Stubs and mocks

Validation and Verification
University of Rennes 1

Erwan Bousse (erwan bousse@irisa fr)

Last update October 3, 2013

You will first have to analyze a simple project using CodePro Analytix, then you will use the Mockito framework to test its few classes. The main objective is to understand the difference between stubs and mocks, and to be able to decide when to use them.

1 Background

1.1 Stubs and mocks

We will focus here on two very similar concepts, both handled (in our case) by the Mockito framework: *stubs* and *mocks*. Both are constructed in the same way using Mockito, and both represent instances of "copies" of existing classes. However, we call *stubs* the ones used for unit testing, in order to be able to isolate a class even when there are dependency cycles. And we call *mocks* the ones used for integration testing, in which the behavior of a class is verified by checking that it interacts with other classes as it was specified (e.g. using UML sequence diagrams).

1.2 The Mockito framework

The Mockito[1] framework is a very helpful way to generate stubs and mocks without rewriting (partial) implementations of the interfaces of the classes (which is also helpful when the code does not use interfaces at all – which is not recommended, of course). It uses Java annotations to reduce the amount of setup to make, and defines very intuitive methods to manipulate mocks and stubs. An example is given Figure 1.

There is nothing to do for the installation since Mockito should be in the dependencies of the Maven configuration file that was given to you.

[1] http //code google com/p/mockito/

```java
//It is recommended to import Mockito statically so that the code looks clearer
import static org.mockito.Mockito.*;
// Eclipse might not find this one automatically.
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class) // This is required for mocks to work
public class SomeTest{

  @Mock // This annotation declares a mock attribute
  private LinkedList<String> list1;

  /* This annotation tries to automatically create the annoted attribute
  by using previously declared mocks (in this case, list1)//
  Note that it doesn't work with complex constructors  */
  @InjectMocks
  private ClassUsingLinkedList linkedListUser1;

  private LinkedList<String> list2;
  private ClassUsingLinkedList linkedListUser2;

  @Before
  public void setUp() {
    list2 = mock(LinkedList.class); // Equivalent to @Mock
    linkedListUser2 = new ClassUsingLinkedList(list2); // Equivalent to @InjectMock
  }

  /*
   * In these tests we do a bit of unit testing of ClassUsingLinkedList by stubing
   * LinkedList (note that this is a little stupid, since we can trust the Java LinkedList class)
   */
  @Test
  public void stubing1() {
    when(list1.get(0)).thenReturn("first");
    assertEquals(linkedListUser1.getFirst(),"first");
  }
  @Test(expected=RuntimeException.class)
  public void stubing2() {
    when(list1.get(1)).thenThrow(new RuntimeException());
    linkedListUser1.getSecond();
  }

  /*
   * In this test we verify the interactions between
   * ClassUsingLinkedList and LinkedList
   */
  @Test
  public void mocking() {
    linkedListUser2.reinitWith("one");
    verify(list2).clear();
    verify(list2).add("one");
  }
}
```

Figure 1: Example with Mockito, for both stubing and mocking.

- The setting is a rectangular board, divided in $X \times Y$ squares, with one "bonus" square, and with N pawns distributed over the board
- Each pawn has a number, and turns are based on the order of the numbers (e.g. 0 plays first, then 1 plays, then 2 plays, etc.)
- Each turn, a pawn moves in a direction (up, down, left, right). There are two special cases:
  - It encounters a pawn. In that case, it attackes the pawn and hurts it
  - It tries to go out of the board, and loses its turn
- When attacking, a pawn normally inflicts 1 damage to another pawn. However, it inflicts twice the amount if it is on the "bonus" square. If a pawn kills another one, it earns 1 gold.
- A pawn has 5 hitpoints
- The game is over either when there is 1 pawn left, either when a pawn has 3 golds.

Figure 2: Rules of the game

1.3 Google CodePro Analytix

Google CodePro Analytix[2] is an Eclipse tool with many features to analyze Java source code. In this session, we will only use the Audit plugin, and more specifically the dependency analyzer. It can automatically draw dependency graphs, which are very useful to localize cycles.

Installation Add the update site(`https //developers google com/java-dev-tools/codepro/doc/installation/updatesite_3 5`), and install the Audit plugin.

Usage Right click on a project → Code Pro Tool → Analyze dependencies.

2 Testing multiple classes

2.1 The program to test: a tiny board game

You have to test a very little game which consists in moving pawns over a board. Each player has a pawn, and it can attack other players. The complete set of rules are available in Figure 2. The Java program consists in 6 classes distributed between 3 packages (see Figure 3).

[2] `https://developers.google.com/java-dev-tools/codepro/doc/`

simplegame.cli Package with the user interface and the main method.

CLIMain The main class, to play the game with a console interface.

simplegame.core Package with the main classes.

Board Class that describes the board.

Direction Enumeration of the directions a Pawn can follow.

Game Class to initialize and interact with the game.

Pawn Class that describes the state and behavior of a Pawn.

simplegame.exception Package with the exceptions used.

OutOfBoardException Sent when a Pawn tries to leave the board.

Figure 3: Organisation of the packages/classes or the project

2.2 Unit testing with stubs

In this first part, you will have to unit test each class of the project using stubs if necessary.

Question 1 *Use CodePro Analytix dependency analysis tool to visualize the dependency graph of the project. To be able to see all interesting classes in the same view, select both simpleGame.core and simpleGame.cli packages, do a right click, and do 'Explore → Contents'. What do you notice?*

Answer There is a dependency cycle between Pawn and Board. It means that a stub must be made (either Pawn or Board) in order to be able to unit test the project correctly.

Question 2 *Write a testing plan: in which order you would test the classes, which ones you plan to stub, etc.*

Answer A correct test plan must follow this:

1. Test Pawn/Board. Two alternatives:
   (a) Test Board by stubbing Pawn, then test Board with the real Pawn and test Pawn with the real Board
   (b) Test Pawn by stubbing Board, then test Pawn with the real Board and test Board with the real Pawn

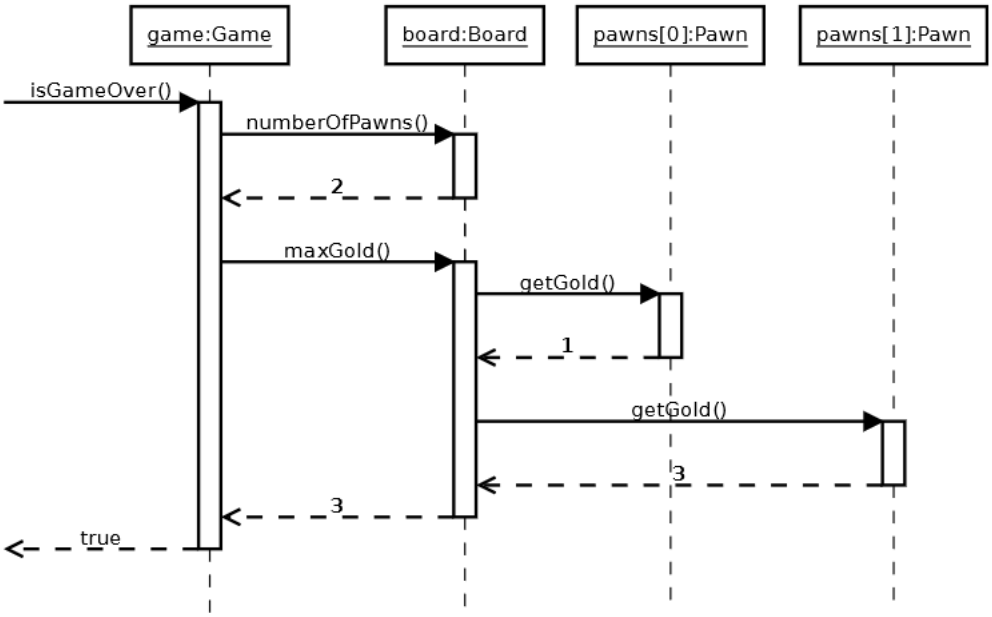

Figure 4: Sequence diagram representing an `isGameOver()` call scenario.

2. Test Game
3. Test CLIMain

Programming 1 *Apply your testing plan to unit test the Board and Pawn classes*

2.3 Integration testing with mocks

In this second part, you have to verify that the interactions in the code comply with the UML sequence diagram given in Figure 4 using mocks. Put all your tests for this part in a class called TestSequenceDiagram.

Question 3 *Which classes should be mocked? Why?*

Answer There are two behaviors to verify: isGameOver() and maxGold(). For the first one, we have to mock Board. For the second one, we have to mock two pawns.

Question 4 *For which reasons might you want to use Mockito methods* `when` *or* `thenReturns`*?*

Answer You don't want to use them: we want to verify interactions, not the functionnalities. You might have to use them only to allow the scenario to be possible: in our case, this is required to be able to use addPawn(Pawn) in order to give the mocks to Board (something like `when(mockPawn0 getX()) thenReturn(1)`; four times).

Programming 2 *Without modifying the project code, write a test for the* `numberOfPawns()` *call. Consider using the Mockito annotation* `@InjectMocks`.

Answer You can do this by simply declaring first a Board with @Mock, then a Game with @InjectMocks. See TestSeqDiag.java

Programming 3 *Write a test for the* `maxGold()` *call. Note that* `@InjectMocks` *cannot work here; you might have to add one method in* Board.

Answer @InjectMocks doesn't work because Mockito is not able to understand the constructor of Board in order to inject the Pawn mocks. Thus, you have to add some code to Board: either a removeAllPawns() method, to clean the board before adding pawns with addPawn(); or a new constructor that can receive pawns. See TestSeqDiag.java.

3 What to produce

You have to produce:

- An Eclipse Maven project in zip format (Export... → Archive). It must contain:
  - The source code of the tested/patched system
  - The source code of all your commented tests (which includes the class TestSequenceDiagram made in section 2.3)
  - The generated test report (with javadoc)
  - The generated test coverage report (with jacoco)
- A report in PDF format with the answers to all the questions.