Diary 2 – Mock Objects

Name: William Hutchinson, Sebastian Blessing
Username: wjh12, scb12
Class: Software Engineering for Industry – MSc Computing (Software Engineering), S5

Lecture Title: Mock Objects

Three key points from the lecture were:

- Test driven development can be used as a way to develop software by first producing failure-first tests to implement objects that pass the minimal requirements, and then by modifying such objects to extend their functionality (Red-Green-Refactor cycle). This method assures that objects are developed to be small and succinct, with more complex behaviour added afterwards. This also helps to assure the internal quality of software systems.
- Objects can be defined in terms of their roles, responsibilities and collaborators. Roles refer to an object's overall role within a system and are defined by its responsibilities. Responsibilities are tasks that other objects within the system can expect of a certain object. Collaborators define relationships and interactions between objects. However, objects here are defined at an abstract layer rather than focusing on their technical implementations.
- As unit tests are best suited to test "value-type" objects whereby the state of the object does not change, mock objects can be used to test entities, which have various dependencies and allow for the testing of messages going to and from a target object. This takes into account how well an object interoperates with other objects in the system.

Further Reading

Growing Object-Oriented Software, Guided by Tests (Pryce, 2010) describes how objects can employ the "*Law of Demeter"* which means that a target object should expect certain roles and responsibilities from a neighbouring object and should not replicate these within itself. This is also a good approach as it allows for a clear track of dependencies and improves internal quality. Such approaches should prevent issues such as train-wrecking which introduces code that is complex to read, as well as creating a train of dependencies spreading through the system. Although this approach is rigorous, in some legacy systems it may be necessary to keep train-wrecking in order to avoid vast refactoring of a system just for a particular case.

Gary Pollice describes within his IBM article (Pollice, 2006) how mock objects can be used and can aid in the efficiency of unit testing. He explains how such mock objects can be automatically created using modern tools. In the case where an object may use a train of dependencies, simply using a mock object to map the output and input allows for more precise conclusions to unit tests removing side effects from collaborating objects. It seems that although mock objects are useful within unit testing, creating them from scratch can be difficult and in which case automated tools should be relied on. However, the use of such tools introduces a new component to the system for test purposes which may compromise the rigour and aim of testing as it is the last opportunity that a developer has to determine how a system reacts.

Previous Experience

We are sure that many developers, ourselves included, have experienced train-wrecked code at one point or another. Having attempted to refactor or maintain such code in the past has led to far more time being spent on understanding such code. Therefore, avoiding such techniques by ensuring "Tell, Don't Ask" styles where possible seems sensible.

Although mock objects seem a practical technique for objects which have dependencies. However, using them for testing simplistic objects may become slightly overkill and perhaps waste development time. In scenarios such as this, techniques should be scrutinised as to their necessity; however, as with several of these techniques, although the introduction of them may require a loss of productivity at first, they assure more workable and robust code further within the development, which should ultimately save time further down the line.

The use of mock objects is a powerful technique as they allow how an object reacts given a particular scenario such as timed events or signals from external sources. This cuts down the amount of time that is necessary to set up a test environment for testing particular functionality of a specific object.

Tutorial Exercise

Within the tutorial exercise, the objective was to implement a central component of a tourist audio guide using test-driven-development methodologies. We start by writing a failing test for a given feature to be implemented. Afterwards an actual implementation satisfying its requirements is provided which passes the test, followed by any necessary refactoring of the code (red-

green-refactor cycle). To implement tests we used JUnit and JMock. As stated in the exercise specification there are five main features to be implemented:

1) On the first location received, play the corresponding audio track for this location.

In order to be notified about location changes, the AudioGuide class implements the LocationAware interface. In particular, the method *locationChanged(Location newLocation)* is called when a location change event occurs. At first we wrote a test that focuses on this behaviour:

```
@Test
public void playsMediaTrackForInitialLocation() {
    final Location initialLocation = new Location("South Kensington");
    final Track skTrack = context.mock(Track.class);

    context.checking(new Expectations() {{
        allowing(mediaLibrary).getTrackFor(initialLocation); will(returnValue(skTrack));
        oneOf(mediaPlayer).play(skTrack);
    }});

    guide.locationChanged(initialLocation);
}
```

By following the test-driven development approach we got a precise understanding of what is required to implement this feature. In this case, a method that returns the corresponding track for a given location is required (we provided this in the MediaLibrary interface) which is then used as an input parameter for *MediaPlayerControl::play(Track track)*.

2 and 3) No interruption of a track that is currently played. If the location is changed during playback, play the track of the new location after completion of the current track. Otherwise play immediately.

The important step for implementing this feature is that its tests can be split up into two different test cases. On the one hand we test that a track is not being interrupted and on the other hand we test that if the location has changed during playback, the track of the new location is automatically started when the current track is completed. For the first one to be executed successfully the AudioGuide class simply requires the ability to know that there is a track currently being played. To do so, we use a private boolean instance variable "*playbackOn_*" . Within the *locationChanged* handler we implement a check for this variable; only playing a track if evaluated to false. In order to pass the second test, the AudioGuide needs to be informed that a track is finished. This is achieved by implementing the *MediaPlayerListener* interface. The test itself checks for the invocation of *play(Track track)* on the given MediaPlayerControl mock object after the AudioGuide received a *trackFinished* message. In our case it turned out, that feature 3 has been implemented implicitly by feature 2. However, we verified this by implementing an additional test.

4) Do not play anything if there is no corresponding track for a given location.

As null references are only to be used within one class as sentinels and never to be passed within messages between objects, we are using the name property of a Track for identification purposes. If *MediaLibrary::getTrackFor(Location location)* does not find a corresponding track it returns a Track object with name "Unknown". Only Tracks with a name not equal to "Unknown" are than subject for playback. As we mock the Track implementation within tests, some refactoring on previous tests was required by allowing *Track::name()* to be called returning a name or "Unknown". In an ideal situation, the MediaLibrary would pass back an abstract class of track, which may be either a playable track or an unknown track object. However, in this exercise we have no access to the track implementation, as it is only available in pre-compiled format.

5) Play each track only once.

We implemented a test that requires *MediaPlayerControl::play(Track track)* only to be called for distinct tracks (i.e. returning to a previous location but expect *play()* only to be called once). To implement this we use an additional private instance variable "*playedTracks_"* of type Vector<Track>. A track is only played if not in this list, otherwise we append it and invoke *play()*.

These exercises emphasize the power of mock objects, particularly with regard to the dependencies on the MediaPlayer implementation. As mock objects allow for objects to be tested independently, the MediaPlayer implementation is not essential in order to complete other implementations. This becomes particularly important for work in distributed or large teams.

Pollice, G. (2006). *Using mock objects for complex unit tests*. Retrieved 10 2012, 15 from http://www.ibm.com/developerworks/rational/library/oct06/pollice/

Simmons, S. F. (2012). *Retrofitting Unit Tests*. Retrieved 10 2012, 10 from http://cf.agilealliance.org/articles/system/article/file/1032/file.pdf