critères de performance. Par exemple, vous pouvez tester que chaque page Web se charge en moins de trois secondes et que la moyenne du temps de chargement est de moins de cinq secondes lorsque le site gère plus de mille utilisateurs simultanés. JUnitPerf vous permet d'effectuer assez facilement

des tests en charge et de voir ces tests échouer s'ils ne répondent pas au bout d'une certaine durée, définie au préalable.

- **JMeter.** Outil Open Source de la fondation Apache de mesure des performances du Web permettant d'analyser la performance et le comportement d'un serveur soumis à des charges différentes. Il peut être utilisé pour simuler une lourde charge sur un serveur, le réseau ou l'objet à évaluer. Des scénarios, appelés « plans de test » doivent être définis pour effectuer les tests. JMeter est une application JAVA qui offre de nombreuses fonctionnalités :
  - génération de scénario de tests ;
  - enregistrement à partir d'un butineur ;
  - support du protocole sécurisé SSL ;
  - support de proxy.
- **StrutsTestCase.** Extension des classes `TestCase` de JUnit conçue pour les tests du framework Struts. Disponible à l'adresse *http://strutstestcase.sourceforge.net/*.
- **Maven.** Framework issu du sous-projet Turbine de la fondation Apache et hébergé sur le site de la même fondation *(http://maven.apache.org)*. Son principal objectif est de simplifier et de standardiser le processus de construction selon une démarche « agile » et de définir des règles de développement claires tout en permettant une visualisation cohérente de l'information du projet par l'équipe.

Le framework Cactus

Cactus est conçu pour l'écriture de tests unitaires d'intégration qui s'exécutent au sein d'un moteur de servlets (Tomcat, Resin, etc.). Pour cette raison, ces tests sont dits in-container, c'est-à-dire dans le conteneur.

Au contraire du framework HTTPUnit, qui teste seulement *via* une interface HTTP, il traite des objets Java et offre un contrôle des méthodes lorsqu'un cas de test Cactus est construit.

Architecture du framework Cactus

Les composants et les mécanismes d'invocation qui constituent le framework Cactus sont illustrés à la figure 6.13.

L'un des premiers objectifs du framework Cactus est de pouvoir effectuer des tests unitaires des méthodes Java côté serveur, qui utilisent des objets comme `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

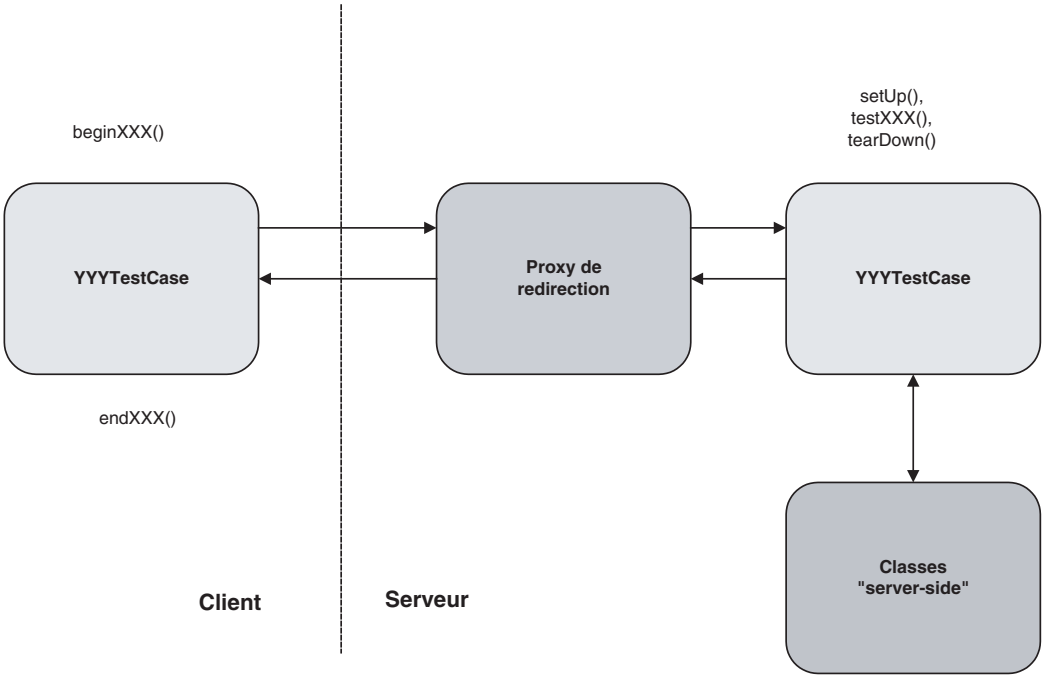


Figure 6.13
Architecture du framework CACTUS

Structure de la classe *TestCase*

Pour tester des objets *server-side,* la classe à tester doit étendre la classe `org.apache.commons.cactus.ServletTestCase`. Pour tester les objets JSP, la classe à tester doit étendre la classe `org.apache.commons.cactus.JspTestCase`.

Pour simuler différents environnements fondés sur les servlets/JSP, le framework Cactus fournit l'accès aux objets récapitulés au tableau 6.1.

Tableau 6.1 Accès des objets Web par le framework Cactus

Objet	Type
`request`	`HttpServletRequest` `org.apache.cactus.WebRequest`
`response`	`HttpServletResponse` `org.apache.cactus.WebResponse`
`session`	`HttpSession`
`config`	`ServletConfig`
`pageContext`	`PageContext`
`out`	`JspWriter`

Comme pour un cas de test JUnit standard, les méthodes Cactus standards suivantes sont définies :

- Un constructeur avec un paramètre unique (le nom du test).
- Une méthode `main()` pour démarrer le lanceur de test JUnit.
- Une méthode `suite()` pour lister les tests qui doivent être exécutés par la classe de test.

Comme pour JUnit, les méthodes `setUp()` et `tearDown()` peuvent être définies optionnellement. Comme pour JUnit également, la méthode `main` responsable des tests est `testXXX()`. Chaque cas de test `XXX` doit avoir une méthode `testXXX()` définie.

La méthode `testXXX()` peut contenir les éléments suivants :

- instanciation de la classe à tester ;
- initialisation de chaque objet serveur (comme les objets `HttpServletRequest`, `ServletConfig`, `HttpSession`) ;
- appel de la méthode appropriée de la classe à tester ;
- appel des méthodes `assert` JUnit standards, comme `asserts(…)`, `assertEquals[…]`, `fail[…]`, etc., afin de vérifier que le test est OK.

Pour chaque cas de test `XXX`, une méthode correspondante `beginXXX()` peut être définie optionnellement. L'initialisation de paramètres HTTP est en principe effectuée à travers cette méthode. De manière symétrique, pour chaque cas de test `XXX`, une méthode correspondante `endXXX()` peut être définie optionnellement, cette méthode étant utilisée pour vérifier les paramètres HTTP retournés du cas de test.

Mise en œuvre du framework Cactus avec Eclipse

À l'heure ou nous écrivons ces lignes, il n'existe pas de plug-in Cactus pour Eclipse disponible sur le site dédié au produit, celle qui s'y trouvait ayant été retirée suite à des problèmes de fiabilité. Vous en serez quitte pour une adaptation de la distribution Cactus dans l'environnement Eclipse en attendant la sortie d'une version plugable.

Nous supposons installé et décompressé dans un répertoire de votre choix le package **jakarta-cactus-13-1.6.1.zip,** téléchargeable depuis le site de Jakarta *(http://jakarta.apache.org/cactus).*

Configuration du framework Cactus dans Eclipse

1. Créez un nouveau projet Java dans Eclipse, et nommez-le **CactusWebTest.**
2. À partir du menu contextuel du projet, sélectionnez Propriétés.
3. Dans la zone de gauche de la boîte de dialogue Propriétés du projet, choisissez Chemin de compilation Java.

4. Dans l'onglet Bibliothèques, ajoutez les bibliothèques Cactus associées *(voir figure 6.14) :*

   – **aspectjrt-1.1.1.jar**

   – **cactus-1.6.1.jar**

   – **commons-httpclient-2.0.jar**

   – **commons-logging-1.0.3.jar**

   – **junit-3.8.1.jar**

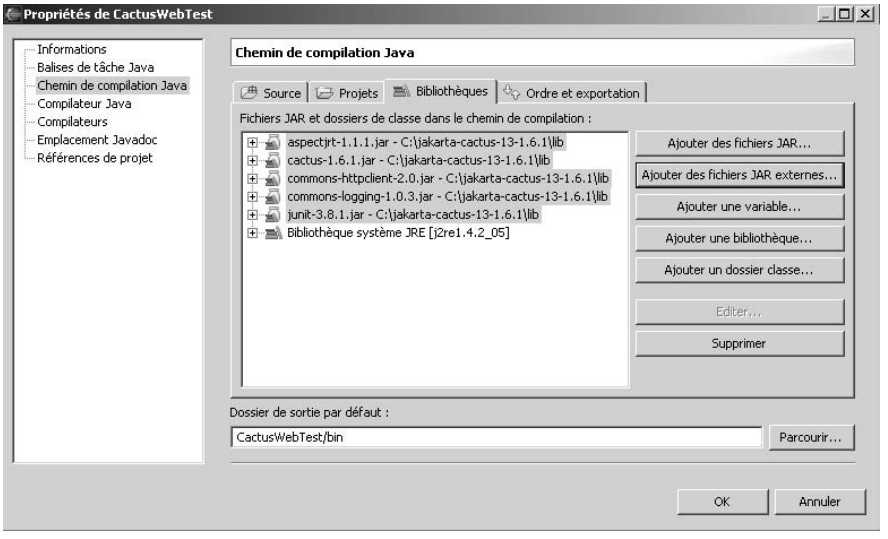


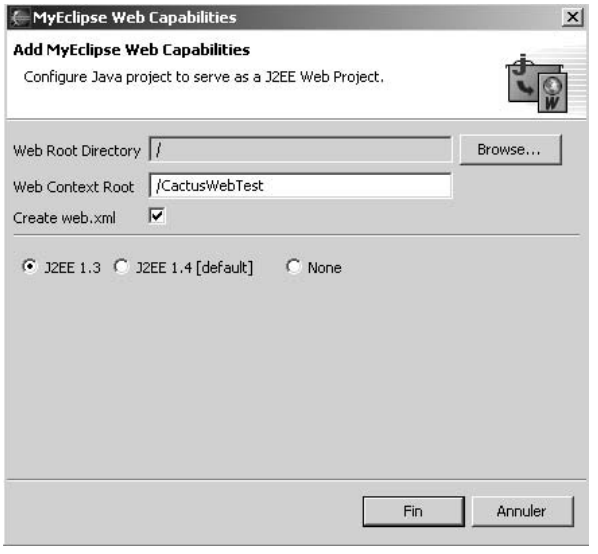
Figure 6.14
Configuration du CLASSPATH pour la mise en œuvre du framework Cactus

5. Étant donné que vous allez tester les fonctionnalités Web de Cactus, vous devez ajouter le support Web en cliquant sur MyEclipse dans la barre de menus d'Eclipse puis en sélectionnant Add Webproject Capabilities.

> **Support Web**
>
> Vous pouvez ajouter le support Web « à la main », sans passer par l'environnement MyEclipse, en copiant les bibliothèques requises pour le support des servlets et fournies dans Tomcat (fichier **servlet-api.jar** notamment) et en créant « votre » fichier descripteur **web.xml.** Le déploiement du composant servlet peut également s'effectuer en copiant l'arborescence du projet dans le répertoire **webapps** de Tomcat. Tout l'avantage des fonctionnalités intégrées de MyEclipse est de vous épargner cette peine.

6. Sélectionnez le support du niveau J2EE 1.3, comme illustré à la figure 6.15, en laissant inchangées les options proposées par défaut.

Figure 6.15
Support Web du projet Java



7. Cliquez sur Fin. La structure arborescente si caractéristique des projets Web apparaît dans la vue Packages (création des répertoires **WEB-INF** et **META-INF** puis du descripteur **web.xml** ainsi que d'un certain nombre de bibliothèques de support aux composants Web, notamment servlets).

8. Copiez les fichiers JAR nécessaires au fonctionnement du framework Cactus dans le répertoire **WEB-INF/lib** du projet (les JAR requis à l'étape 4).

Création d'une servlet dans Cactus

Vous allez à présent créer une servlet de test que vous testerez ensuite à l'aide du framework Cactus :

1. Créez une classe `servlet` en pressant Ctrl+N puis en sélectionnant l'assistant de création des servlets *(voir figure 6.16)*.

2. Cliquez sur Suivant.

3. Dans la fenêtre de l'assistant illustrée à la figure 6.17, saisissez **TestServlet** dans le champ Nom.

4. Désactivez l'option Create doPost, et ne gardez que Create doGet. Laissez inchangées les autres options proposées par défaut.

5. Précisez le nom du package destiné à contenir l'objet `servlet`.

6. Cliquez sur Suivant. La page suivante de l'assistant vous propose un ensemble d'options décrivant la servlet (champs Display Name et Description) ainsi que le mapping de la servlet avec l'URL associée.

Figure 6.16
Sélection de l'assistant de création de servlets

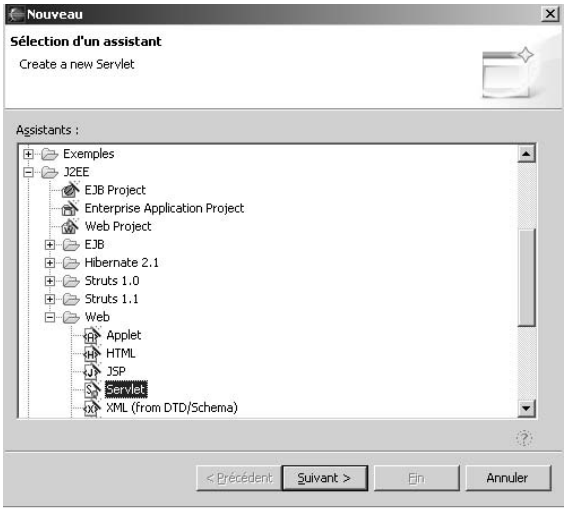
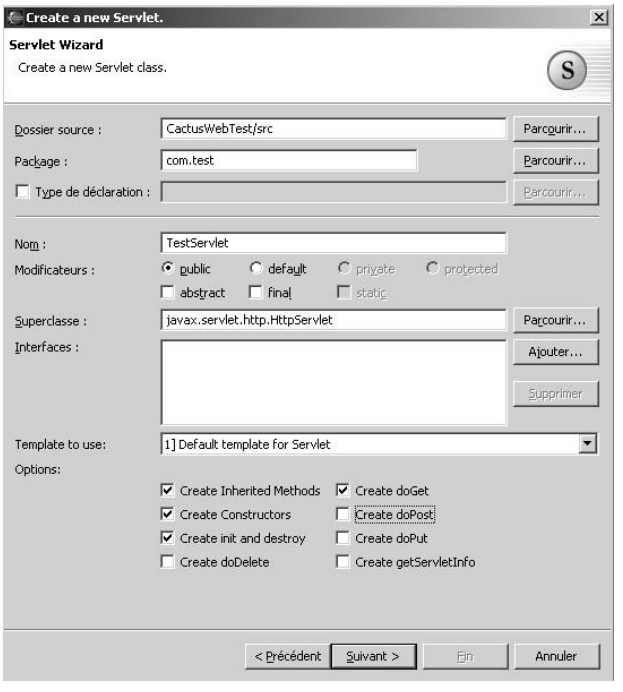


Figure 6.17
Création d'un objet servlet avec l'assistant MyEclipse



7. Cliquez sur Fin.
8. La servlet `MaServlet` est créée dans Eclipse, et l'éditeur Java s'ouvre sur le fichier **TestServlet.java.**

9. Remplacez le contenu de la méthode généré par défaut par le contenu suivant :

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println ("Message de la servlet TestServlet!");
            out.flush();
            out.close();
}
```

10. Le ficher descripteur **web.xml** est mis à jour automatiquement en fonction des informations de la servlet :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
➥"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>MaServlet</servlet-name>
    <display-name>Nom affiche du composant servlet</display-name>
    <description>Description du composant servlet</description>
    <servlet-class>com.test.cactus.MaServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>MaServlet</servlet-name>
    <url-pattern>/servlet/MaServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Déploiement de la servlet

Vous pouvez à présent passer à l'étape de déploiement de votre servlet `TestServlet`, sachant que cette étape est entièrement prise en charge par MyEclipse Workbench.

Nous supposons préalablement installée la version Apache Tomcat 5.*x* sur votre poste et configurée dans l'IDE Eclipse *via* l'option Préférences. Nous supposons également la création d'une configuration spécifique pour un déploiement sur ce serveur. Pour plus de détails sur la configuration du déploiement voir la partie III de l'ouvrage, consacrée au développement Web avec MyEclipse.

1. Lancez votre serveur en cliquant sur l'icône ci-contre puis en sélectionnant Tomcat 5 puis Start. Vous pouvez alors suivre la progression du serveur dans la vue console.
2. Saisissez l'URL *http://localhost:8080/CactusWebTest/servlet/MaServlet* dans votre navigateur. Vous devez voir s'afficher le message illustré à la figure 6.18.

À ce stade, vous avez une servlet fonctionnelle. Vous allez maintenant la tester.

Figure 6.18
La servlet TestServlet *en action*

Ajout d'un filtre Cactus et configuration du fichier descripteur *web.xml*

Vous allez commencer par configurer le projet contenant les bibliothèques Cactus nécessaires au test in-container.

Ajoutez le code suivant (en gras) dans le descripteur de déploiement **web.xml** afin de fournir des informations de filtrage spécifiques au framework cactus :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
➥"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <filter>
     <filter-name>FilterRedirector</filter-name>
     <filter-class>org.apache.cactus.server.FilterTestRedirector</filter-class>
  </filter>
  <filter-mapping>
     <filter-name>FilterRedirector</filter-name>
     <url-pattern>/FilterRedirector</url-pattern>
  </filter-mapping>
  <servlet>
    <servlet-name>MaServlet</servlet-name>
    <display-name>Nom de la servlet affichee </display-name>
    <description>Description de ma servlet</description>
    <servlet-class>com.test.cactus.MaServlet</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>ServletRedirector</servlet-name>
    <servlet-class>
org.apache.cactus.server.ServletTestRedirector</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Servletredirector</servlet-name>
    <url-pattern>/TestServlet/ServletRedirector</url-pattern>
  </servlet-mapping>
```

```
  <servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/servlet/TestServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

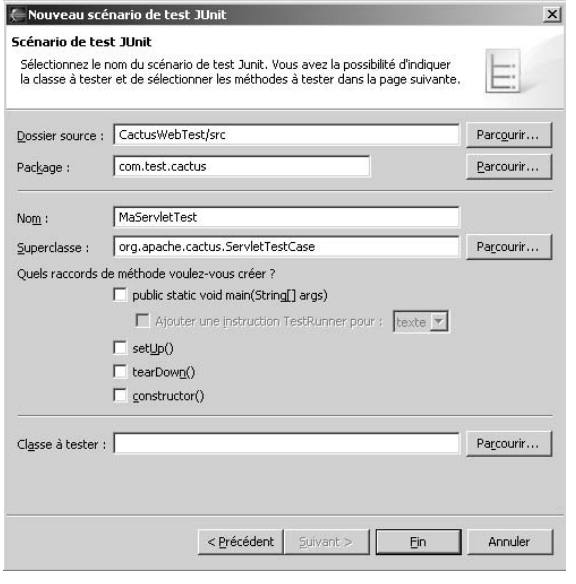
Le seul élément notable de ce code est la manière dont la servlet `ServletRedirector` est utilisée. Dans la mesure où la servlet `MaServlet` est mappée sur le nom `servlet/MaServlet`, le composant `ServletRedirector` doit être défini relativement à la servlet à tester, ce qui explique l'utilisation de cette URL dans la balise `<url-pattern>`.

Création et configuration de la classe de test Cactus

Vous allez à présent étendre la classe `ServletTestCase` pour écrire les tests utilisant les API servlet :

1. Pressez Ctrl+N pour ouvrir la boîte de dialogue de création de projet Eclipse, et sélectionnez Java, JUnit et Scénario de test JUnit.
2. Cliquez sur Suivant. La boîte de dialogue définissant le scénario de test JUnit s'affiche, comme illustré à la figure 6.19.
3. Saisissez **MaServletTest** comme nom de servlet, et changez le nom de la superclasse en **org.apache.cactus.ServletTestCase.**
4. Vous n'utiliserez pas ici les méthodes `setUp()` et `tearDown()`, qui seront substituées respectivement dans le framework Cactus par `begin<nom>` de la méthode `methode_a _tester()` et `end<nom>` de la méthode `methode_a _tester()`. Dans votre cas de test Cactus il s'agira de la méthode `doGet()` associée aux méthodes `endDoGet()` et `testDoGet()`.

Figure 6.19
Configuration du scénario de test Cactus



5. Cliquez sur Fin pour fermer la boîte de dialogue.

6. Reprenez le contenu du squelette de test Cactus, et saisissez le code suivant (en gras) :

```
/*
 * Créé le 5 janv. 2005
 *
 * TODO Pour changer le modèle de ce fichier généré, allez à :
 * Fenêtre - Préférences - Java - Style de code - Modèles de code
 */
package com.test.cactus;

import java.io.IOException;

import javax.servlet.ServletException;

import org.apache.cactus.ServletTestCase;
import org.apache.cactus.WebResponse;

/**
 * @author djafaka
 *
 * TODO Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre - Préférences - Java - Style de code - Modèles de code
 */
public class TestServlet extends ServletTestCase {

    public void testDoGet() {
        MaServlet servlet = new MaServlet();

        try {
            servlet.doGet(request,response);
        }
        catch (ServletException e){
            e.printStackTrace();
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
    public void endDoGet(WebResponse response){
        String texte=response.getText();
        String texteAttendu="Message de MaServlet?";
        assertNotNull(texte);
        assertEquals(texteAttendu,texte);
    }
}
```

Seules les méthodes `testDoGet()` et `endDoGet()` sont utilisées ici, dans la mesure où vous souhaitez vérifier le texte de sortie de la servlet `MaServlet` dans la méthode `endDoGet()` en le comparant à `texteAttendu`. Remarquez l'utilisation de l'objet particulier `WebResponse`, qui représente la réponse HTTP de la servlet `MaServlet`.

La méthode `testDoGet()` se contente d'instancier la servlet `MaServlet` et d'invoquer la méthode à tester à l'aide des objets prédéfinis `request` et `response` en prenant bien soin d'intercepter toute exception fatale. Le résultat de la réponse stocké dans l'objet `response` est ensuite analysé par la méthode `endDoGet()` pour comparaison avec le texte attendu.

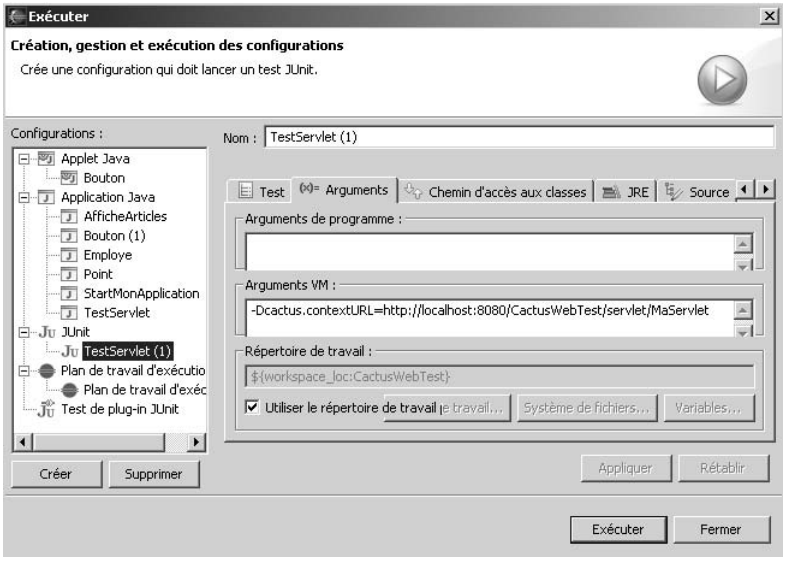
Le déploiement de ce cas de test Cactus dans Tomcat ne présente aucune difficulté, si ce n'est le positionnement dans la JVM d'une variable d'environnement représentant l'URL à tester :

1. Faites un clic droit sur la classe `MaServletTest` puis, à partir du menu contextuel, sélectionnez Exécuter et à nouveau Exécuter.

2. Dans la zone Arguments VM de l'onglet Arguments, saisissez la ligne suivante :

```
Dcactus.contextURL=http://localhost:8080/CactusWebTest/servlet/MaServlet
```

Figure 6.20
Configuration de l'URL pour le test de la servlet avec Cactus



3. Cliquez sur Appliquer puis Fermer.

Fichier *cactus.properties*

Vous pouvez également positionner cette valeur d'URL en passant par un fichier **cactus.properties** contenant l'URL de contexte du projet Web ainsi que le nom de la servlet de redirection :

```
cactus.contextURL = http://localhost:8080/CactusWebTest/servlet/MaServlet
cactus.servletRedirectorName = ServletRedirector
```

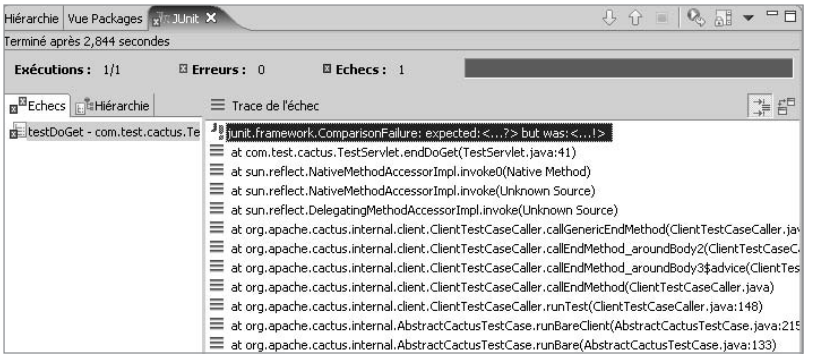
La référence à ce fichier s'effectuera *via* les variables de CLASSPATH habituelles (accessibles par le biais des propriétés du projet).

Lancement du test Cactus

Vous voici prêt pour exécuter votre test Cactus en utilisant JUnit.

1. Relancez votre serveur Tomcat par le biais de l'icône ci-contre s'il est arrêté.
2. Vérifiez que votre servlet `MaServlet` est bien opérationnelle en saisissant l'URL *http://localhost:8080/CactusWebTest/servlet/MaServlet* dans votre navigateur.
3. Faites un clic droit sur votre classe de test `MaServletTest` dans la vue Explorateur, puis sélectionnez Exécuter et Test JUnit à partir du menu contextuel. Le lanceur Eclipse utilisera la configuration de lancement précédemment configurée en utilisant la seule classe `MaServletTest`.
4. L'exécution génère une vue JUnit *via* la GUI `TestRunner` de JUnit, qui affiche une barre rouge *(en noir sur la figure 6.21)* montrant l'échec (intentionnel) du cas de test.

Figure 6.21
Affichage de la barre rouge d'échec via *la GUI JUnit*



5. Nous avons en effet volontairement changé la chaîne de texte attendue. Pour vous en rendre compte, double-cliquez sur la ligne incriminée, `junit.framework.ComparisonFailure`, dans la zone Trace de l'échec. La figure 6.22 illustre la comparaison des résultats.
6. Revenez sur votre code, et modifiez le texte attendu en **Message de MaServlet !**
7. Cette fois le code est normalement concluant, et vous devez voir s'afficher la barre verte du succès *(en gris sur la figure 6.23).*

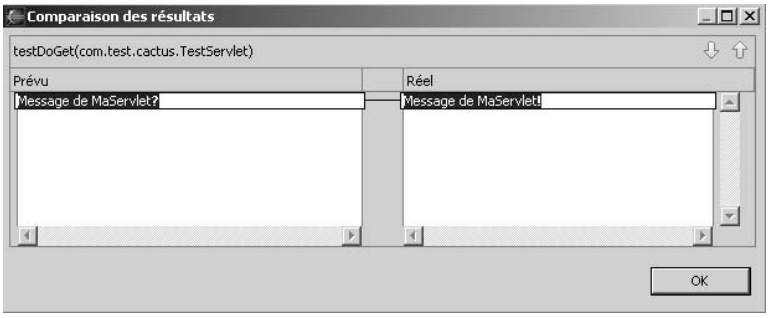


Figure 6.22
Fenêtre de comparaison des résultats

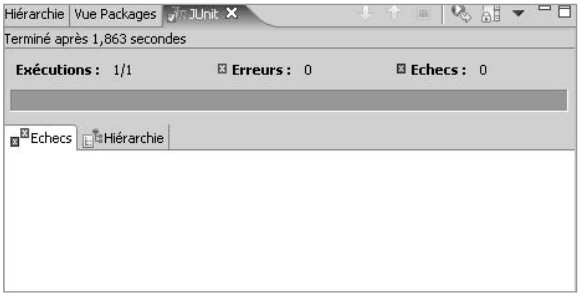


Figure 6.23
Le lanceur JUnit affichant la barre de succès du test

En résumé, les étapes clés effectuées par le framework Cactus pour le test de votre servlet en fonction des différents paramétrages effectués jusqu'ici sont les suivantes :

1. Le code client `testServlet` exécute et effectue un certain nombre d'initialisations JUnit.
2. Le framework Cactus vérifie la présence éventuelle d'une méthode `beginDoGet()` et l'exécute avant toute autre méthode (cette étape n'est pas effectuée ici, vu l'absence de cette méthode).
3. Le framework Cactus connecte le code client JUnit à l'objet serveur *via* le proxy d'indirection représenté par la classe Cactus `ServletRedirector`, qui est déployée avec la servlet `MaServlet` par le biais du descripteur **web.xml.**
4. L'objet `ServletRedirector` instancie la classe `TestServlet` côté serveur sous le contrôle du conteneur Tomcat et appelle la méthode `testDoGet()`, qui crée une instance de `MaServlet` *(voir code).*
5. Lorsque le code de la méthode `testDoGet()` se termine, l'objet `ServletRedirector` retourne à la partie client et appelle la méthode `endDoGet()`, laquelle reçoit l'objet `WebResponse` contenant le texte de sortie de la servlet `MaServlet`, concluant le test.

En résumé

Ce chapitre vous a permis de mesurer l'excellente complémentarité des frameworks JUnit et Cactus dans la mise en œuvre d'une stratégie de test de bout en bout dans un seul et même environnement, Eclipse.

Le chapitre suivant conclut cette partie consacrée à la mise en route d'Eclipse et de JBoss en présentant le serveur d'applications JBoss, l'autre thème central de cet ouvrage.