Generating Mock-Based Test Automatically

Sabrina F. Souto[1], Renato Miceli[1], Dalton Serey[1]
1Systems and Computing Department –
Federal University of Campina Grande (UFCG)
Caixa Postal: 10.106 - 5918109-970 – Campina Grande – PB – Brazil

{sabrina, dalton}@dsc.ufcg.edu.br, renato@lsd.ufcg.edu.br

Abstract—*Mock objects are used to improve both efficiency and effectiveness of unit testing. They can completely isolate objects under test from the rest of the application allowing easier root cause analysis of defects. Writing tests that use mocks, however, can be a tedious, costly task and may lead to the inclusion of defects. Furthermore, mock-based unit tests are known to be short-lived – they are usually discarded due to several design changes on the system. In this paper, we propose a technique that generates mock-based tests to face the mentioned drawbacks. Based on the analysis of execution traces, interactions between a target object and its collaborators are captured, by using Aspect Oriented Programming. We also present Automock, a proof of concept tool developed to evaluate the feasibility of the technique.*

Keywords- Software testing; aspect oriented programming; mock objects; test automation.

I. Introduction

According to Kerievsky [1], the main purpose of mock objects is to isolate the class under test (CUT) by replacing its collaborators by test implementations, or mocks. In each test, the developer describes the interactions he/she expects the object to have with its collaborators, and simulates any response expected by the behavior under test. During the test, the mocks check whether they have been invoked as expected. They also react in predefined ways, providing support to complex tests in which long interaction patterns must be considered. In such a test, the CUT is unable to perceive whether it is interacting with real or with mock objects. Mocks, therefore, can be of great help in writing real unit testing, in which only one unit is effectively tested.

In testing, a wide range of objects can play the role of collaborators. They can be memory-based objects or they can access external systems, databases, internet connections, among others. Figure 1 illustrates a typical testing scenario, in which a CUT interacts with real collaborators. Figure 2, depicts the same scenario in which the collaborators were substituted by mocks.

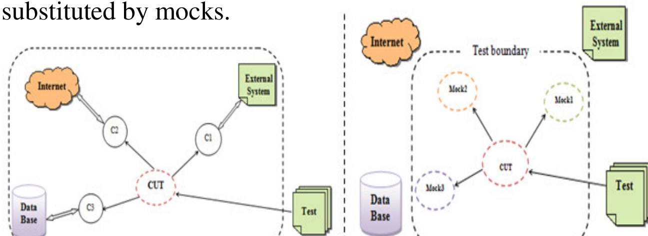

Figure 1a. A test using real collaborators

Figure 1b. A test using mock objects instead of real collaborators

Mocks can also make tests more effective and efficient. Mackinnon [2] states that mocks make testing more effective because mock-based tests provide more precise information about failures and defects. And, because mock-based tests use simplified simulations of the real collaborators, they tend to run faster, especially when collaborators access databases, internet connections or other external systems. On the other hand, in practice, develop, maintain and reuse mock objects is a repetitive and time-consuming task, due to the fact that mock objects need several settings to work, even with the use of frameworks, like EasyMock [3], to write them.

In this paper, we present a technique and a tool that automate the synthesis of mock-based unit tests. The technique is based on automatically identifying interactions among objects in a given testing (or use) scenario. The tool analyses execution traces and identifies all interactions, including data exchanged, between a chosen target object and its collaborators. That analysis is achieved through instrumentation of the previous existing test code, using Aspect Oriented Programming [4].

II. Our Approach

The technique consists of three phases: static analysis, dynamic analysis and mock code generation (see Figure 2 for a high level representation of this workflow). During static analysis, we identify all objects that collaborate with the CUT. In the dynamic analysis phase, we instrument and execute the original test, which is a JUnit [5] test class, in order to capture and record the interactions between the CUT and the previously identified collaborators. In the mock code generation phase, the recorded interactions are used to generate the code for the mock based version of the input unit test.

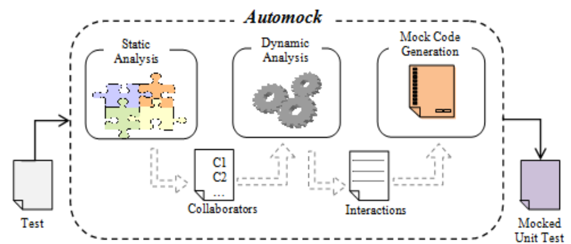

Figure 2. General flow of *Automock*

A. *Implementation Deatils*

The static analysis is performed through a parsing technique. It reads a JUnit [5] test source code and produces

a mapping of instances and types of all objects that collaborate with the CUT.

During the dynamic analysis, the test is instrumented using aspect oriented programming [4]. Compared to conventional instrumentation techniques, aspects provide a clean and structured way of implementing the capture of interactions. The output of this phase consists of a structured log describing all interaction and data exchanged between the CUT and the collaborator objects, in the context of a given test. It is important to mention that only the target object, its collaborators and the test are instrumented. The result is a more efficient instrumentation compared to ones that instrument the entire the system.

The third phase ends the process by generating the mock based test code. The original test is used as a template, in which the actual collaborators are substituted by their corresponding mocks, by using EasyMock [2] framework.

B. *Aspect Oriented Programming*

In *Automock*, AOP is used to instrument code by capturing the interactions among the CUT and its collaborators, in the context of the test, by the use of AspectJ, an Aspect Oriented Programming (AOP) extension to the Java language [4, 6]. This approach is similar to the capture phase of GenUTest [7], which utilizes aspects for the capturing process. The crosscuting concerns used here are captured by means of the following pointcut:

```
Pointcut testClass(): cflow(execution(
* testCase.*(..))) && call(* *.*(..)) &&  !(within(Automock))
```

The capture code, specified by pointcut defined above, captures important data about the actual states of the objects being analyzed by using the reflective constructor thisJoinPoint and by creating a special object called objectRecord containing all the analyzed data. These objects are stored on to a list, and then serialized and logged by using a special library supporting serialization. The dynamic analysis ends after parsing all the objectRecords captured by the aspect to an xml file. This type of file is useful for easy data manipulation, and can be accessed by using the same library to deserialize the list.

C. *Preliminary Evaluation*

Our approach has been investigated by means of an experiment in order to reduce the efforts of testers and the time spent on implementing mock code for tests. These experiments have been done using OurBackup software [8]. Our experiment is in the initial phase, and consists in two main tasks:

1) Develop mock code for test classes of the system in manually way, with the aid of the framework EasyMock [2];

2) Generate mock code for the same test classes using Automock.

In order to execute these tasks, we chose two test classes of the system, one with 243 lines of code and other with 905 lines of code. We have timed each task and counted the number of lines of each resulting mocked test, two manually and two generated by *Automock*. As a result, we have the number of lines of mock code produced.

We have evaluated the results in terms of effort and time spent to develop mock code for each test class. Comparing the lines of code with and without mock code, we have that 58% of programmer's effort in produce test code is related to mock code. If the programmer does not have to develop mock code, our reduction is of 58%, it means that the programmer will only have to produce the test code normally, and then generate mock code using *Automock*. In terms of time reduction, we have compared the time to produce mocks both manually and automatically. As mentioned before, we selected two tests to compare the time spent in producing mock-based tests. The first test took 15 minutes using *Automock*, and 30 minutes to be produced manually – a gain of 51% of time. For the second test, the gain was of 94% – times were 20 minutes against 6 hours, with and without *Automock*, respectively.

Further experiments are still necessary to support a full evaluation and to generalize results. This, however, is under work.

III. Conclusion and Future Work

In this paper, we presented a technique that automatically generates mock code for unit tests. In order to support and evaluate the technique, we developed a tool and applied it in a test development environment. Although the evaluation can only be considered as preliminary, the results are promising: development effort measured in terms of development hours was reduced in 51% and 94% in the two scenarios evaluated. Furthermore, testers were convinced that the technique can be very helpful during test development. And that the mock code generated is as readable as man-made mock based code.

Future work will follow two directions. First, we will further evaluate the gains that can be derived by applying the technique by means of more rigorous experiments. Second, we plan to improve the tool in order to make it both more efficient and easier to use possibly by developing an *Automock* plug-in to the Eclipse IDE [9].

References

[1] Kerievsky, J. (2007), "TDD: Don't Mock It Up with Too Many Mocks", In: Point/counterpoint. IEEE Software, 24(3):81–83.

[2] Mackinnon, T., Freeman, S. and Craig, P. (2001) "Endo-testing: unit testing with mock objects" In: Extreme Programming Examined, Addison-Wesley, pages 287-301.

[3] Freese, T. (2004), "Easymock". (Visited May 2009), http://www.easymock.org.

[4] Kiczales, G. (1997) "Aspect-oriented programming" In: Proceedings of the European Conference on Object-Oriented Programming.

[5] JUnit. (Visited November 2008), http://www.junit.org/.

[6] The AspectJ Project. (Visited September 2008), http://www.eclipse.org/aspectj.

[7] Pasternak, B., Tyszberowicz, S. S., and Yehudai, A. (2007) "GenUTest: A Unit Test and Mock Aspect Generation Tool", In: Haifa Verification Conference, pages 252–266.

[8] Oliveira, M. I. S., Cirne, W., Brasileiro, F., and Guerrero, D. (2008) "On the impact of the data redundancy strategy on the recoverability of friend-to-friend backup systems", In: Brazilian Symposium on Computer Networks and Distributed Systems.

[9] Eclipse (Visited June 2009), http://www.eclipse.org