Integration testing using mocks

Validation and Verification
University of Rennes 1

Simon Allier(simon.allier@inria.fr)
José Á. Galindo(jagalindo@inria.fr)

Last update September 21, 2015

The objective of this lab session is to understand what it means to test multiple inter-dependent classes. This goes from the realization of a *test plan* to the use of mocks for *stubbing* and *interaction testing*.

1 Background

1.1 Mocks?

We will focus here on two testing activities that are used for integration testing, both handled (in our case) by the Mockito framework: *stubbing* and *interaction testing*. Both are done using *mocks*, which represent instances of "copies" of existing classes. Stubbing consists in using mocks to isolate a class when there are dependency cycles. Interaction testing consists in checking that a class interacts with other classes as specified (e.g. in a UML sequence diagram).

1.2 The Mockito framework

The Mockito[1] framework is a very helpful way to generate 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 provides very intuitive operations 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.

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).

[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{

    @Test
    public void stubingSimple() {
        // Creating context
        PhonyList<String> mockList = mock(PhonyList.class);
        MyStructure struct = new MyStructure(mockList);
        when(mockList.get(0)).thenReturn("first");

        // Calling the tested operation
        String result = struct.getFirst();

        // Oracle
        assertEquals(result,"first");
    }

    @Test(expected=RuntimeException.class)
    public void stubingWithException() {
        // Creating context
        PhonyList<String> mockList = mock(PhonyList.class);
        MyStructure struct = new MyStructure(mockList);
        when(mockList.get(1)).thenThrow(new RuntimeException());

        // Calling the tested operation
        struct.getSecond();

        // The oracle is "expected=RuntimeException.class", so nothing to assert
    }

    @Test
    public void interactions() {
        // Creating context
        PhonyList<String> mockList = mock(PhonyList.class);
        MyStructure struct = new MyStructure(mockList);

        // Calling the tested operation
        struct.reinitWith("one");

        // Oracle: we check that the operation made some specific calls
        verify(mockList).clear();
        verify(mockList).add("one");
    }
}
```

Figure 1: Example of Mockito usage for stubbing and interactions testing.

- 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 attacks 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

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

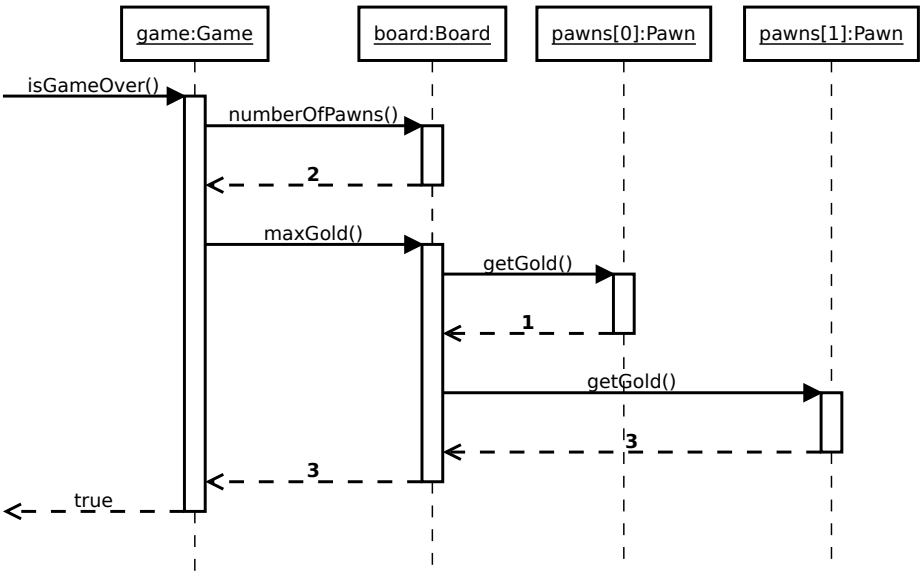

Figure 4: Sequence diagram representing an `isGameOver()` call scenario.

2.2 Stubbing

In this first part, you will have to plan how to test each class of the project using stubs if necessary. Then you will have to test two classes in particular.

Question 1 *Make a dependency graph of the classes of the project. You can use IntelliJ dependency analyzer to discover the dependencies of a class (right click → Analyze → Analyze Dependencies...).*

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

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

2.3 Interactions testing

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?*

Programming 2 *Write a test for the `isGameOver()` call.*

Programming 3 *Write a test for the `maxGold()` call.*

3 What to produce

You have to produce:

- A Maven project in `tar.gz` or `zip` format. 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.