


RISE

A stylized, light gray fish logo is positioned on the right side of the slide, partially overlapping the black background. The fish is facing left and has a simple, rounded body with a small circle for an eye.

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

Madhuri R. Marri, Tao Xie  
(NC State University)

Nikolai Tillmann, Peli de Halleux, Wolfram Schulte  
(Microsoft Research)

# File-System-Dependent Software



- Characteristic: frequent interactions with the file system environment
- Challenge: construction of high-covering unit tests and their execution require appropriate interactions with the environment
- Solution: use of **mock objects**
- Contribution: an empirical study in investigating the benefits and challenges of using mock objects

# Outline



- Mock Object
- Parameterized Model
- Study Setup
- Study Results
  - Benefits of Mock Objects
  - Challenges of Using Mock Objects

# Mock Object



- An implementation for simulating the required environment
- Example: `IFileSystem` (abstraction layer over file system)

## FileSystem

```
public string[] GetFiles(  
    string path, string searchPattern)  
{  
    return Directory.  
        GetFiles(path, searchPattern);  
}
```

## MockFileSystem

```
public string[] GetFiles(  
    string path, string searchPattern)  
{  
    return new string[0];  
}
```

# Mock Object (Cont.)



Three types of implementations based on complexity

## ■ Lightweight simulation

accepts any input values and return default or random values for mock methods  
e.g., implementation of `MockFileSystem`

## ■ Complex simulation

Imposes restrictions on the input values and return values of mock methods  
e.g., when the path is valid, `FileExists` returns true

## ■ Sophisticated simulation

maintains the state  
e.g., the information of a file (such as file exists or file content) with the name `xyz` can be maintained by an instance of a mock object of `FileSystem`

# Mock Object (Cont.)



- With lightweight simulation and complex simulation, developers need manually include all the expected return values of the methods
- ***Parameterized model*** can be used to address the issue

# Parameterized Model



- Generalized mock object (sophisticated simulation)
  - A single call to a mock method can return different values expected by the unit under test
- Related to Parameterized Unit Tests (PUTs)

# Parameterized Unit Testing



- *Parameterized Unit Test* =
  - Unit Test with *Parameters*
- Separation of concerns
  - Data is generated by a tool
  - Developer can focus on functional specification

```
void Add(List list, int item) {  
    var count = list.Count;  
    list.Add(item);  
    Assert.AreEqual(count + 1, list.Count);  
}
```

# Pex on MSDN DevLabs

## Incubation Project for Visual Studio 2010



Pex – Automated White Box Testing for .NET - Windows Internet Explorer

http://msdn.microsoft.com/en-us/devlabs/cc950525.aspx

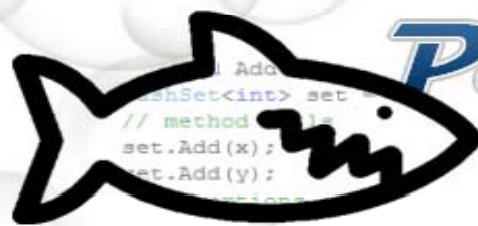
msdn Search MSDN with Live Search Web

DevLabs

Home About Projects

Microsoft Popfly Small Basic **Pex** CHESS

<http://research.microsoft.com/pex>



Pex

Automated White Box Testing for .NET

Dynamic Symbolic Execution

value	Details...	PexMethod	Add Preco
1 null	Stack trace	public str	Add Precond
2 ""	System.NullRe	at StringExt	if (value == r
3 \0	at StringExt	Run Pe	

About Pex – Automated White Box Testing for .NET [see all DevLabs projects...](#)

Pex (Program EXploration) produces a traditional unit test suite with high code coverage. A parameterized unit test is simply a method that takes parameters, calls the code under test, and states assertions. Given a parameterized unit test written in a .NET language, Pex automatically produces a small unit test suite with high code and assertion coverage. To do so, Pex performs a systematic white box program analysis.

Pex learns the program behavior by monitoring execution traces, and uses a constraint solver to produce new test cases with different behavior. At Microsoft, this technique has proven highly effective in testing even an extremely well-tested component.

Play with Pex, stress it, evaluate it, and [tell us what you think.](#)

# PFileSystem Model



## ■ PFileSystem

```
public string[] GetFiles(
    string path)
{
    ...
    if (!info.FilesGenerated)
    {
        info.FilesGenerated = true;
        var call=
            PexChoose.FromCall(this);
        while (call.ChooseValue<bool>)
        { //Generate new file under
            "path"
            ...
            ...
        }
    }
}
```

- Code under test interacts with a parameterized model during testing
- Depending on subsequent branching conditions on the value returned by a mock method
  - Execute the PUT multiple times, trying different return values to explore new execution paths

```
...
string [] files = f.GetFiles(s);
if (files.length == 2)
    if (files[0] == @"c:\config.txt")
        ...
}
```

# Study Setup



- Goal: study benefits and challenges of using mock objects & parameterized model
- Subject application: CodePlex Client, a source control version management system
- Testing 8 methods of `TfsLibrary` & `CodePlexClientLibrary`
  - interactions with file-system environment replaced with parameterized model `PFileSystem`

<http://www.codeplex.com/CodePlexClient>

# Benefits

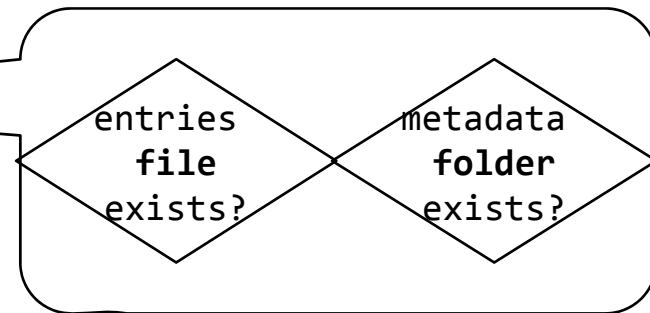


- B1: Mock objects enable unit testing of the code that interacts with environment
- B2: Parameterized model helps generation of various possible states of mock objects to enable generating high-covering unit tests
- Automatically generated unit tests achieved 58% of block coverage (maximum 70% for a method) covering most of cases related to `IFileSystem` (`PFileSystem`)

# Benefits (Cont.)

## ■ Example

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



# Benefits (Cont.)

```
//pseudo code
bool FileExists(string path)
{
    info = getInfo(path);
    if(info is null)
    {
        var call= PexChoose.FromCall(this);
        if (call.ChooseValue<bool>)
        {
            //create new file info and store
            return true;
        }
        return false;
    }
    return true;
}
```

tNotNull]string value)

entries  
file  
exists?

metadata  
folder  
exists?

maintains info,  
generates  
high-covering test

(1) symbolic exec when  
actual file and directory do  
not exist  
(2) generate high-covering  
tests for combinations of  
whether a particular file or  
directory exists

# Challenges



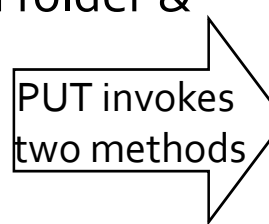
- C1: Mocking a **single API** is not sufficient when the code under test invokes **multiple APIs** and these APIs depend on how the mock object for the mocked API works
- C2: **Multiple APIs** invoked by the code under test interact with the same data or environment, and if the APIs are mocked, then their **mock objects should be synchronized with each other**

# Challenges (Cont.)

## ■ Example

Consider a PUT to add a folder & delete the folder

PUT invokes  
two methods



### ■ Psuedo code of method to add

```
.....  
if(ValidDirectoryStructure)  
{  
    fileSystem.CreateDirectory(path)  
}
```

### ■ Pseudo code of method to delete

```
.....  
if (answer == "d")  
{  
    .....  
    Directory.Delete(item.LocalName, true)  
}
```

# Challenges (Cont.)



## ■ Example

Consider a PUT to add a folder & delete the folder

PUT invokes two methods

### ■ Pseudo code of method to add

```
.....  
if(ValidDirectoryStructure)  
{  
    fileSystem.CreateDirectory(path)  
}
```

### ■ Pseudo code of method to delete

```
.....  
if (answer == "d")  
.....  
    Directory.Delete(item.LocalName, true)  
}
```

Generated unit tests fail;  
multiple APIs to interact  
with same environment

# Challenges (Cont.)



## ■ Example

Consider a PUT to add a folder & delete the folder

PUT invol  
to met

## ■ Psuedo code of method to add

```
.....  
if (Val DirectoryStructure)  
{  
    stem.CreateDirectory(path)
```

ethod to delete

Pex shows testability  
issue with **Directory.Delete**;  
Cannot symbolically execute

```
(item.LocalName, true)
```

**Directory.Delete()** does not find the  
directory path

# Challenges (Cont.)

## ■ Example

Consider a PUT to add a folder & delete the folder

PUT invokes two methods

## ■ Pseudo code of method to add

```
.....  
if(ValidDirectoryStructure)  
{  
    fileSystem.CreateDirectory(path)  
}
```

## ■ Pseudo code

```
.....  
if (answer == true)  
{  
    .....  
}
```

**IFileSystem**  
is not the only way of  
interacting  
with the file system  
environment

# Challenges (Cont.)

## ■ Example

Consider a PUT to add a folder & delete the folder

PUT invokes two methods

Therefore, it is important to **identify all the required objects** that need to be mocked or modeled & **identify the dependencies** between the mock objects

## ■ Pseudo code of method to add

```
.....  
if(ValidDirectoryStructure)  
{  
    fileSystem.CreatedDirectory(path)  
}
```

## ■ Pseudo code

```
.....  
if (answer == true)  
{  
    .....  
}
```

IFileSystem is not the only way of interacting with the file system environment

# Conclusion & Future Work



- Empirical study to investigate using mock objects in unit testing
- Benefits and challenges of using mock objects
  - Helpful to generate high-covering unit tests
  - Challenge when more than one API used to interact with the same environment
- Future work
  - Automate the process of building mock objects
  - Methodology of assisting developers in building required mock objects

*Automated*  
**Software** Research  
Group  
Engineering @NCSU

Microsoft  
**Research**

**Thank you**

RISE

<http://research.microsoft.com/pex>

<https://sites.google.com/site/aserggrp/>