Mocking and Test Driven Development

Marcus Mathioudakis Gajan Sivapackiarasa

The lecture on Mock Objects and Test Driven Development can be summarised by the following three key points:

1. The core TDD feedback loop: if it's hard to write a test for some code then it should be refactored.
2. The principle of designing a system in terms of relationships between objects (Role based design).
3. The concept of mocking and its role in test driven design.

In the lecture we started by discussing test driven development, i.e. the notion of writing tests for code which you have not yet written. We reviewed the benefits of tests driven development, such as the living documentation that the resulting classification tests provide, and the fact that it helps you locate and fix errors while the code is still fresh in your mind, thus saving you time and effort. We had been introduced to these benefits of TDD before both in our software engineering course last year and in our industrial placements, and were thus familiar with idea of having a test driven development (implementation) strategy. However the focus of the lecture was on something new to us, namely the idea of using TDD not only as a development strategy, but as a *design strategy*.

The core of this idea is the TDD feedback loop: if it's hard to write tests for a certain component in the system, then that is a poorly designed component. Specifically, the difficulty in testing a particular component is most probably due to design issues related to high coupling, i.e. too many dependencies on other components, and low cohesion, i.e. the component under test performs many unrelated functions. Thus the test driven approach gives us feedback on the quality of our design, and can help us identify design issues while they are still relatively easy to fix.

This brings us to the second key point, namely that of designing a system in terms of relationships between objects, instead of relationships between classes. The basic idea here was that a system should be defined entirely in terms of the roles of various objects, the responsibilities assigned to each role, and the collaboration between the various roles. Test driven development encourages this, as we start by defining interactions between objects in the tests, and thinking about roles and responsibilities, before actually writing any of the code under test. In the context of this role based design we were introduced to the principle of "Tell don't ask". The basic idea is that objects should tell their neighbours what to do, instead of asking for information they need to do something, thus increasing encapsulation and reducing coupling. However, if we follow this principle, then we are left with a bunch of objects that hide their state completely, only exposing the API corresponding to their roles. How can we do some kind of sensing (assertions about object state) in our tests if we can't access the object's state?

This brings us to the final key point of the lecture: Mocking. Mocking allows us to essentially create mock instances of the various other components that the object under test interacts with, and then make assertions about the way that we expect the object under test to communicate with the other objects, in terms of assertions about the methods it calls on them. The notion of making assertions about the interactions between objects instead of the state of the objects themselves encourages the role based approach previously mentioned, as it encourages us to think in terms of communication between objects, before thinking about the objects themselves. Finally, mocking frees the developer from having to write any seam objects, as they can just mock them and define expectations on the mocked objects.

The points discussed above were really made clear when we tackled the exercise on mocking.

In the exercise, we were given some classes modelling a tourist audio guide. The central components were an AudioGuide class, which implemented a LocationAware interface in order to know when the location of the user had changed, a MediaLibrary interface, and finally a

MediaPlayerControl which could be used to play tracks. We were also given an outline of the required behaviour of the AudioGuide, which essentially defined it's role and responsibilities, but without specifying how these would be achieved in terms of collaborations with other objects. The API for the MediaPlayerControl had also already been defined, and thus so had its role. The remaining components were however largely undefined, and thus it was left to us to assign roles and responsibilities, and define the collaborations between objects in order to come up with a design for the system.
Prompted by the lecture outlined above, we followed a test-driven approach in coming up with our role based design. We thus started by setting up a test class for the AudioGuide component. The plan was to write a test for each of the responsibilities outlined in the specification, and in trying to write each test we would gradually assign responsibilities to other components, and define how the AudioGuide would interact with them. We started by looking at the tests for the LocationTracker class, in order to familiarise ourselves with the JMock API and the way we would lay out our tests. The standard approach was to start by mocking all the objects that a given class relied on, and then in each test method define a set of expectations, followed by some method calls which should satisfy those expectations, followed by some more expectations, and so on.
However, once we started writing the first tests for the Audio guide, the problem proved to be harder than anticipated. As we had not yet written any of the code we were testing, it was quite hard to decide how to allocate responsibilities. A good example was the responsibility of playing a track for a new location when a track finishes, if the user has changed location. Who would store the location? The LocationTracker or the AudioGuide? Who should check if the location had changed? Should we have a method exposing the location of the AudioGuide? At this point it was very tempting to go and write the code for each of these classes before defining the expectations, as it is much easier to reason about code which you have already written. The problem is that in doing this the developer starts to think more in terms of classes, than in terms of how objects should communicate with each other. Thus writing a whole test first, and then writing the code which adds the responsibilities specified in the test, ensures that the design is as object-oriented as possible, in that it focuses on the communications and interactions between objects instead of the objects themselves. So we stuck with this approach, and although hard at first, after a while it was fairly straight forward. By the end we allocated responsibilities in a way which seemed to minimise coupling, in that it proved easy to add new tests (beyond the ones specified in the exercise specification).

The exercise and the lecture strongly advocated the idea that the tests a developer writes should not only make them think about implementation quality, but also about design quality. This spurred us to read more about the "Tell don't Ask" principle, or the "Law of Demeter as it is widely referred to in the literature. The article "Introducing Demeter and its Laws" by Brad Appleton (http://www.bradapp.com/docs/demeter-intro.html) introduces the motivation for the principle, and gives some historical background. More interestingly it outlines a key difficulty relating to the principle, namely that of having to write many wrapper methods in order to propagate method calls to components (and avoid train wreck code), and also discusses a solution to this in the form of a tool suite known as the Demeter Tools.
We then turned to the recommended text book "Growing Object Oriented Software, Guided by Tests"" by Steve Freeman and Nat Pryce. The book introduced a useful design technique which can facilitate the process of Role Based Design. The idea is to use index cards to write down the candidate, responsibilities and collaborators (CRC). Even though the tutorial exercise was relatively straight forward to model and to have a mental picture of the object structure due to the small size of the system, it was clear that this technique could come in handy when trying to design a larger system.

Overall we learnt a lot this week, and most importantly re evaluated Test Driven Development as a valuable design strategy. We were happy with both the lecture and the tutorial, as well as how they complemented each other. So we have no particularly useful feedback to offer ☺