JUnit 4 Tutorial

Anthony Anjorin, 2006

Technische Universität Darmstadt · FB Informatik · FG Metamodellierung

1 Einführung

Dieses Tutorial gibt einen kurzen Einblick in JUnit 4. Als weiteres Einführungsdokument können wir JUnit 4.0 in 10 minutes empfehlen.
Um dieses Tutorial durcharbeiten zu können, sollten Sie mit der Bedienung von Eclipse und der Programmierung in Java vertraut sein. Tiefergehendes Wissen über Qualitätssicherung ist nicht erforderlich.

1.1 Was ist JUnit?

JUnit ist ein Framework das einem Entwickler ermöglicht, relativ schnell Unittests zu schreiben und zu jedem Zeitpunkt eine Sammlung von Tests—eine sogenannte *Testsuite*—automatisch ablaufen zu lassen.

JUnit bietet unter anderem verschiedene Methoden (wie z.B., `assertTrue()`, `assertFalse()`, `assertEquals()`) , um leicht Bedingungen im Code nachzuprüfen und viele Annotationen (Annotationen erfordern allerdings Java 5) womit man Methoden als *Test*, *Setup*, *Teardown* oder *Ignored* auszeichnen kann.

Um eine Klasse zu testen, werden in der Regel folgende Methoden erstellt:

- eine *Setup*-Methode, die alle nötigen Objekte anlegt und auf den für den Test nötigen Ausgangszustand bringt.
- eine Reihe von *Tests*, die die eigentlichen Tests beinhalten.
- eine *Teardown*-Methode, um aufzuräumen.

Mittels einer GUI ist es dann möglich die Tests automatisch ablaufen zu lassen, um sich die Ergebnisse und Ursachen im Falle eines Fehlschlags anzeigen zu lassen. Die Wirkungsweise von JUnit 4 wird im Abschnitt 3 anhand von Beispielen näher erklärt.

1.2 Wieso sollte ich JUnit verwenden?

Test Driven Development: TDD ist eine Entwurfsmethodik welche sich zunehmend verbreitet und zurecht empfohlen wird. Sie gibt vor, dass man *zuerst* Tests schreibt, und *danach* den Code implementiert, der diese Tests bestehen muss. TDD ist einer der Kernbestandteile von XP (eXtreme Programming) und wurde mittlerweile von den meisten agilen Vorgehensweisen aufgegriffen. JUnit eignet sich hervorragend für eine Umsetzung von TDD und unterstützt den Entwickler so dabei, sich über Gestalt und Zweck einer Methode oder Klasse im Klaren zu werden. Dabei sind *Ignore*-Annotationen sehr hilfreich, denn sie ermöglichen es, einzelne Tests kurzfristig auszuschalten, solange man noch an der dazugehörigen Implementierung schreibt.

Regressiontesting: Eine Regressionstestsuite zu schreiben, ist ohne den Einsatz von Tools wie JUnit nicht denkbar. Regressionstests werden nach jeder Änderung ausgeführt und gewährleisten so, dass nach Änderungen am Programm keine ungewollten Seiteneffekte die Korrektheit der alten Funktionalität gefährden. Für grössere Projekte ist es aus Zeitgründen einfach nicht möglich eine Testsuite immer wieder von Hand auszuführen.

2 Installation und Verwendung mit Eclipse

Eclipse 3.2 enthält bereits JUnit 4.1. Sie müssen lediglich ein neues Projekt anlegen, auf `Project` rechts-klicken und `New` → `JUnit Test Case` auswählen. In dem dann erscheinenden Dialog kann man `New JUnit 4 test` auswählen. Wenn unten im Fenster eine Warnung angezeigt wird, dass JUnit 4 sich nicht im *build path* befände, einfach auf `Click here` klicken und `OK` auswählen um das Problem zu beheben. Danach den Klassennamen als `JUnitHelloWorld` angeben und den folgenden Code eingeben:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class JUnitHelloWorldEclipse3_2 {
  @Test
  public void testHelloWorld() {
    String s = "HelloWorld";
    assertEquals("Just a test to see if everything works ...",
        "HelloWorld", s);
  }
}
```

Jetzt auf die Klasse im `Package Explorer` rechts-klicken, und `Run As` → `JUnit Test` auswählen, um den Test ablaufen zu lassen.

Falls Sie eine ältere Version von Eclipse benutzen wollen, lesen Sie bitte die folgenden Abschnitte. Ansonsten können Sie mit Abschnitt 3 weitermachen.

2.1 JUnit 4 runterladen

Unter www.junit.org auf Download gehen und JUnit 4 auswählen. Nach dem Entpacken die Datei `junit-4.1.jar` in einem Verzeichnis Ihrer Wahl speichern.

2.2 In Eclipse integrieren

Ein neues Java-Projekt erstellen oder ein schon Bestehendes öffnen bzw. auswählen. Auf `Project` → `Properties` klicken, `Java Build Path, Libraries` auswählen, und auf `Add External JARs` klicken. Jetzt zu `junit-4.1-jar` navigieren und auswählen. `Okay` drücken.

Eine neue Klasse `JUnitHelloWorld` erstellen und folgenden Code eingeben:

```
import org.junit.Test;
import static org.junit.Assert.*;
import junit.framework.JUnit4TestAdapter;

public class JUnitHelloWorldEclipseOld {
  @Test
  public void testHelloWorld() {
    String s = "HelloWorld";
    assertEquals("Just a test to see if everything works...",
        "HelloWorld", s);
  }
```

```
  public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(JUnitHelloWorldEclipseOld.class);
  }
}
```

Jetzt auf die Klasse im `Package Explorer` rechts-klicken, und `Run As` → `JUnit Test` auswählen, um den Test ablaufen zu lassen.

Der `JUnit4TestAdapter`, der die `suite()`-Methode zurückgibt, wird benötigt, um die neuen JUnit4-tests mit den alten JUnit-runners ablaufen zu lassen.

3 Beispieltestsuite für ein TicTacToe-Spiel

Anhand der folgenden Beispieltests für ein einfaches TicTacToe-Spiel werden die wichtigsten Features von JUnit erklärt. Um die Beispiele auf älteren Versionen von Eclipse ablaufen zu lassen, müssen Sie den Import und die `suite()`-Methode für den JUnit4TestAdapter ergänzen (siehe Abschnitt 2.2).

In der Qualitätsicherung unterscheidet man zwischen *whitebox-tests* und *blackbox-tests*. JUnit eignet sich für beide Testarten und in den folgenden Beispielen, sehen wir uns einen White-Box-Test für das Model, und einen Black-Box-Test für den View an.

Das Archiv mit dem Quellcode und der Testsuite steht zum Download bereit. Zusätzlich bieten wir eine PDF-Version des Tutorials an, in der die Testsuite mit abgedruckt ist.

3.1 White-Box-Test für das Model

Die folgenden Features von JUnit 4 werden verwendet und erklärt:

@BeforeClass und @AfterClass: Methoden, die vor und nach allen Tests ausgeführt werden.

@Before und @After: Methoden, die vor und nach jedem Test ausgeführt werden.

@Test: Die tatsächlichen Testmethoden.

Assertmethoden: Mittels der von JUnit zur Verfügung gestellten Methoden wie assertTrue, assertEquals u.s.w. können Bedingungen im Code leicht überprüft werden.

Erwartete Exceptions: In der Test-Annotation ist es möglich eine Exception anzugeben, die der Test werfen sollte.

@Ignore: Methoden, die noch nicht bestehende Funktionalität testen, können temporär ignoriert werden.

Hilfsmethoden: Es ist auch möglich beliebig viele normalen Methoden in der Klasse zu benutzen.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;

import ticTacToeMVC.TicTacToeModel;
import static observerPattern.Observable.Player.*;
import observerPattern.Observable.Player;
```

```
/**
 * We shall strive to test the public interface of the model, ignoring
 * setters and getters. The tests are arranged so that we start with
 * functionality used by other methods. i.e. makeMove() is used by
 * computerMove() etc.
 */
public class WhiteBoxTestForModel {

  private static TicTacToeModel model;

  private static int rows;

  private static int columns;

  /**
   * Methods with the annotation 'BeforeClass' are executed once before
   * the first of the series of tests. External resources that are used
   * by all tests should be initialised here.
   */
  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    model = new TicTacToeModel();
    rows = model.getROWS();
    columns = model.getCOLS();
  }

  /**
   * Methods with the annotation 'AfterClass' are executed once after the
   * last test has been run. Resources used by the tests that need to be
   * released such as streams etc. should be set free here. In our
   * example there is nothing to be done.
   */
  @AfterClass
  public static void tearDownAfterClass() throws Exception {
  }

  /**
   * Methods with the annotation 'Before' are executed before every test.
   * The test object should be brought to the initial state all tests
   * assume it to be in.
   */
  @Before
  public void setUp() throws Exception {
    model.reset();
  }

  /**
   * Methods with the annotation 'tearDown' are executed after every
   * test.
   */
  @After
  public void tearDown() throws Exception {
  }

  /**
   * Methods with the annotation 'Test' are the actual test methods.
   */
  @Test
```

```
public void reset() throws Exception {
  // Are all fields correctly initialised?
  Player[][] expected = {
      { NOBODY, NOBODY, NOBODY },
      { NOBODY, NOBODY, NOBODY },
      { NOBODY, NOBODY, NOBODY } };

  checkStateOfModel(
      expected,
      "After a reset, the playing fields should be initialised and
         belong to nobody.");

  assertTrue("After a reset, by convention it should be X's turn.",
      model.getCurrentPlayer() == Player.X);
}

/**
 * Via the static import of org.junit.Assert.* a series of assert
    methods
 * can be used. In the following tests we use assertTrue, assertFalse
 * and assertEquals.
 *
 * @throws Exception
 */
@Test
public void nextPlayer() throws Exception {
  // After a reset it is X's turn. Hence after calling nextPlayer it
  // should be O's turn
  model.nextPlayer();
  assertTrue("next player should be O",
      model.getCurrentPlayer() == Player.O);
  model.nextPlayer();
  assertTrue("next player should be X",
      model.getCurrentPlayer() == Player.X);
}

/**
 * The Test Annotation can be extended by stating what exception should
 * be thrown when the test is executed, the test fails if no exception
 * or if the wrong exception is thrown.
 */
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void makeMove() throws Exception {
  assertTrue("Since field is free, a move should be possible", model
      .makeMove(0, 0));
  assertTrue("Since field is free, a move should be possible", model
      .makeMove(rows - 1, columns - 1));
  assertTrue("Since field is free, a move should be possible", model
      .makeMove(0, columns - 1));
  assertTrue("Since field is free, a move should be possible", model
      .makeMove(rows - 1, 0));

  Player[][] expected = {
      { X   , NOBODY, X   },
      { NOBODY, NOBODY, NOBODY},
      { X   , NOBODY, X   } };

  checkStateOfModel(expected, "Model is in an unexpected state.");
```

```
    assertFalse(
        "Since the field is already occupied, a move should be
            impossible",
        model.makeMove(0, 0));

    // Should cause an exception to be thrown since (rows, columns) is
    // never a valid field.
    model.makeMove(rows, columns);
  }

  /**
   * One should always try to write robust unit tests that won't be
   * broken by every minute change to the class under test. Using a loop
   * here from 0 to rows*columns instead of nine function calls is an
   * example.
   *
   * @throws Exception
   */
  @Test
  public void computerMove() throws Exception {
    for (int i = 0; i < rows * columns; i++) {
      assertTrue(
          "Expected computer to find a free field and make a move.",
          model.computerMove());
    }

    assertFalse("Since board is full, expect false as return value.",
        model.computerMove());

    assertTrue("Model should be full", model.isFull());

    Player[][] expected = {
        { X, X, X },
        { X, X, X },
        { X, X, X } };

    checkStateOfModel(expected, "Model is in an unexpected state.");
  }

  /**
   * The assert methods take as an extra argument a message that is
   * displayed when the test fails.
   *
   * @throws Exception
   */
  @Test
  public void isFull() throws Exception {
    assertFalse("Board should not be full", model.isFull());

    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
        model.makeMove(i, j);
      }
    }

    assertTrue("Board should be full", model.isFull());
```

```
  }

  /**
   * To support TDD, hence writing tests before implementing, 'Ignore'
   * tags can be used to exclude tests for functionality that hasn't been
   * implemented yet. To demonstrate this the following test has been
   * 'ignored'. Run the test and notice how JUnit reports which tests
   * have been ignored. Remove the 'Ignored' tag to run the test
   * normally.
   *
   * @throws Exception
   */
  @Ignore
  @Test
  public void isWin() throws Exception {
    assertFalse("Empty board - no win", model.isWin());

    for (int i = 0; i < rows; i++) {
      model.makeMove(i, i);
    }

    assertTrue("Three diagonal \\ should be a win.", model.isWin());
    model.reset();

    for (int i = 0; i < rows; i++) {
      model.makeMove(i, 0);
    }

    assertTrue("Three down should be a win.", model.isWin());
    model.reset();

    for (int i = 0; i < columns; i++) {
      model.makeMove(0, i);
    }

    assertTrue("Three across should be a win.", model.isWin());
  }

  /**
   * All auxiliary methods without tags used by the test methods or
   * otherwise, are treated as normal methods.
   *
   * @param expected
   *            The state the model is expected to be in.
   * @param message
   *            The message to be displayed when assert fails.
   */
  private void checkStateOfModel(Player[][] expected, String message) {
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
        assertEquals(message, expected[i][j], model
            .getOwnerOfFigureOnPlayingField(i, j));
      }
    }
  }

}
```

3.2 Black-Box-Test für den View

In dem Beispiel wird ein Klick simuliert und in diesem Zusammenhang einiges zur Testbarkeit einer Klasse erläutert.

```
import org.junit.BeforeClass;
import org.junit.Test;

import ticTacToeMVC.TicTacToeController;
import ticTacToeMVC.TicTacToeModel;
import ticTacToeMVC.TicTacToeView;
import static observerPattern.Observable.Player.*;
import static org.junit.Assert.*;

/**
 * The View is used to convey a move by clicking on a field. We shall
 * simulate this, and test if the model has been changed appropriately. We
 * assume the controller has been tested and functions properly. We shall
 * write a black-box test in the sense that we write tests without
 * knowledge of the inner structure of the view, i.e. how many different
 * methods there are etc.
 */
public class BlackBoxTestForView {

  private static TicTacToeModel model;

  private static TicTacToeView view;

  private static TicTacToeController controller;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    model = new TicTacToeModel();
    controller = new TicTacToeController(model);
    view = new TicTacToeView(model, controller);
  }

  /**
   * One thing we can test is if a click is relayed properly to the
   * model. The Dialogs that should pop up for a win, tie etc. would have
   * to be tested by hand. At times automated testing is limited if one
   * doesn't want to invest to great an effort. It also depends a great
   * deal on if the class under test is 'testable' i.e. without the
   * getBtn() method in the view, we would have a problem. There are
   * actually special tools, specifically for GUI-testing which might be
   * more appropriate under certain conditions.
   *
   * @throws Exception
   */
  @Test
  public void simulateClick() throws Exception {
    int testRow = 0, testColumn = 0;

    view.getBtn(testRow, testColumn).getActionListeners()[0]
        .actionPerformed(null);

    assertTrue("Expected currentPlayer X on field(" + testRow + ","
        + testColumn + ")", model.getPlayingField(testRow,
        testColumn).isPlayer(X));
```

```
        view.getBtn(testRow, testColumn).getActionListeners()[0]
            .actionPerformed(null);

        assertTrue("Expected currentPlayer X on field(" + testRow + ","
            + testColumn + ")", model.getPlayingField(testRow,
            testColumn).isPlayer(X));

        view.getBtn(testRow, testColumn + 1).getActionListeners()[0]
            .actionPerformed(null);

        assertTrue("Expected currentPlayer O on field(" + testRow + ","
            + (testColumn + 1) + ")", model.getPlayingField(0, 1)
            .isPlayer(O));
    }
}
```