Isolation and Pex: case study of cooperation

Dávid Honfi, Zoltán Micskei, and András Vörös

Budapest University of Technology and Economics
davidhonfi@gmail.com, {micskeiz,vori}@mit.bme.hu

September 2013

Abstract

Quality assurance of software is currently getting a wider spread, therefore testing is given a more important role in their development. Selecting the inputs for tests is an elaborate task, but there are tools for generating test inputs from the model or even from the logic of the software. In a recent testing project we experimented with Pex, a test input generation tool, which analyzes the structure of the code. Unit testing involves the isolation of external dependencies from the unit, which is commonly implemented with isolation frameworks. Pex is shipped with an isolation tool called Moles, which is now in deprecated state. Thus we performed a case study to determine whether Pex could seamlessly collaborate with other isolation frameworks. Our initial results show that is is possible to use other frameworks, but Pex could loose from its functionality.

1 Introduction

Nowadays, there is a quickly growing demand for quality assured software, therefore testing is getting an increasingly important place in the development processes. IEEE defines testing as "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [IEE10]. Selecting those test input from the numerous possible combinations, which could uncover bugs in the software or achieve high coverage is a laborious task, therefore several tools have been developed that can derive test inputs based on the source code or some model of the system. When performing testing at the unit (or component) level, one of the main tasks is to isolate the unit under tests from its dependencies. This is usually performed by using an isolation framework to create stubs or mocks.

Recently, we have been working on testing a model checker tool written in .NET. For automated, code-based test generation we experimented with

Pex [TH08]. Pex is a mature structural testing tool that was created by Microsoft Research, and now it is available as a Visual Studio Power Tool. Pex was shipped with an isolation framework called Moles [HT10], however, Moles was deprecated, it will be not developed further. A successor of the Moles framework, called Fakes was published as a part of some versions of Visual Studio 2012. Unfortunately, Fakes cannot be used with the current release of Pex, since Pex is not compatible with Visual Studio 2012. Thus, Pex is currently in a transitional state, where there is no recommended isolation framework available except Moles.

Objective The objective of this case study is to determine whether Pex could seamlessly collaborate with any other isolation framework than Moles.

Contributions We collected the currently available .NET isolation frameworks, and selected the most promising ones. We performed a case study to decide whether those frameworks could collaborate with Pex without sacrificing some of it's test generation capabilities.

The structure of the paper is as follows. Section 2 presents the problem and objective in more detail, and Section 3 formulates research questions and describes the case study design. Section 4 reports on our results, while Section 5 summarizes implications and limitations.

2 Background

Software testing can be categorized from several aspects including it's place in the development process or it's design technique.

Unit testing In this kind of tests, the software is divided into isolated and well-defined parts called units, which are going to be tested separately to avoid that errors outside a unit influence the test results. Isolation can be implemented in many ways, for example using mocks or stubs as shown in Figure 1 [Mes07]. Unit tests are commonly used in object-oriented softwares, in which the unit is one or at the most a few classes. The unit test case itself is a simple call or sequence of calls to a method of the class under test with having it's results compared against the expected. A common pattern used in writing test code is Arrange-Act-Assert which provides a guide how to create these test method's body. It states that the following layout should be used.

- *First period (Arrange):* Preparation, creation, parameterization of the unit under test and other auxiliary variables (e.g. mocks/stubs).

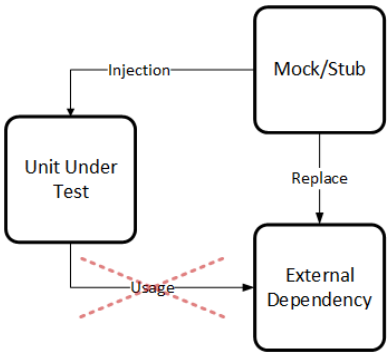

Figure 1: Unit testing with isolation.

- *Second period (Act):* Calls to the unit under test with the variables prepared in the first period.
- *Third period (Assert):* Checks of the results of the call by comparing them to the expected results.

Some test methods may be very similar to each other (e.g. difference only in values of variables), but they represent separate test cases. Parameterized unit tests helps controlling these situations.

Parameterized unit tests (PUTs) This kind of unit tests provide parameters for the method of the test [TS05], which allows the tester to give different inputs. A test case is a list of concrete input values. PUTs lead to the idea of automated input generation.

Test design techniques Automatized input generation can be achieved by the two main ways of test design techniques.

- *Black-box input generation:* The code itself is considered as a black box, which means that the only base is the specification for generating the inputs.
- *White-box input generation:* Also called as glass-box technique, which covers that the code is visible for analysis to create test inputs.

Automated input generation The white-box design technique holds the key for automated generation of inputs, since data can be collected from the code including the feasible paths, possible branches and their conditions. Symbolic execution (SE) uses this idea, because it collects data from the code by running it with symbolic values, then it solves the given constraints by a specific solver to get the concrete inputs for each execution path. Generally speaking about symbolic execution, Pasareanu and Visser

gave an overview about it's place in testing and analysis [PV09]. SE holds a limitation, since the constraint solver may not be able to solve all of the symbolic constraints due to their complexity. Dynamic symbolic execution (DSE) provides solution for this problem in certain situations. It combines concrete and symbolic values during the execution, which makes constraint solving a less complex problem. There are some tools using the idea of DSE, e.g. Pex developed by Microsoft Research. Pex is a .NET tool, which generates values for parameterized inputs using the DSE, thus test cases with high code coverage can be obtained.

Isolation in .NET In .NET there are two main ways for implementing isolation.

- *Proxy-based:* This technique uses a runtime proxy class for detouring the calls to the real object. The disadvantage of this approach is that it can be only used when the Dependency Injection design pattern is used, so the code under test has to be designed to accept object injections. Only this can ensure that the real object can be replaced by the test object during testing.

- *CLR profiler:* This other technique uses the low-level layor of .NET to detour calls in runtime by catching calls to the objects or type of objects. The main advantages over the proxy-based technique are that it can isolate almost every situation and there is no need to design the code for testing.

There are many .NET isolation frameworks, some of them are open-source, but the CLR profiler based tools are usually not, because those are more powerful and indeed more complex softwares, than the proxy-based tools. We collected data of eight .NET isolation frameworks including four proxy-based and four CLR based as shown in Figure 6. Currently, there are three most competitive open-source isolation frameworks for .NET, these are Moq, Rhino Mocks and NSubstitute. Their most important properties are the following.

- *Moq:* Simple, easy to learn syntax, large amount of sources and examples on the internet, one of the most popular frameworks.

- *RhinoMocks:* Comprehensible syntax, many example code, one of the oldest frameworks.

- *NSubstitute:* Very clear and easily understandable syntax, one of the most promising frameworks.

Pex comes with a tool called Moles, which is a CLR profiler based isolation framework that interoperates with Pex. Figure 2 shows the workflow of

testing with Pex and Moles. First of all, the parameterized unit test methods should be created for the source code. These methods include the testing logic and the implementation of isolation with mocks or stubs too by using Moles. Then, Pex should be able to generate inputs for the parameterized method, thus test cases are created.

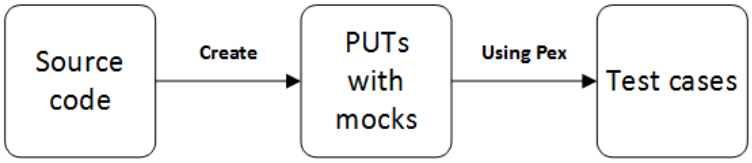

Figure 2: Testing workflow using Pex and Moles.

The current versions of Pex and Moles (both have version 0.94) are capable of cooperating in Visual Studio 2010 with .NET 4.0. The problem is that the development of Moles had already ended and it's not supported anymore, since the successor, Fakes Framework, took it's place in Visual Studio 2012. At first sight this might not look like a problem, but Pex version 0.94 cannot run in the mentioned environment with Fakes. A solution could be to run Fakes in Visual Studio 2010, but it is a solid part of the version 2012, therefore it is impossible to use it in earlier versions. Thus, the cooperation of Pex and other isolation frameworks should be examined.

3 Design overview

In the section, we introduce the research questions and the detailed design of the case study.

3.1 Research questions

We could derive two questions from the main objective of this case study. The importance of these questions in not restricted to our project as Pex may be used in industrial software developments too, since it is publicly available. The answers of our report and case study can help in those situations.

1. Is it possible to use Pex with any isolation framework other than Moles?

2. Does Pex lose from it's functionality when using the mentioned other isolation framework?

3.2 Case study

Method The main part of the context is the C# project called `SimpleExample`, which was created with taking all the possible cases of isolation into account.

The example project consists of two classes in different namespaces as shown in Figure 3. Class A is going to be the class under test. The other class (B) represents the external dependency which has to be isolated. We had to explicitly make sure that Pex handles the B class as an external dependency, so we deleted the appropriate command (`PexInstrumentAssembly`) in the assembly info file of the test project.

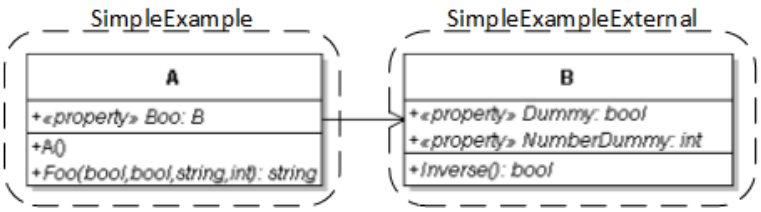

Figure 3: Class diagram of SimpleExample project.

Case We decided to create our own dummy project for the study, because it is easier to include various isolation cases in a self-made, artificial environment specially created for this analysis. Since we are working on an academic project mentioned in the introduction, we had to use open-source isolation frameworks for this study, which are only proxy-based. As two of the most competitive frameworks, namely Moq and Rhino Mocks have very similar functionality, we voted for Moq because we are familiar with it's syntax from earlier projects. NSubstitute was the other framework that we did choose due to it is continuously under development and has a new type of syntax. However, it is important to mention that both Moq and NSubstitute are proxy-based frameworks, which means that they have limited functionality compared to CLR profiler-based frameworks like Moles, JustMock or Isolator. The versions of each tool are shown in Table 1. Our case study gives quantitative results of the numbers and types of generated test cases. Each test case is identified by it's input parameters. We compare these results to reference, which is in this study the test case results of the collaboration of Pex and Moles.

IDE version	Visual Studio 2010 Ultimate
.NET version	4.0.30319
Moles version	0.94.51023.0
Moq version	4.0.108247
NSubstitute version	1.4.3.0

Table 1: Used tools and versions.

Analysis We did experiments with Pex and different isolation cases. This study is built from our experiences, so the isolation cases were designed to cover most of the typical use cases of an external dependency and the usages of Pex. We separated two main actions related to external dependencies, as shown in the following enumeration.

1. *First part:* Call to the external dependent object. (Three different cases.)
2. *Second part:* Access to the external dependency. (Two different cases.)

Figure 5 shows the combination of these types and also marks the possible cases. Naturally, Case 1.2 and 2.2 involves extending the list of parameters of method `Foo` with a `B` type of value, which is not shown in Figure 3. The return value of method `Foo` depends on two bool parameters, but before that it calls a `B` object's `NumberDummy` setter by assigning a value to the property. This setter was implemented so that if the received value equals fifteen, then it throws an exception. This process is shown in Figure 4.

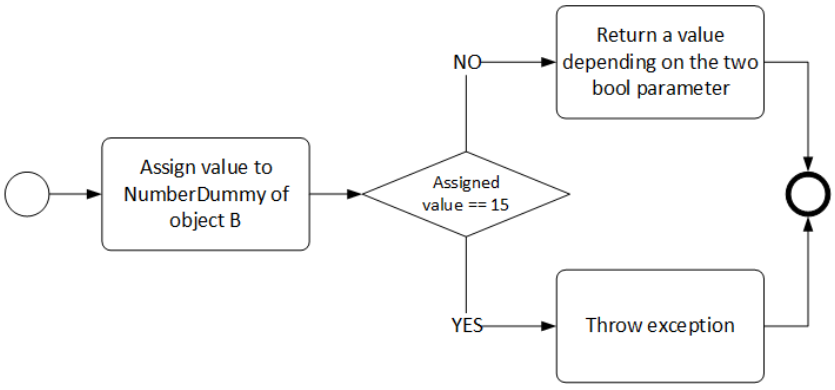

Figure 4: The inner logic of method Foo.

The `Inverse` method of class `B` always returns the negated value of it's property named `Dummy`. This method is used in cases 2.1-2.3.

4 Results

In this section, we introduce the results of our case study.

4.1 Case 1.1: Constructor with simple call

This case shows the situation, when the external dependency is created in the constructor and is used with a simple function call.

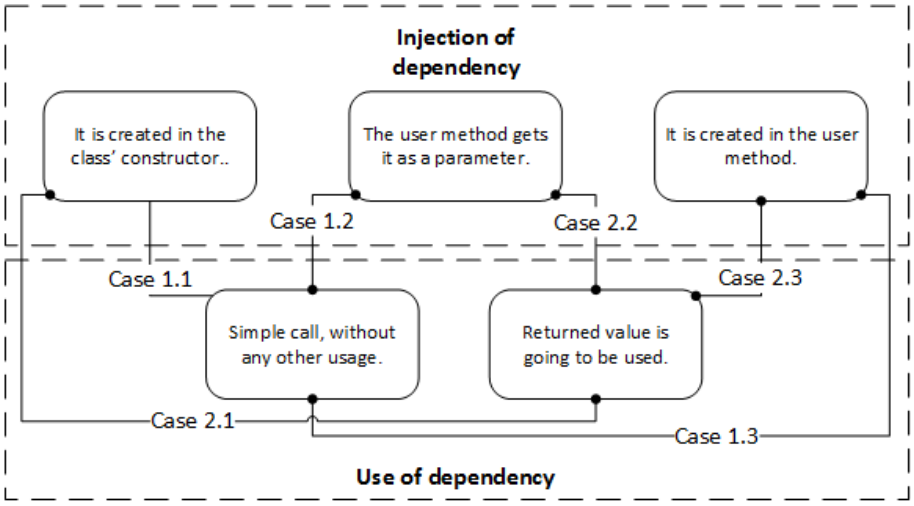

Figure 5: Cases of isolation in the SimpleExample project.

Moles As shown in Code A.1, only the constructor and the two properties (`Dummy`, `NumberDummy`) were isolated. The set method of `Dummy` has an empty body, because the received value was not used, so the implementation is irrelevant. In the set method of `NumberDummy`, the logic is the same as in the original setter. Pex used with Moles generates 4 test cases for this method. These cases cover all the paths that could be found in the method's logic, so they are a very good base of comparison. The generated test cases are depicted in Table 2.

Pex inputs						
x	y	text	number	Result	Summary	Message
false	false	null	0	"X:false, Y:false"		
true	false	null	0	null		
false	true	null	0	"X:false, Y:true"		
false	false	null	15		Exception	"This is a hidden Exception in the external library."

Table 2: Case 1.1. - Standard results with Moles

Moq We had to add a line to the project's `AssemblyInfo.cs` file to clarify that Pex should check Moq's class library (`PexInstrumentAssembly`). Without that Pex wouldn't check the dynamic proxy object, so that there wouldn't be any isolation. Code A.2 shows the statements of the code in this case. The generated test cases are the same as in Table 2.

NSubstitute It generated the same test cases as Moles or Moq. The most interesting fact was that the assembly instrumentation settings of Pex were the same with Moq. This is due to the proxy class library called Castle, which is used by both of them. This means that they were built on the same basis. Code A.3 shows the body of the test method.

Summary The results show that this case including an external object created in the constructor of the unit under test and a simple call to it can be easily isolated by the three selected frameworks.

4.2 Case 1.2: Parameter with simple call

The case includes the external dependency as a parameter and is used with a simple function call.

Moles Compared to Case 1.1, the only change is that an object with type of `B` is created and is given to the `Foo` method as a parameter, thus it's list of parameters was extended. Fortunately, this is not a problem, because we used `AllInstances` class of Moles to isolate in Case 1.1., which affects all `B` objects. This is shown in Code A.4. The generated test cases were the same as shown in Table 2.

Moq The only change in Moq's case in comparison with the previous one (1.1) was that the isolated `B` object is not getting passed to the `Boo` property, but it is getting passed as a parameter to the `Foo` method, which is shown in Code A.5. The generated test cases were also the same as in Table 2.

NSubstitute Also for NSubstitute, the only change was that object `B` is getting passed to `Foo` method instead of setting it's value to the `Boo` property. Code A.6 shows this change. In terms of results, only one test case had been generated, which means a problem. The details of this case can be found in Table 3. Pex noticed that the dynamic proxy object of NSubstitute is a testability issue, but either when instrumenting it's assembly with the `PexInstrumentAssembly` or not, Pex did not generate any more test cases.

Pex inputs						
x	y	text	number	Result	Summary	Message
false	false	null	0	"X:false, Y:false"		

Table 3: Case 1.2. - NSubstitute results

Summary These results show that Pex can have problems with the inner structure of NSubstitute in specific cases, which influences the test case generation.

4.3 Case 1.3: Inner create with simple call

In this case, the external dependent class is instantiated in the method under test, and is used by a simple function call.

Moles In this case, the logic of the Moles parameterized test method is the same as it was in Case 1.1, which could be found in Code A.1. This ensures that the generated test cases are also the same as in Table 2.

Moq In Moq's case the situation is the same as Case 1.1 of Moq, so the code can be found in Code A.2. Pex generated only three of the four expected test cases, which is due to the fact that Moq is a proxy-based isolation frameworm. Thus, it cannot reach the dependency to isolate it without using the Dependency Injection design pattern. This is the case where the CLR profiler based Moles has an advantage. In order to enable Pex to find all test cases, we have to refactor the code and use the mentioned pattern. The generated test cases can be found in Table 4.

Pex inputs						
x	y	text	number	Result	Summary	Message
false	false	null	0	"X:false, Y:false"		
true	false	null	0	null		
false	true	null	0	"X:false, Y:true"		

Table 4: Case 1.3. - Moq results

NSubstitute Also in the case of NSubstitute, the test code was the same as in Case 1.1, and results were the expected as a proxy-based framework, which can be found in Table 4.

Summary The results of this case show the limitations of the proxy-based isolation frameworks, and also show that this limitation prevents Pex to generate more test cases.

4.4 Case 2.1: Constructor with value usage

In this case the external object is instantiated in the constructor, and the returned value of the external object is used by the object under test.

Moles In the case of Moles the changes were minimal compared to Case 1.1: a bool typed variable (`value`) was introduced in the test method, which represents the value of property `Boo`. We also had to isolate the `Inverse` method, where the code of the mock implements that it returns the inversed value of the variable named `value`. With these modifications, Pex could generate all the four required test cases, which can be found in Table 2.

Moq We applied the same modifications in the test code shown in Code A.8. In terms of results, all of the four test cases had been generated, which are shown in Table 2.

NSubstitute We revealed some problems when we used Nsubstitute with Pex for this case: only two test cases were generated by Pex instead of the required four, altough the code is similar to the configuration of other tools as shown in Code A.9. The cause of this problem might be that the outcome of a computational branch depends on the inner value of the isolated object. This may affect NSubstitute and due to it's inner structure prevents Pex from checking the dynamic proxy object. The generated test cases can be found in Table 5.

Pex inputs						
x	y	text	number	Result	Summary	Message
false	false	null	0	"X:false, Y:false"		
false	true	null	0	"X:false, Y:true"		

Table 5: Case 2.1. - NSubstitute results

4.5 Case 2.2: Parameter with value usage

This case of isolation contains the situation when the object under test receives an external dependency as a parameter and it has to use the return value of this dependency.

Moles The previous test code of Moles was extended by an object `B`, which is passed to the `Foo` method as an input parameter. Code A.10 shows this modification. As it is expected, all of the four test cases were generated by Pex.

Moq Code A.11 shows the modification which was needed for this case like the modifications done in the case of Moles. The generated test cases were the expected: all four test cases were created by Pex.

NSubstitute The required modifications were also done in this case, but only one test case was generated, this was exactly the same as in Case 1.2. (Table 3).

Summary This case strengthens the hypothesis that Pex cannot check the dynamic proxy objects in some specific cases.

4.6 Case 2.3: Inner instantiation with value usage

In this case, the external dependency object is created in the method under test, and it's return value is also used there.

Moles The test code was the same as in Case 2.1. shown in Code A.7. Furthermore, the results were also the expected, i.e. all four test cases were generated, which can be found in Table 2.

Moq Similarly to Moles, when using Moq, the test code remained the same as in Code A.8. However, as it was in Case 1.3., without the usage of the Dependency Injection design pattern, it is impossible to use proxy-based frameworks effectively. This is the reason for generating only two test cases, which are summarized in Table 5.

NSubstitute This case is similar to the former case of NSubstitute, so the testing code can be seen in A.9. The limitations of proxy-based techniques are still valid, so the test cases are exactly the same as in Table 5.

Summary This case further confirmed that the inner structure of the code highly affects the effectiveness of the test case generation through the deficient capabilities of proxy-based isolation frameworks compared to the CLR profiler based tools.

5 Conclusions and future work

This section summarizes our work and results, and also gives an overview of the possible future work.

Summary The results made it clear that Moq and NSubstitute are not able to compete with Moles as a result of their proxy-based implementation. But one of the objective of this case study was to answer the research questions, so as a summary of the results, the answers are the following.

1. *Is it possible to use Pex with any isolation framework other than Moles?* Yes, it is possible, since Pex was able to collaborate with either Moq and NSubstitute at a specific level.

2. *Does Pex loose from it's functionality when using the mentioned other isolation framework?* Yes, it could lose, but it is not necessary: Moq could seamlessly collaborate with Pex, altough NSubstitute couldn't.

The results also showed that Moq and NSubstitute are based on the same class library (Castle), but they perform differently in terms of collaboration with Pex. This is most likely due to NSubstitute has an inner structure that prevents Pex to check it's dynamic proxys in some specific cases. These cases include the followings.

1. Isolated object gets passed as a parameter.
2. Property of a mock object is used.
3. Return value of a mock object is used.

Implications As an implication, we can state that Moq could be a very good replacement for Moles in a context, that has same behavior like those in this case study. In spite of this, we must also state that in such cases where proxy-based frameworks fail, Moles has to be replaced with an other CLR profiler-based framework, or the code has to be refactored.

Limitations The limitations of the case studies being introduced in this paper are the following.

- The properties, version numbers of this context could affect the results, so if it would be reexecuted in other contexts, it is not ensured that the same results would appear.
- This study is based on the SimpleExample project, which might not cover all the possible isolational cases, so the results may not be valid in other, real-world contexts.

Future work In our opinion, there's two ways to continue, extend this study. These are the following.

- Extending this comparison with other isolation frameworks that include commercial licensed ones too, because most of them are CLR profiler-based (JustMock, Isolator).
- Reexecute the whole study with newer version of Pex and the framework, which may generate other results due to the changes in the implementations.

References

[HT10] J. Halleux and N. Tillmann. "Moles: Tool-Assisted Environment Isolation with Closures". In: *Objects, Models, Components, Patterns*. Vol. 6141. LNCS. Springer, 2010, pp. 253–270. DOI: 10.1007/978-3-642-13953-6_14.

[IEE10] Institute of Electrical and Electronics Engineers. *Systems and software engineering – Vocabulary*. Standard 24765:2010. Dec. 2010, pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[Mes07] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007. ISBN: 9780132797467.

[PV09] C. S. Păsăreanu and W. Visser. "A survey of new trends in symbolic execution for software testing and analysis". In: *International journal on software tools for technology transfer* 11.4 (2009), pp. 339–353.

[TH08] N. Tillmann and J. Halleux. "Pex–White Box Test Generation for .NET". In: *Tests and Proofs*. Vol. 4966. LNCS. Springer, 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10.

[TS05] N. Tillmann and W. Schulte. "Parameterized unit tests". In: *Proc. of ESEC/FSE-13*. Lisbon, Portugal: ACM, 2005, pp. 253–262. DOI: 10.1145/1081706.1081749.

A Listings of the study's cases

```
// Arrange
// MB is the mock of class B.
MB.Constructor = (t) => { };
MB.AllInstances.DummySetBoolean = (t1,t2) => { };
MB.AllInstances.NumberDummySetInt32 = (t1, t2) =>
{
    if (t2 == 15)
    {
            throw new
            Exception("This is a hidden Exception in the external library.");
    }
};
// Act
string result = target.Foo(x, y, text, number);
```

Code A.1: Case 1.1. - Moles test method

```
// Arrange
var mock = new Mock<B>();
mock.SetupSet(m => m.Dummy = It.IsAny<bool>());
mock.SetupSet(m => m.NumberDummy = It.IsAny<int>())
    .Callback((int value) =>{ if(value==15)
    throw new Exception("This is hidden Exception in the external library.");
    });
target.Boo = mock.Object;
// Act
string result = target.Foo(x, y, text, number);
```

Code A.2: Case 1.1. - Moq test method

```
// Arrange
var mock = Substitute.For<B>();
mock.When(m => m.Dummy = Arg.Any<bool>()).Do(m => { });
mock.When(m => m.NumberDummy = Arg.Any<int>())
    .Do((value) => { if((int)value.Args()[0] == 15 )
    throw new Exception("This is a hidden Exception in the external library.");
    });
target.Boo = mock;
// Act
string result = target.Foo(x, y, text, number);
```

Code A.3: Case 1.1. - NSubstitute test method

```
// Arrange
MB.Constructor = (t) => { };
MB.AllInstances.DummySetBoolean = (t1,t2) => { };
MB.AllInstances.NumberDummySetInt32 = (t1, t2) =>
{
    if (t2 == 15)
    {
            throw new
            Exception("This is a hidden Exception in the external library.");
    }
};
B boo = new B();
// Act
string result = target.Foo(x, y, text, number, boo);
```

Code A.4: Case 1.2. - Moles test method

```
// Arrange
var mock = new Mock<B>();
mock.SetupSet(m => m.Dummy = It.IsAny<bool>());
mock.SetupSet(m => m.NumberDummy = It.IsAny<int>())
        .Callback((int value) => { if (value == 15)
        throw new
        Exception("This is a hidden Exception in the external library.");
        });
// Act
string result = target.Foo(x, y, text, number,mock.Object);
```

Code A.5: Case 1.2. - Moq test method

```
// Arrange
var mock = Substitute.For<B>();
mock.When(m => m.Dummy = Arg.Any<bool>()).Do(m => { });
mock.When(m => m.NumberDummy = Arg.Any<int>())
    .Do((value) => { if((int)value.Args()[0] == 15 )
    throw new Exception("This is a hidden Exception in the external library.");
    });
// Act
string result = target.Foo(x, y, text, number, mock);
```

Code A.6: Case 1.2. - NSubstitute test method

```
// Arrange
MB.Constructor = (t) => { };
bool value = false;
MB.AllInstances.DummySetBoolean = (t1, t2) =>
{
    value = t2;
};
MB.AllInstances.Inverse = (t1) => { return !value; };
MB.AllInstances.NumberDummySetInt32 = (t1, t2) =>
{
    if (t2 == 15)
    {
            throw new
            Exception("This is a hidden Exception in the external library.");
    }
};
// Act
string result = target.Foo(x, y, text, number);
```

Code A.7: Case 2.1. - Moles test method

```
// Arrange
var mock = new Mock<B>();
bool boolValue = false;
mock.SetupSet(m => m.Dummy = It.IsAny<bool>())
        .Callback((bool b) => { boolValue = b; });
mock.SetupSet(m => m.NumberDummy = It.IsAny<int>())
        .Callback((int value) => { if (value == 15)
        throw new
        Exception("This is a hidden Exception in the external library.");
        });
mock.Setup(m => m.Inverse()).Returns(() => { return !boolValue; });
target.Boo = mock.Object;
// Act
string result = target.Foo(x, y, text, number);
```

Code A.8: Case 2.1. - Moq test method

```
// Arrange
var mock = Substitute.For<B>();
bool boolValue = false;
mock.When(m => m.Dummy = Arg.Any<bool>())
        .Do(m => { boolValue = (bool)m.Args()[0]; });
mock.When(m => m.NumberDummy = Arg.Any<int>())
        .Do((value) => { if ((int)value.Args()[0] == 15)
        throw new
        Exception("This is a hidden Exception in the external library.");
        });
mock.Inverse().Returns(!boolValue);
// Act
string result = target.Foo(x, y, text, number);
```

Code A.9: Case 2.1. - NSubstitute test method

```
// Arrange
MB.Constructor = (t) => { };
bool value = false;
MB.AllInstances.DummySetBoolean = (t1, t2) =>
{
    value = t2;
};
MB.AllInstances.Inverse = (t1) => { return !value; };
MB.AllInstances.NumberDummySetInt32 = (t1, t2) =>
{
    if (t2 == 15)
    {
            throw new
            Exception("This is a hidden Exception in the external library.");
    }
};
B boo = new B();
// Act
string result = target.Foo(x, y, text, number, boo);
```

Code A.10: Case 2.2. - Moles test method

```
// Arrange
var mock = new Mock<B>();
bool boolValue = true;
mock.SetupSet(m => m.Dummy = It.IsAny<bool>())
        .Callback((bool b) => { boolValue = b; });
mock.SetupSet(m => m.NumberDummy = It.IsAny<int>())
        .Callback((int value) => { if (value == 15)
        throw new
        Exception("This is a hidden Exception in the external library.");
        });
mock.Setup(m => m.Inverse()).Returns(() => { return !boolValue; });
// Act
string result = target.Foo(x, y, text, number, mock.Object);
```

Code A.11: Case 2.2. - Moq test method

```
// Arrange
var mock = Substitute.For<B>();
bool boolValue = true;
mock.When(m => m.Dummy = Arg.Any<bool>())
        .Do(m => { boolValue = (bool)m.Args()[0]; });
mock.When(m => m.NumberDummy = Arg.Any<int>())
        .Do((value) => { if ((int)value.Args()[0] == 15)
        throw new
        Exception("This is a hidden Exception in the external library.");
        });
mock.Inverse().Returns(!boolValue);
// Act
string result = target.Foo(x, y, text, number, mock);
```

Code A.12: Case 2.2. - NSubstitute test method

B Isolation frameworks comparison

	Type	Profiler-based				Proxy-based			
Basic data	**Name**	**Moles**	**Fakes Framework**	**JustMock**	**Isolator**	**Rhino Mocks**	**NMock3**	**Moq**	**NSubstitute**
	Developer	Microsoft	Microsoft	Telerik	Typemock	Hibernating Rhinos	CodePlex open project	Google Code open project	GitHub open project
	Version	0.94.51023.0	-	Q1 2013	7.3.0	3.6	3.3	4.0.108247	1.4.3.0
	Licence	MSR-LA	VS 2012 Ultimate	Commercial	Commercial	Open Source (BSD)	Open Source (Apache)	Open Source (BSD)	Open Source (BSD)
Isolation capabilities	**Interfaces**	•	•	•	•	•	•	•	•
	Non-virtual methods	•	•	•	•				
	Static members	•	•	•	•				
	Sealed members	•	•	•	•				
	Private members	•	•	•	•				
	Abstract classes	•	•	•	•	•	•	•	•
	Events	•	•	•	•	•	•	•	•
	Internal members	•	•	•	•	•	•	•	•
	Assertion statements			•	•	•	•	•	•
Subjective characteristics	**Syntax**	Little bit complex	Little harder to understand	Simple, intuitive	Clean and easy to learn	Quickly learnable and understandable	Complicated at first sight	Fastly learnable, simple	Clean, easy to understand
	Developer support	No longer	Q&A with answers of developers	24h customer support	Forum	E-mail, phone	Forum	Mailing list, forum	Forum
	Community	Forum	Q&A, videos	Developer forum	Forum	Mailing list	Forum	Blogs, forum	Forum
	Documentation	Guides, videos, examples, articles	MSDN guide	Guides, videos, FAQ, KB	Guide, videos, case studys	Examples, videos	FAQ, notes	Wiki	Guide, forum
	Forward-looking index	2/10	7/10	8/10	7/10	7/10	4/10	7/10	8/10
	Comment	Proxy-based mode is supported. No longer developed.	Proxy-based mode is supported. Successor of Moles.	-	-	-	Developing is slow and lack of documentation.	One of the most popular free frameworks.	-

Figure 6: Comparison of currently available and popular .NET isolation frameworks.