Private Accessor

Author: Alex Shindich. Also see the *Acknowledgments* section.

Intent

This pattern offers a method for creating unit testable code while retaining proper encapsulation of implementation class' data and methods (*Inaccessible Members* hereafter). The pattern is applicable to most environments that offer run-time introspection. MS .NET and Java are examples of such environments.

Motivation

In the world of Test Driven Development all coding starts with a unit test. Typically, unit test classes are kept separate from the actual implementation classes. In fact, the implementation and the test methods are often separated by module boundaries. In order for a test method of an outside module to gain access to a private/protected/internal method (*Inaccessible Method* hereafter) of an implementation module, one sometimes chooses to break the encapsulation and expose the *Inaccessible Method* as public. A similar issue exists with accessing or modifying private/protected/internal variables/properties (*Inaccessible Data* hereafter) of the implementation class.

Frequently the need to test a private method arises when implementing a call-back method for a GUI event. Let's consider the following example:

```
private void InitializeComponent()
{
    …
    this.mButton.Click += new System.EventHandler(this.OnClick);
    …
}

private void OnClick(object sender, EventArgs e)
{
    …
}
```

`OnClick` callback method is declared as private, as it is only called in response to a button click event and there is no need to call it from anywhere else in the application. To test this *Inaccessible Method* one would have to break method's encapsulation and to promote its accessibility to either a public or an internal level.

Protected utility methods offer another example of *Inaccessible Methods* that need to be tested. In .NET 2.0 this problem is frequently solved by promoting protected methods to the internal

Copyright © 2007, Alex Shindich
Permission is granted to copy for the PLoP 2007 conference. All other rights reserved.

level of visibility (in Java a method with a package scope of visibility) and adding the test assembly to the list of "friendly" assemblies that have access to the internals of the implementation assembly. In .NET 1.1 and 1.0 one had to either declare utility methods as public or mark them as internal and keep the test code inside the implementation assembly.

.NET 1.1 example:

```
...
public void MyUtilityMethod ()
{
...
}
```

.NET 2.0 example:

```
...
[assembly: InternalsVisibleTo("MyTest")]
...
internal void MyUtilityMethod ()
{
...
}
```

NOTE: The *Applicability* section discusses how Java's package-level methods can be tested with the use of parallel directory hierarchies.

The purpose of this paper is to demonstrate that through the use of runtime introspection (Reflection) it is possible to unit test *Inaccessible Methods* and to access and manipulate *Inaccessible Data* without breaking the proper encapsulation.

Solution

The solution involves a utility class called *Private Accessor* that uses introspection to gain access to *Inaccessible Members* of the class being tested. The test code calls the *CallMethod* method of the *PrivateAccessor* class and provides the following as parameters for the call: a reference to the instance of the implementation class, a string with the name of the implementation method and a list of arguments.

```
bool result = PrivateAccessor.CallMethod (testObj, "PrivateMethod", param1, param2);
```

For static methods, a reference to the type of the implementation class is provided instead of the object reference.

```
bool result = PrivateAccessor.CallMethod (typeof(ImplementationClass),
"StaticPrivateMethod", param1, param2);
```

Fig. 1 depicts a class diagram and Fig. 2 depicts a sequence diagram for this solution.

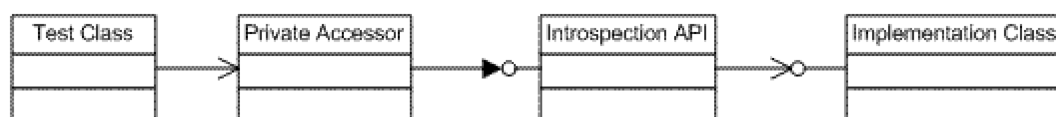

Fig. 1

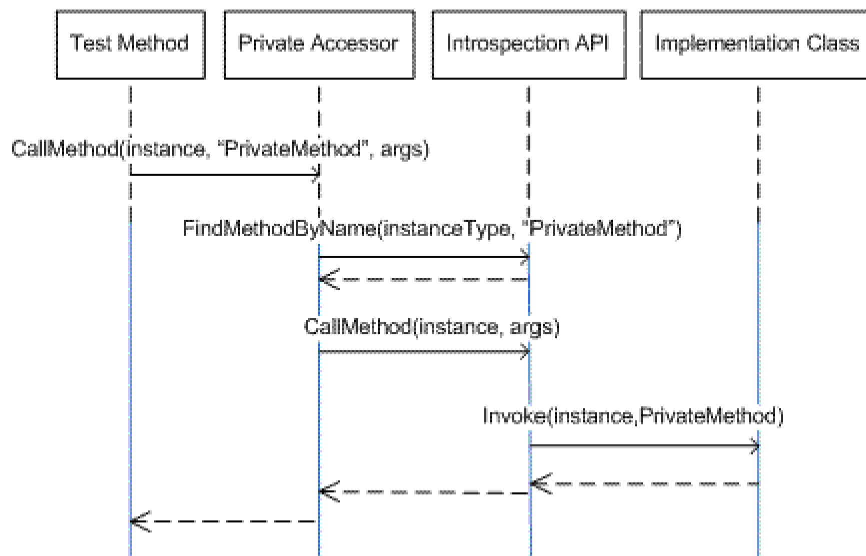

Fig. 2

Applicability

DOs:

Use the *Private Accessor* pattern every time you feel compelled to break the data/method encapsulation of the implementation class for the sake of unit testing. As mentioned in the *Motivation* section, this pattern is frequently used to test private callbacks for such events as a button click or an application exit. This pattern is also good for testing inaccessible utility methods.

When it comes to accessing/modifying *Inaccessible Data*, consider using this pattern when you are tempted to create a public accessor or a setter for a private data member for the sole purpose of being able to manipulate its value from within a unit test. Please note that the use of the *PrivateAccessor* is only warranted if the data members being manipulated are highly unlikely to change in the future. This pattern is frequently used to assign test implementations of singletons, as the name and type of the variable holding a singleton reference are not very likely to change. The code in the example below assigns a reference to an instance of the `FakeMenuManager` class to a private static variable `MenuManager.sMenuManagerInstance` that holds a pointer to a singleton instance of the `MenuManager` class.

Example:

```
[TestFixtureSetUp]
public void TestFixtureSetUp()
{
    PrivateAccessor.SetField(typeof(MenuManager),
                             "sMenuManagerInstance",
                             new FakeMenuManager ());
}
```

Additionally, the *Private Accessor* pattern can be used to call private constructors or to implement external factories.

DON'Ts:

Avoid using this pattern when it is possible to test an *Inaccessible Method* without breaking its encapsulation. This is frequently the case when testing methods with internal/package-level scope of visibility. For example, in Java it is possible to keep the test code in the same package as that of the implementation class, but in a parallel directory structure. This way test classes have access to methods with package level of visibility, but test directories are not deployed in production. [(1)] Microsoft® .NET framework 2.0 allows accessing internal methods of an implementation assembly by explicitly declaring that internals are visible to the test assembly.

Example: `[assembly: InternalsVisibleTo("MyTest")]`

If your tests excessively manipulate *Inaccessible Data* members of implementation classes, then you may often find yourself fixing broken unit tests when re-factoring. Consider abandoning the use of this pattern if your code becomes hard to refactor.

Example: PrivateAccessor.SetField(obj, "mPrivateField1", newValue);

Keep in mind that this pattern makes it more difficult to use tools provided by IDEs, such as refactoring or reference lookup tools, as these tools cannot deduce that methods/data being refactored are referenced via strings. [(1)]

Example: PrivateAccessor.CallMethod(obj, "InaccessibleMethod", param1, param2);

Structure

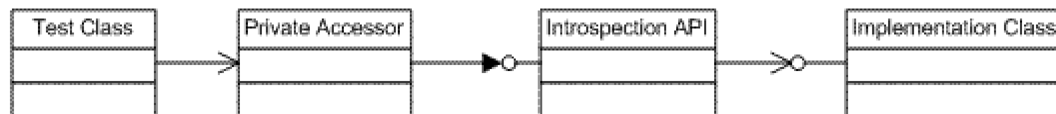

Fig. 1

Participants

- *Test Class* - contains the unit testing code
- *Private Accessor* - a reusable class that provides the ability to access *Inaccessible Members* of another class via introspection(Reflection)
- *Introspection API* - a library provided by the runtime vendor that offers introspection capability. Java provides *java.lang.reflection* and .NET provides *System.Reflection*.
- *Implementation Class* - a class that contains the code that needs to be tested
- *Inaccessible Members* – a method/property/variable that is inaccessible to the code in the test module due to encapsulation (private/protected/internal)

---

(1) The ideas for these points were partially taken from *Testing Private Methods with JUnit and SuiteRunner* article, by Bill Venners. See *References* section for more details.

Collaborations

The *Test Class* calls methods of the *Private Accessor* to gain access to the *Inaccessible Members*. *Inaccessible Members* of the *Implementation Class* are identified by strings containing their names.

Example:

```
PrivateAccessor.SetField(obj, "mField1", val);
```

The *PrivateAccessor* class utilizes the *Introspection API* to look up the binding for the *Inaccessible Member* by name. The *Private Accessor* then uses the binding information to access/invoke the *Inaccessible Member*.

Example:

```
fieldInfo = type.GetField(fieldName,
                          BindingFlags.Public |
                          BindingFlags.NonPublic |
                          BindingFlags.Instance);
return fieldInfo.GetValue(instance);
```

Consequences

On the positive side, *Private Accessor* allows preserving the proper encapsulation of the implementation code while still properly unit testing it. The pattern removes the need for creating setter methods/properties that are only applicable for unit testing, as it is possible to get/set values of *Inaccessible Data* directly for testing purposes.

On the negative side, the use of this pattern leads to more verbose test code that is a little harder to read. Manipulation of *Inaccessible Data* through the *PrivateAccessor* may lead to unnecessary entanglement of the test and implementation code.

Implementation

It is important to implement the `PrivateAccessor` in an exception-transparent manner, i.e. `PrivateAccessor` should re-throw inner exceptions. Failure to do so may result in the loss of the call stack and error message of the inner exception.

When manipulating *Inaccessible Data*, keep in mind that in .NET has multiple static scopes for an assembly: one per application domain, and one per loading context. This affects the use of the `PrivateAccessor` class and not the actual implementation of it.

A sophisticated implementation of the `PrivateAccessor.CallMethod` should search for methods using the parameter specification to properly support method overloading. The sample code below does not do so.

You may consider permitting to specify both the type and the name of the *Inaccessible Member*, to properly deal with the cases when a base class' *Inaccessible Member* is being overshadowed by a derived class' *Inaccessible Member* with the same name.

Sample Code

Implementation: The following is a sample C# implementation of *PrivateAccessor*.**CallMethod** that can invoke a private instance method:

```
using System;
using System.Reflection;
namespace PrivateAccessor
{
    public static class PrivateAccessor
    {
        public static object CallMethod(object instance, string methodName,
                                        params object[] parameters)
        {
            if (instance == null)
            {
                throw new ArgumentNullException("instance");
            }
            if (methodName == null)
            {
                throw new ArgumentNullException("methodName");
            }

            MethodInfo methodInfo = null;
            for (Type type = instance.GetType();
                 type != null && methodInfo == null;
                 type = type.BaseType)
            {
                methodInfo = type.GetMethod(methodName,
                                            BindingFlags.Instance |
                                            BindingFlags.Public |
                                            BindingFlags.NonPublic |
                                            BindingFlags.DeclaredOnly);
            }

            if (methodInfo == null)
            {
                throw new MissingMethodException(instance.GetType ().Name,
                                                 methodName);
            }

            try
            {
                return methodInfo.Invoke(instance, parameters);
            }
            catch (TargetInvocationException trgEx)
            {
                throw trgEx.InnerException;
            }

            return null;
        }
    }
}
```

Example 1: The following is a real life example for the use of *PrivateAccessor.CallMethod*:

The callback method **OnShutdown** resides in the implementation assembly:

```
public class ExternalRequest
{
    protected ExternalRequestManager()
    {
        …
        mExitEvent += OnShutdown;
    }

    private void OnShutdown()
    {
        if (WindowManager.InvokeRequired)
        {
            WindowManager.Invoke(new CommandTypeDelegate(OnShutdown));
            return;
        }

        ExitApplication();
    }
   …
}
```

The test class resides in the test assembly:

```
[TestFixture]
public class ExternalRequestManagerTest
{
    FakeWindowManager mWinMan;
    [SetUp]
    public void SetUp()
    {
        mWinMan = new FakeWindowManager();
    }

    [TearDown]
    public void TearDown()
    {
        mWinMan.Dispose();
    }

    [Test]
    public void OnShutdown_InvokeNotRequiredTest()
    {
        mWinMan.mInvokeRequired = false;
        FakeTritonEntry entry = new FakeTritonEntry();

        PrivateAccessor.CallMethod(mReqMan, "OnShutdown", entry);

        Assert.IsTrue(mReqMan.mExitApplicationCalled);
    }
   …
}
```

Example 2: The following is a real life example for the use of *PrivateAccessor*.**GetField** and *PrivateAccessor*.**SetField** methods:

The following is a partial definition for a pseudo-static class called MenuManager:

```
public class WindowManager
{
    private static WindowManager sWindowManager = new WindowManager();
    public static bool InvokeRequired
    {
        get { return sWindowManager.InvokeRequiredImpl; }
    }

    protected virtual bool InvokeRequiredImpl
    {
        get { return mMainForm.InvokeRequired; }
    }

    public static void Invoke(Delegate method, params object[] args)
    {
        sWindowManager.InvokeImpl(method, args);
    }

    protected virtual void InvokeImpl(Delegate method, params object[] args)
    {
        mMainForm.Invoke(method, args);
    }
    …
}
```

The following is a "Fake" helper class that is used in unit tests as a replacement for the real WindowManager:

```
public class FakeWindowManager : WindowManager, IDisposable
{
    public bool mInvokeRequired = false;
    public bool mInvokeCalled = false;
    public object[] mArgs;

    private WindowManager mOriginalWindowManager;
    public FakeWindowManager()
    {
        mOriginalWindowManager =
            (WindowManager)PrivateAccessor.GetField(typeof(WindowManager),
                                                    "sWindowManager");
        PrivateAccessor.SetField(typeof(WindowManager), "sWindowManager", this);
    }

    public void Dispose()
    {
        PrivateAccessor.SetField(typeof(WindowManager),
                                 "sWindowManager", mOriginalWindowManager);
    }

    protected override bool InvokeRequiredImpl
    {
        get { return mInvokeRequired; }
    }
    protected override void InvokeImpl(Delegate method, object[] args)
    {
        mArgs = args;
        mInvokeCalled = true;
    }
}
```

Known Uses

As seen from some of the articles listed in the *References* section, this pattern describes a solution that is an industry-wide accepted practice. For instance, Visual Studio Codename Orcas offers a class called *PrivateObject* (*Microsoft.VisualStudio.TestTools.UnitTesting.PrivateObject*) that provides unrestricted access to *Inaccessible Members*.

Related Patterns

I consider this pattern related to the practice of Type or Object Mocking.

References

- NMock Web Site
  [http://nmock.org/]
- How To: Unit Test Hidden Classes, by Nick Berardi
  [http://coderjournal.com/?p=38]
- How to Test Private and Protected methods in .NET, by Tim Stall
  [http://www.codeproject.com/csharp/TestNonPublicMembers.asp]
- Unit testing with .NET, by Kevin Jones
  [http://www.itarchitect.co.uk/articles/display.asp?id=171]
- Testing Private Methods with JUnit and SuiteRunner, by Bill Venners
  [http://www.artima.com/suiterunner/private.html]

Acknowledgments

The .NET Private Accessor class was developed based on the work done by Wesley Steiner. Adam Adorjan and Dmitry Gritsai developed the Private Accessor class and Alex Shindich added code for accessing static methods, properties and data members. Please note that similar solutions, as evident from the *References* section, have been independently developed by many others both for .NET and for Java.

Special thanks go to my PloP shepherd Jason Yip for his help during the revision of this paper.