An Empirical Study of Testing File-System-Dependent Software with Mock Objects

Madhuri R. Marri[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2], Wolfram Schulte[2]

[1]Department of Computer Science, North Carolina State University, Raleigh, NC
[2]Microsoft Research, One Microsoft Way, Redmond, WA
[1]{mrmarri, txie}@ncsu.edu, [2]{nikolait, jhalleux, schulte}@microsoft.com

Abstract

Unit testing is a technique of testing a single unit of a program in isolation. The testability of the unit under test can be reduced when the unit interacts with its environment. The construction of high-covering unit tests and their execution require appropriate interactions with the environment such as a file system or database. To help set up the required environment, developers can use mock objects to simulate the behavior of the environment. In this paper, we present an empirical study to analyze the use of mock objects to test file-system-dependent software. We use a mock object of the `FileSystem` *API provided with the Pex automatic testing tool in our study. We share our insights gained on the benefits of using mock objects in unit testing and discuss the faced challenges.*

1 Introduction

Unit testing aims to test a single unit in isolation, such as a class, independent of other units [6]. Often, it is difficult to test units in isolation when they interact with other units or environment such as a database. Test generation for such units requires much effort to achieve a required state of the environment. For instance, testing a unit that interacts with the `FileSystem` API [2] requires "real" files to exist or be created in order to ensure comprehensive testing of the unit (e.g., achieving high structural coverage). In such cases, automated test generation tools would fail to generate high-covering tests. To simulate and model the required environment to test the unit under test, developers can use *mock objects* [9].

A mock object[1] is an implementation for simulating the required environment. A mock object is not the unit under test in unit testing, but is essential for test generation or execution. There exist frameworks [3] that generate mock objects by providing trivial method implementations of an actual object. The generated methods of these mock objects provide some default return values without reflecting the logic of the actual object. To generate necessary environments to sufficiently test the unit, developers need to specify and model the behavior of the mock methods[2]. Furthermore, developers have to manually include all the expected return values of the methods in the mock objects.

Parameterized mock objects [11] can be used to generalize mock objects and address the preceding issue by simulating the various possible return values automatically. Parameterized mock objects are related to *Parameterized Unit Tests* (PUTs) [13] used by Pex [12] in test generation. PUTs are unit tests with parameters. Developers can write PUTs, which can be used to generate numerous conventional unit tests. Pex is an automated white-box testing tool that accepts PUTs and generates conventional unit tests by symbolically executing the code [7]. Pex generates a set of conventional unit tests that try to achieve high block coverage of the code under test.

As PUTs can be used to generalize conventional unit tests, parameterized mock objects can be used to generalize mock objects. Developers can write parameterized mock objects so that a single call to a mock method of mock objects can return different values expected by the unit under test. Pex uses symbolic execution [7] to track how the value returned by a method of parameterized mock objects is used. Depending on the subsequent branching conditions on the value returned by a mock method, Pex executes the PUT multiple times, trying different return values to explore new execution paths.

In this paper, we present an empirical study of using a parameterized model[3] of the `FileSystem` API [2] to test the *CodePlex Client* [1] project. We present the benefits of using mock objects (especially the parameterized model) and the challenges of using mock objects.

[1]Unlike the traditional definition of an object, which is an instance of a class, we refer to a simulating class as a mock object.

[2]Methods in mock objects are referred to as *mock methods*.

[3]A parameterized mock object is referred to as a parameterized model throughout the rest of the paper.

AST'09, May 18–19, 2009, Vancouver, Canada
978-1-4244-3711-5/09/$25.00 © 2009 IEEE

The rest of the paper is organized as follows. Section 2 presents background on mock objects. Section 3 presents details on the software under study and presents an example. Section 4 presents the benefits of mock objects. Section 5 presents challenges of using mock objects. Finally, Section 6 concludes.

2 Background on Mock Objects

In this section, we present background information on mock objects. There are two major purposes of using mock objects: (1) to test if the code under test interacts in an expected manner with the surrounding objects in the system [4] and (2) to provide the required environment for a test generation tool to generate high-covering tests for the unit under test. This paper focuses on a study of using mock objects primarily for the second purpose, i.e., to provide the required environment for a test generation tool to generate high-covering tests.

2.1 Complexity Levels of Mock Objects

Developers can implement mock objects with various levels of complexity, providing various levels of accuracy in simulating actual object behaviors. Based on the patterns proposed by Tillmann and Halleux [11], mock objects can be implemented to be a lightweight simulation (called structural simulation) or a complex simulation (called structural and logical simulation) of a real environment. The lightweight simulation can accept any input values and return default or random values for mock methods. A logical simulation can impose restrictions on the input values and the return values of mock methods. An implementation with such constraints can specify that there is a specific return value for a given input value. A more sophisticated implementation of a mock object maintains the state of the mock object instance and behaves consistently. The state of an instance is defined by information maintained by the instance (e.g., the existence of a file with the name *xyz* can be maintained by an instance of a mock object of `FileSystem`).

2.2 Parameterized Model

The `PFileSystem` [8] parameterized model used in our study is an example of a sophisticated mock object implementation that maintains its state. This type of implementation maintains an internal state and assures consistent behavior. An instance of the `PFileSystem` model logs information about any created file or directory. The state of the `PFileSystem` instance is defined by the file and directory information maintained by that instance and is only internal to that instance. When the parameterized model's state does not contain information needed to return a specific value, the parameterized model is implemented in such a way that Pex can choose to return a value based on how it is used in the subsequent code.

```
01:  bool DirectoryExists(string path)
02:  {.......
03:      /*dirStack all the directories that
04:      need to exist for the path to exist*/
05:      foreach(var dirPath in dirStack)
06:          {//check if exists already
07:          var dirInfo = Locate(dirPath);
08:          ..........
09:          if (dirInfo.DirectoryCreated)
10:          {
11:            check = false;//set to false if found
12:            break;
13:          }
14:      }
15:      // Create if possible
16:      if (check){
17:      // Ask it to Pex
18:          var call = PexChoose.FromCall(this);
19:          if (call.ChooseValue<bool>("Create Directory
                 ¨ + path + "¨ or Not")){
20:            // Ensure path to file
21:            foreach (var dirPath in dirStack){
22:              var dirInfo = Locate(dirPath);
23:              ......
24:              CreateSingleDirectory(dirPath, false);
.........
```

Figure 1. A method from `PFileSystem` that checks if a directory exists

We next illustrate the behavior of the parameterized model with an example. Figure 1 shows the `DirectoryExists` method of the `PFileSystem`. In the method, when the information about the directory being looked up is not found (Lines 11 and 16), the used `PexChoose`[4] API (Line 18) can provide values to decide whether to create a directory. In the method, a choice provider is obtained by invoking the static method `PexChoose.FromCall` (Line 18). When Pex identifies the method call to choose a value (the method call to `ChooseValue` shown in Line 19) on the choice provider `call`, Pex generates a value of the `bool` datatype. Pex indentifies a condition check on the generated value (Line 19) and in the following executions, Pex generates values to cover other paths in the code. Therefore, Pex can generate various necessary states of an instance of the parameterized model to test the code under test.

In our study, we show that there is a need for an enhanced model of mock objects. When multiple APIs invoked by the unit under test interact with the same environment, then all these APIs need to be mocked and information needs to be maintained for assuring consistent states for their corresponding multiple mock objects. The enhanced model should allow all the *related* mock objects to behave consistently. For example, `PFileSystem` is a mock object of the `FileSystem` API and `PDirectory` is a mock object

[4] Pex choices APIs supply relevant values to trigger different execution paths in the code [11].

of the `Directory` API. Since both APIs deal with directory operations (interacting with the same environment), the state maintained by an instance of `PFileSystem` should be consistent with that of an instance of `PDirectory`, e.g., if `PFileSystem.DirectoryExists()` returns `true`, then the `Directory.isExists()` should also return true.

3 Software under Study

CodePlexClient [1] is a source control client that enables users to edit workspace offline and synchronize with the server when connected. We tested 8 methods belonging to the `TfsLibrary` and `CodePlexClientLibrary` namespaces in the CodePlexClient project. The rationale behind choosing these methods is that they involved interactions with the `IFileSystem` interface. `FileSystem` [2] is one commonly used API in file-system-dependent software. The latest Pex release [8] provides a model of `FileSystem` called `PFileSystem` with its release package. `PFileSystem` is a mock implementation of the `FileSystem` API built on top of the `IFileSystem` interface. The `PFileSystem` model uses the Pex choices API [11] to generate various possible states of a mock object instance. When `PFileSystem` is used in a PUT to test a unit, Pex generates different object states of the `PFileSystem`. We used `PFileSystem` API to test the 8 methods and achieved 58% block coverage.

We next show an example to illustrate how we use the parameterized model in our testing. When a folder is shared with the server (i.e., the folder is *tracked*), a metadata folder and an entry file are created locally to indicate that the folder is tracked. Figure 2 shows a PUT to test the method `UntrackFolder` of the `TfsState` class. This method under test is responsible for untracking a folder, i.e., removing any associated metadata folder or the entry file that holds information about the folder being tracked. To test the code, we carry out the following procedure to write a PUT. We pass test inputs as parameters to the PUT; in our example, the test input is the path of the folder that was actually deleted (and needs to be untracked). In the body of the PUT, we create an instance of the `PFileSystem` and pass it as the parameter to the constructor of `TfsState`. The constructor of `TfsState` accepts an instance of a class implementing the `IFileSystem` interface; this scenario shows that using an abstraction layer such as the `IFileSystem` interface is one way of improving the testability of the unit under test. We then assert that the metadata folder was successfully deleted. On executing the PUT, we achieve 70% block coverage, covering most of the cases related to the `PFileSystem`.

```
01: public void testDeleteMetadataPUT(
02:  [PexAssumeUnderTest]string value){
03:     PexAssume.IsTrue(value.Length > 0);
04:     PexAssume.IsTrue(
            value.StartsWith(@"A"));
05:     PFileSystem system = new PFileSystem();
06:     TfsState state = new TfsState(system);
07:     state.UntrackFolder(value);
08:     string st = state.GetMetadataFolder(value);
09:     PFileInfo info = system.Locate(st);
10:     PexAssert.IsFalse(info.DirectoryExists,
11:         "File was not successfully deleted");}
```

Figure 2. PUT to test the method `UntrackFolder` of the `TfsState` class

4 Benefits of Using Mock Objects

We next present the benefits of using mock objects in general and parameterized models in particular. Our study identifies the following two benefits: (1) mock objects enable unit testing of the code that interacts with external APIs related to the environment such as a file system, and (2) a parameterized model helps generate various possible states of mock objects to enable the generation of high-covering unit tests. We next elaborate on these benefits with the example shown in Section 3.

The PUT shown in Figure 2 tests the method `UntrackFolder`. The method `UntrackFolder` initially checks whether the folder is tracked by checking whether a particular entry file exists (using the `FileExists` method of `IFileSystem`) and whether a metadata folder exists (using the `DirectoryExists` method of `IFileSystem`). Therefore, the number of tests depends on the various combinations of whether the metadata folder and the entry file exist. For Pex to generate high-covering tests, there are two requirements with respect to the interactions of the code under test with the `IFileSystem`: (1) Pex needs to carry out symbolic execution when the actual file and directory do not exist, (2) Pex needs to generate high-covering tests for various combinations of whether a particular file or directory exists. These two requirements can be fulfilled by using the `PFileSystem`, a parameterized model of the `FileSystem`. In the `PFileSystem` model, the `DirectoryExists` and `FileExists` methods return `true` if a directory or file already exists (reflected by the state maintained by the `PFileSystem` instance). However, when the directory or file does not exist, due to the use of Pex choices APIs, Pex chooses to either create the required directory or file and return `true` or return `false` (as shown in Section 2.2). Consequently, Pex generates various states of the parameterized model for testing the code under test.

Through our example, we show that mock objects enable unit testing of the code that interacts with external APIs such as `FileSystem`, and mock objects provide an easy way to generate a complex object state that is essential to test a unit. With a parameterized model, the concern of be-

ing able to generate various possible *states* of the mock object (such as the states with different combinations of the file and directory existences in our example) is reduced in the way shown in our example.

5 Challenges When Using Mock Objects

In our study, we identify that using mock objects can face difficulties when there are other APIs invoked by the unit under test that depend on the *environment* created or maintained by an instance of a mock object. In this section, we highlight that when using mock objects, it is essential to maintain consistent environment states across multiple mock objects (if multiple mock objects are used to interact with the same environment). We next present description of the challenges and then illustrate them with an example.

5.1 Description of Challenges

Previous work [10, 5] identifies general challenges when writing mock objects or models in unit testing. A well-known challenge with mock objects is the amount of effort required to implement sufficient mock objects. There exist frameworks [3] that automatically generate a mock object; however, it is still the responsibility of developers to simulate possible return values of the mock methods. Pex addresses the preceding issue with a parameterized model as shown in our example in Section 4.

However, using mock objects can cause trouble when the code under test involves interactions with multiple APIs that use the same data or interact with the same environment. An example of such code under test can be code that involves interactions with `File`, `Directory` and `FileSystem` APIs [2] to interact with the same file system environment. Using a single mock object can lead to problems (such as a false failure warning due to invalid test setup) since there are other APIs that depend on the environment created or maintained by an instance of the mock object. The challenges are two-fold in such a case:

- Challenge 1. Mocking a single API is not sufficient when the code under test invokes multiple APIs and these APIs depend on how the mock object for the mocked API works.
- Challenge 2. Multiple APIs invoked by the code under test interact with the same data or environment, and if the APIs are mocked, then their mock objects should be synchronized with each other.

5.2 Example of Challenges

We next illustrate the challenges with an example. Figure 3 shows the method responsible for adding files

```
01: public void Add(string localPath,
        bool recursive,SourceItemCallback callback)
02: {
03:     Guard.ArgumentNotNullOrEmpty(localPath, "localPath");
04:     if (fileSystem.DirectoryExists(localPath))
05:         Add_Folder(localPath, recursive, callback, true);
06:     else if (fileSystem.FileExists(localPath))
07:         Add_File(localPath, callback, true);
08:         ......
```

Figure 3. Code to add a file in the workspace offline

```
01: bool OnBeforeAddItem(SourceItem item)
02: {
03:     if (alwaysAdd)
04:         return true;.....
05:     if (answer == "a")
06:         alwaysAdd = true;
07:     else if (answer == "d")
08:     {
09:         if (item.ItemType == ItemType.File)
10:             File.Delete(item.LocalName);
11:         else
12:             Directory.Delete(item.LocalName, true);
13:     }
14:     return (answer == "y" || answer == "a");
15: }
```

Figure 4. Code to add or delete on synchronizing with the server

or directories locally. Figure 4 shows the method that deletes the files or directories (if the user chooses to) on synchronizing with the repository. We write a PUT to add a directory and check whether the directory is deleted on synchronizing with the server. The method to add the directory (`Add_Folder()` in Line 5 of Figure 3) invokes the `IFileSystem.CreateDirectory()` method. In our PUT, we use the `PFileSystem` model (recall that `PFileSystem` is a class implementing the `IFileSystem` interface) and therefore the `Add_Folder()` method invokes the `PFileSystem.CreateDirectory()` method. The method to delete the directory invokes the `Directory.Delete()` method to delete the directory (Line 12 in Figure 4).

On executing the PUT, the generated unit tests fail with an exception and Pex shows a *Testability Issue* (Pex shows a testability issue if the code under test invokes an API that depends on the environment). The reason for the exception is that the method to delete the directory (`Directory.Delete()`) does not find the directory path. The `Directory` API is not able to find the path since the `PFileSystem` does not actually create a directory but only logs the information locally since `PFileSystem` is a mock object.

The reason for the exception can be mapped to the challenges described in Section 5.1:

- Challenge 1. Using the `PFileSystem` mock object alone was not sufficient to test the unit as there were other APIs (invoked by the code under test) that interact with the same environment as the mock object.

- Challenge 2. Simply mocking the `Directory` API would not solve the problem, as the same directory information should be shared by both mock objects, i.e., a mock object of the `Directory` API should use the same log used by `PFileSystem`.

A solution to the problem described in our example can be to modify the code under test to replace the `Directory.Delete` method call with the `IFileSystem.DeleteDirectory` method call for the purpose of testing. However, the complexity could be elevated when the code under test requires invoking a method of the `Directory` API whose functionality is not provided by the `PFileSystem` or even its interface, such as accessing the date or time information of the directory. In such a case, a possible solution can be to implement (or extend) a single mock object to provide all the methods that are required to interact with a certain environment, which is otherwise accessed using multiple APIs. In our example, we can extend the `PFileSystem` to provide methods that can be used to replace the actual method calls to the `Directory` or `File` APIs. Pex gives hints on the existence of problematic APIs (that interact with the environment) in the code under test through the testability flags and suggests that the testability issues be resolved by writing a mock object[5]. However, manually identifying the different APIs that interact with the same data or environment is cumbersome as the process involves understanding the information dependency among various APIs.

As a result of our study, we find that it is important to identify all the required objects that need to be mocked or modeled. In addition, it is essential to identify the dependency among APIs; the dependency need not only be defined by direct interaction such as invoking methods, but also be defined by indirect interactions such as operations on the same data or environment. We believe that these tasks can be achieved by automating the process of identifying different external APIs used and analyzing the information dependency among these APIs.

6 Conclusion

We conducted an empirical study to analyze the use of mock objects in unit testing, specifically to test a file-system- dependent software. We used a paramterized model of the `FileSystem` API to test the CodePlex Client project. We showed that using a mock object can ease the process of unit testing. We also identified the challenges faced in testing code when there are multiple APIs that need to be mocked. Additionally, we highlighted the need to automate the process of identifying APIs that need to be mocked and dependencies between the identified APIs to ease the process of testing.

[5] `http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/testability\%20issue.html`

Acknowledgments

This work is supported in part by NSF grant CCF-0725190, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

References

[1] Codeplex client. `http://www.codeplex.com/CodePlexClient`.

[2] .NET Framework Class Library. `http://msdn.microsoft.com/en-us/library/ms229335.aspx`.

[3] Wiki. mock objects. `http://www.mockobjects.com/`.

[4] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock Roles, not Objects. In *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 236–246, 2004.

[5] R. Hightower, W. Onstine, P. Visan, D. Payne, J. D. Gradecki, K. Rhodes, R. Watkins, and E. Meade. Professional Java Tools for Extreme Programming Ant, XDoclet, JUnit, Cactus, and Maven. Wiley Publishing, Inc.

[6] IEEE-Standards-Board. IEEE Standard for Software Unit Testing: An American National Standard, 1999.

[7] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[8] S. H. Kong, N. Tillmann, and J. de Halleux. Automated Testing of Environment-Dependent Programs - A case study of modeling the File System for Pex. In *Proceedings of the International Conference on Information Technology - New Generations (INFG)*, 2009.

[9] T. Mackinnon, S. Freeman, and P. Craig. Endo-Testing: Unit Testing with Mock Objects. In *Proceedings of the Internation Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)*, pages 287 – 301, 2000.

[10] S. Stewart. Approaches to Mocking. `http://www.onjava.com/pub/a/onjava/2004/02/11/mocks.html#Approaches`.

[11] N. Tillmann and J. de Halleux. Parameterized Test Patterns For Effective Testing with Pex. `http://research.microsoft.com/en-us/projects/pex/pexpatterns.pdf`.

[12] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *Proceedings of the Internation Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[13] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proceedings of the European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.