.NET Unit Testing with Accessor Classes and Reflection

Jason Curl

Contents

Chapter 1

Abstract

There are many best practices and philosophies for unit testing of components in .NET FX (desktop versions 2.0 and later), with tools such as xUnit, nUnit and MSTest among others, supported by Visual Studio, Mono IDE and extensions in NuGet. This article provides examples and "how to" for testing using nUnit and reflection technologies using the RJCP.CodeQuality library.

Chapter 2

Reflection and Accessor Classes

Reflection in .NET is a runtime type technology for discovering and instantiating types, possibly in other assemblies and possibly having compiler inaccessible permissions. By using reflection, one can instantiate a normally inaccessible target type, discover properties and methods, invoke those property and methods. This makes reflection highly suitable (if not fast) for testing internal classes that are in external assemblies. In rare cases, this may also allow for testing of internal state of classes for white box testing.

This work was initially inspired by Home-made Private Accessor for Visual Studio 2012+ and expanded upon for more complex use cases.

2.1 Testing non-Public Implementations

The `internal` keyword of C# is special - it makes a clear indication that functionality is expected to be available for use from within the assembly, but that functionality may not meet the (stricter) requirements for exposing to the outside world. A `private` method or property within an assembly makes a clear intention that no other class be able to access the functionality, even if it is part of the same assembly.

The Accessor model is not intended to replace a good design through the use of interfaces and mocks. While it is possible to access private methods and fields using the Accessor model, one should consider first if the design is correct that this should be done. Often through the use of interfaces and contracts, or abstract classes, can one achieve a better design where no private methods are required for testing.

2.1.1 Test Driven Design

Some champions of test driven design make strong arguments that unit testing should only be performed on public API. If an API is not publicly visible to the outside world, it is not a candidate for testing. Rewriting classes should be testable via the public API and doesn't require any working test cases to be modified.

The other perspective is that software should be broken down into its simplest components using the SOLID principle. Typical engineering practices also suggest breaking work down into its simplest components, that would require testing of those simpler components which are built upon for more complex components. The author of this document sees it is much more pragmatic to test simple classes that may be internal to an assembly and ensuring the correctness of building blocks, which also simplify the implementation of test cases for more complex components by making assumptions that the simpler components work. The internal types should specify public methods which are then tested and made public via Accessor classes.

For example, testing the insertion and removal of an object in a collection is simple. If a more

complex class were to use the collection and the collection is not directly exposed, the number of conditions required to test the more complex class may be significantly higher than being able to test both classes directly. Some safety standards, such as ASIL also take this approach.

2.1.2 Fault Injection

High availability software may also require testing via fault injection. In this special case, using an Accessor class may make it easier to inject specific faults and ensure that software can recover from these faults.

2.2 Accessor Classes

Accessor Classes use functionality provided by the Microsoft Visual Studio `PrivateType` and `PrivateObject` implementations. Such functionality was provided in Visual Studio 2010, where an input class was provided and an Accessor class was generated making methods publicly available. It was removed in versions after Visual Studio 2010 with other solutions as documented by Microsoft.

The Accessor implementation uses reflection to write wrappers for internal objects under test that exist in another assembly.

Some of the benefits of the Accessor model are:

- Type Safety.
- Protection against typographical errors by avoiding strings.
- Easier to handle when a rename is performed.
- Encapsulate and hide the use of `PrivateObject`. The implementation of `PrivateObject` and `PrivateType` has changed between VS2010-2015 and VS2017 that they are no longer 100% compatible. The Accessor classes hide this and provide our own `PrivateObject` compatible for VS2017 that also runs under .NET 4.0.

Some disadvantages are:

- No automatic code generators exist. Accessor classes have to be written by hand.
- It uses reflection, so can be slow.
- Exceptions in private classes are difficult to analyze due to missing stack traces across reflection boundaries.

2.2.1 Functionality Supported by Accessors

The following functionality can be implemented using Accessor Classes:

- Can test instantiable methods of an instantiable class, or static methods of classes;
- Type Parameters can be provided for generic classes (either public or internal)
- Generic methods (those with Type Parameters);
- Nested types, either static or instantiable;
- Return accessor classes;
- Accept accessor classes as parameter inputs;
- Internal enumerations;
- Internal delegates;

2.2.2 Functionality Not Supported by Accessors

It is not possible to test:

- Internal interfaces.

2.3 InternalsVisibleTo

Multiple languages offer features for enabling internal visibility to specific components. In C++ this is the friend keyword. In C# there is the `internal` keyword which makes types and methods public to code within the assembly. The `InternalsVisibleTo` attribute of the assembly can list other assemblies that should see internal methods as public.

While there are many answers to testing by using the `InternalsVisibleTo` attribute, the author of this article advises strongly against its usage on two grounds:

- Architecturally wrong dependencies: `InternalsVisibleTo` reverses dependencies, so a component is now dependent on a "client", which in this case the test assembly is the client.
- It complicates significantly the work required for strong naming components, an important technology and a requirement for installation of assemblies in the GAC.
  - Cyclic dependencies are now introduced, a test library depends on an assembly that is strongly signed, the assembly must however reference a strongly named test assembly

2.4 Delegates

Internal delegates can be tested using reflection, a feature that the publicize.exe tool from Microsoft cannot do. With the use of lambdas in C# 3.0, it is much easier to create delegates via reflection and execute delegates from private code.

Chapter 3

Basic Structure

3.1 Instantiable Classes

Classes that need to be instantiated, i.e. those that are non-static and whose instance is created with the `new` operator, must have an Accessor class that derives from `AccessorBase`.

The `AccessorBase` class can't be instantiated on its own as it's abstract. The Accessor class provides the base class with the details required so it can be instantiated. There are six constructors:

- `AccessorBase(PrivateObject pObject)`: Wrap the Accessor class around an already instantiated object `pObject`.
- `AccessorBase(PrivateType pType, params object[] params)`: Instantiate a new instance of an Accessor class of the type given by `pType`. The optional parameters are used to determine how to construct the object. The `pType` object usually describes the assembly and the type within that assembly which is reflected upon.
- `AccessorBase(PrivateType pType, Type[] parameterTypes, object[] args)`: Instantiate a new instance of an Accessor class of the type given by `pType`. The array of `parameterTypes` specifies concisely the types for the constructor to instantiate, with `args` forming the inputs to the constructor.
- `AccessorBase(string assemblyName, string typeName, params object[] args)`: Instantiates the object given the assembly name and the type name as strings. It is otherwise identical to using the constructor with the `PrivateType` with the `assemblyName` and `typeName`.
- `AccessorBase(string assemblyName, string typeName, Type[] parameterTypes, object[] args)`: Instantiates the object given the assembly name and the type name as strings. It is otherwise identical to using the constructor with the `PrivateType` with the `assemblyName` and `typeName`.
- `AccessorBase(string assemblyName, string typeName, Type[] parameterTypes, object[] args, Type[] typeArguments)`: Instantiates an instance of a generic class, specifying the type arguments in the generic class with the array of `typeArguments`. It is otherwise identical to using the constructor with the `PrivateType` with the `assemblyName`, `typeName` and the `typeArguments`.

See the next section on best practices. It is recommended to use the version of the constructor that takes the `PrivateType` as an input. As complexity grows, it is likely you will need this reference to the `PrivateType` for other methods.

3.2 Static Classes

Static Accessor classes are exposing non-public types and methods. The `AccessorBase` class provides `static` methods to implement static Accessor classes. In all cases, one needs to define the PrivateType and pass this to the static methods.

- `object InvokeStatic(PrivateType type, string methodName, params object[] args);`
- `object InvokeStatic(PrivateType type, string methodName, Type[] paramTypes, object[] args);`
- `object InvokeStatic(PrivateType type, string methodName, Type[] paramTypes, object[] args, Type[] typeArguments);`
- `object GetStaticFieldOrProperty(PrivateType type, string name);`
- `void SetStaticFieldOrProperty(PrivateType type, string name, object value);`

Work was done to try and simplify the usage of the static methods by trying to automatically identify the Accessor class (derived by AccessorBase) to get the `PrivateType` as the object is constructed. Such a solution relies on using stack frames to try and determine the type containing the static method, which is ultimately unsuccessful due to compiler optimizations such as inlining. To overcome inlining, an explicit attribute was required for every method to disable inlining which makes code harder to read. Thus it was decided that a static Accessor should instantiate a static `PrivateType` for the non-public type and allowing code to be more portable to multiple compilers.

Chapter 4

Accessor Best Practices

4.1 Naming Accessor Classes

All types (except enumerations, `enum` type) should have the word `Accessor` appended to it, to make it obvious this is an Accessor class.

4.2 Naming Methods and Properties

The names of methods and properties should have the same name of the methods and properties in the original class.

4.3 Use the "nameof" Operator

The methods and properties being tested in the Accessor class should have the same name as the methods and properties in the class being tested. Therefore, when providing the name of the method or property being tested to the base class, use the `nameof` operator. This allows easier renaming of method names and letting the IDE do the refactoring saving work and errors.

Chapter 5

Writing Accessor Classes Cook Book

The following sections provide common patterns and examples on how an Accessor implementation can be written to test the code. Let's make the following assumptions in all the following examples:

- The namespace for the code being tested and the test class are identical: `RJCP.Assembly`.
- The name of the assembly DLL being tested is `Assembly.dll` and the test assembly is `AssemblyTest.dll`.
- The using statements are assumed and occur before the namespace.

Please note, that the usage of the `PrivateType` and `PrivateObject` classes are instantiated from `RJCP.CodeQuality` and are not from MSTest, due to subtle differences between the different versions available for MS Test Classes.

5.1 Non-Public Classes

The simplest use case is to test a single non-public class where all types are public. It is simple, has a single method and a single property.

```
namespace RJCP.Assembly {
  internal class MyClass {
    public MyClass(int initialValue) {
      Property = initialValue;
    }

    public int Property { get; set; }

    public void DoSomething() {
      Console.WriteLine("{0}", Property);
    }
  }
}
```

The Accessor class would be in the test project and could be implemented as:

```
namespace RJCP.Assembly {
  public class MyClassAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyClass";
    public static readonly PrivateType AccType =
```

```
      new PrivateType(AssemblyName, TypeName);

    public MyClassAccessor(int initialValue)
      : base(AccType, initialValue)
    { }

    public int Property {
      get { return (int)GetFieldOrProperty(nameof(Property)); }
      set { SetFieldOrProperty(nameof(Property), value); }
    }

    public void DoSomething() {
      Invoke(nameof(DoSomething));
    }
  }
}
```

The constructor passes the input value and lets the reflection system guess the type that is required.

```
namespace RJCP.Assembly {
  public class MyClassAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyClass";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyClassAccessor(int initialValue)
      : base(AccType,
        new Type[] { typeof(int) },
        new object[] { initialValue })
    { }

    public int Property {
      get { return (int)GetFieldOrProperty(nameof(Property)); }
      set { SetFieldOrProperty(nameof(Property), value); }
    }

    public void DoSomething() {
      Invoke(nameof(DoSomething));
    }
  }
}
```

The second example makes instantiation explicit for which constructor should be used to give the initial value.

5.2 Non-Public Static Classes

The next simplest use case is to access public or internal static methods of a class (be the class itself static or not). The reason why it is not concerning if the class itself is static or not is because the type is the only information needed to execute a static method or property.

```
namespace RJCP.Assembly {
  internal static class MyStaticTest {
    public static int Property { get; set; }
    public static void DoSomething() {
```

```
      Console.WriteLine("{0}", Property);
    }
  }
}
```

The Accessor class could look as such:

```
namespace RJCP.Assembly {
  public static class MyStaticTestAccessor {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyStaticTest";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public static int Property {
      get {
        return (int)AccessorBase.
          GetStaticFieldOrProperty(AccType, nameof(Property));
      }
      set {
        AccessorBase.
          SetStaticFieldOrProperty(AccType, nameof(Property), value);
      }
    }

    public static void DoSomething {
      AccessorBase.InvokeStatic(AccType, nameof(DoSomething));
    }
  }
}
```

5.3 Non-Public Class with Methods having ref Struct

The type `ref struct` was introduced in .NET C# 7.2, and is an important feature in supporting `ReadOnlySpan<T>` and `Span<T>` types. A `ref struct` is an object that can only live on the stack. This presents a new challenge to reflection and testing of internal methods.

- A `ref struct` can only live on the execution stack. It cannot be static or live as a member of an existing class or `struct` as this would put the object on the heap.
- It cannot be a member of an `async` method or a lambda (see `ReadOnlyMemory<T>` and `Memory<T>` for example). This would cause the `ref struct` to be boxed.
- It cannot be used as a generic type argument, and cannot be used as the type of an array element.

It can be used as:

- A method parameter
- A return type
- A local variable

These challenges means that reflection cannot be used to provide accessor methods.

```
namespace RJCP.Assembly {
  internal class MyTest {
    public int Append(ReadOnlySpan<byte> buffer) {
      ...
```

```
    }
  }
}
```

The accessor code using Linq expressions instead of reflection to access the private methods:

```
namespace RJCP.Assembly {
  using System;
  using System.Linq.Expressions;
  using RJCP.CodeQuality;

  public class MyTestAccessor {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyTest";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyTestAccessor() : base(AccType) { }

    private delegate int AppendDelegate(ReadOnlySpan<byte> buffer);
    public int Append(ReadOnlySpan<byte> buffer) {
      var instance = Expression.Constant(PrivateTargetObject);
      var method = AccType.ReferencedType.
        GetMethod(nameof(Append), new Type[] { typeof(ReadOnlySpan<byte>) });
      var parameter = Expression.
        Parameter(typeof(ReadOnlySpan<byte>), nameof(buffer));
      var call = Expression.Call(instance, method, parameter);
      var expression = Expression.Lambda<AppendDelegate>(call, parameter);
      var func = expression.Compile();
      return func(buffer);
    }
  }
}
```

5.4 Non-Public Class with return type of ref struct

Similar to a `ref struct` as a method parameter, a `ref struct` may be returned by a method, but cannot be referenced using reflection.

```
namespace RJCP.Assembly {
  internal class MyTest {
    public ReadOnlySpan<byte> GetBuffer() {
      ...
    }
  }
}
```

The accessor code using Linq expressions to return the `ref struct` type:

```
namespace RJCP.Assembly {
  using System;
  using System.Linq.Expressions;
  using RJCP.CodeQuality;

  public class MyTestAccessor {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyTest";
```

```
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyTestAccessor() : base(AccType) { }

    private delegate ReadOnlySpan<byte> GetBufferDelegate();
    public ReadOnlySpan<byte> GetBuffer() {
      var instance = Expression.Constant(PrivateTargetObject);
      var method = AccType.ReferencedType.
        GetMethod(nameof(GetBuffer), Type.EmptyTypes);
      var call = Expression.Call(instance, method, Array.Empty<Expression>());
      var expression = Expression.
        Lambda<GetBufferDelegate>(call, Array.Empty<ParameterExpression>());
      var func = expression.Compile();
      return func();
    }
  }
}
```

The example above can be used as a basis for more complex accessors by using Linq expressions.

5.5 Non-Public Class with Static Methods

Note now what happens if the class is not static, but the method is static:

```
namespace RJCP.Assembly {
  internal class MyTest {
    public MyTest(int initialValue) {
      MyTest.Property = initialValue;
    }

    public static int Property { get; set; }
    public static void DoSomething() {
      Console.WriteLine("{0}", Property);
    }
  }
}
```

The original test code for testing the static methods remain. We can also add additional code for the constructor.

```
namespace RJCP.Assembly {
  public class MyTestAccessor {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyTest";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyTestAccessor(int initialValue)
      : base(AccType,
         new Type[] { typeof(int) },
         new object[] { initialValue })
    { }

    public static int Property {
      get {
        return (int)AccessorBase.
```

```
          GetStaticFieldOrProperty(AccType, nameof(Property));
      }
      set {
        AccessorBase.
          SetStaticFieldOrProperty(AccType, nameof(Property), value);
      }
    }

    public static void DoSomething {
      AccessorBase.InvokeStatic(AccType, nameof(DoSomething));
    }
  }
}
```

Now it is apparent the reason for using the constructor that takes the `PrivateType` as its input. The `PrivateType` is used for construction, as well as for the static methods. Using `AccType` to start makes it easier as your classes grow in complexity.

5.6 Derived Classes

Non-public types may derive from other non-public types.

```
namespace RJCP.Assembly {
  internal class MyBase {
    public MyBase(int value) {
      Value = value;
    }

    public int Value { get; set; }

    public virtual void DoSomething() {
      Console.WriteLine("{0}", Value);
    }
  }

  internal class MyDerived : MyBase {
    public string Description { get; set; }

    public MyDerived(int value, string description) : base(value) {
      Description = description;
    }

    public override void DoSomething() {
      Console.WriteLine("{0}: {1}", Description, Value);
    }
  }
}
```

One possible implementation of the Accessor classes is:

```
namespace RJCP.Assembly {
  public class MyBaseAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyBase";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);
```

```
  protected MyBaseAccessor(PrivateType pType,
    Type[] parameterTypes, object[] args)
    : base(pType, parameterTypes, args)
  { }

  public MyBaseAccessor(PrivateObject pObj)
    : base(pObj)
  { }

  public MyBaseAccessor(int value)
    : base(AccType,
      new Type[] { typeof(int) },
      new object[] { value })
  { }

  public int Value {
    get { return (int)GetFieldOrProperty(nameof(Value)); }
    set { SetFieldOrProperty(nameof(Value), value); }
  }

  public virtual void DoSomething() {
    Invoke(nameof(DoSomething));
  }
}

public class MyDerivedAccessor : MyBaseAccessor {
  private const string AssemblyName = "Assembly";
  private const string TypeName = "RJCP.Assembly.MyDerived";
  public static readonly PrivateType AccType =
    new PrivateType(AssemblyName, TypeName);

  protected MyDerivedAccessor(
    PrivateType pType, Type[] parameterTypes, object[] args)
    : base(pType, parameterTypes, args)
  { }

  public MyDerivedAccessor(PrivateObject pObj)
    : base(pObj)
  { }

  public MyDerivedAccessor(int value, string description)
    : base(AccType,
      new Type[] { typeof(int), typeof(string) },
      new object[] { value, description })
  { }

  public string Description {
    get { return (string)GetFieldOrProperty(nameof(Description)); }
    set { SetFieldOrProperty(nameof(Description), value); }
  }

  public override void DoSomething() {
    Invoke(nameof(DoSomething));
  }
}
```

```
]
```

There are new constructors (protected) that support derived types and their instantiation:

- The base class derives from `AccessorBase`. Derived classes derive from the base class instead of `AccessorBase`.
- Both `MyBaseAccessor` and `MyDerivedAccessor` have a protected method that takes the `PrivateType`, the `parameterTypes` and `args`.
- `MyDerivedAccessor` uses the protected constructor from `MyBaseAccessor` which in turn instantiates via the `AccessorBase`.

Thus instantiating `MyDerivedAccessor` creates an object of the correct type, as it uses `MyDerivedAccessor.AccType`. It is no longer necessary to make any of the base methods virtual in the base class, or to override methods in the derived classes - the correct method will always be called as per rules of object inheritance.

5.6.1 Type Conversions

Building a class hierarchy can be very convenient with test code, but one must be very careful. Typecasting between Accessor classes has no effect. To understand why, is to understand that no Accessor classes have state, it is all maintained in the `AccessorBase` class in the `PrivateType` and `PrivateObject`, and so typecasting is simply exchanging one `AccessorBase` with another `AccessorBase` proving to be a no-operation. The only difference is to change the available methods dependent on the Accessor class being used.

Instead, to typecast one Accessor class to another requires instantiation of a new Accessor class with the `PrivateObject` from the original class with the `PrivateType` of the new Accessor class. Note in the implementations of `MyBaseAccessor` and `MyDerivedAccessor` have each a constructor taking a `PrivateObject` as an input. This is important to allow for the type casting.

- Performing an explicit type cast: Create a new Accessor object and provide the type to be casted to. This example is equivalent to `(MyBase)derived`. An explicit type cast is required if behaviour of a object may change if type casted, such as if derived class used the `new` operator.

  ```
  MyDerivedAccessor derived = FactoryAccessor.Create();
  MyBaseAccessor bclass = new MyBaseAccessor(
    new PrivateObject(derived.PrivateTargetObject, MyBaseAccessor.AccType));
  ```

  Even though the underlying type of the object is `MyDerived`, the `MyBaseAccessor` is being instantiated and told that the object should be handled as the `MyBase` type. This is how an explicit type cast works.

- Simply upcasting an object to a derived class: You may not want to typecast the object as such, but simply "transfer" the object being tested to a more correct Accessor class. Such an example occurs in the factory model, where the factory create method creates the derived class but returns it in a base factory. Then your test code simply wants to upcast, but doesn't need to change the type. This is only required if there is functionality in the class that only accessible from the derived class.

  ```
  MyBaseAccessor bclass = FactoryAccessor.Create();
  MyDerivedAccessor derived = new MyDerivedAccessor(
    new PrivateObject(bclass.PrivateTargetObject));
  ```

  Internally, the .NET type system knows the object being tested is of type `MyDerived`. But the implementation of `FactoryAccessor.Create()` maps this to `MyBaseAccessor`. Without the type cast, calls to the base class still execute functionality in the derived class as would normally be done in object oriented programming.

It is important to realize that the type cast of Accessor classes effectively does nothing as the Invokes to the underlying methods do not change as the internal types do not change. It is not possible to implement an `explicit` operator using .NET for classes that are derived from one another.

Your Accessor classes can also use standard .NET code to check if one object is assignable from another object, so you can determine at run time which Accessor class to instantiate based on the object. Use the Type.IsAssignableFrom methods.

```
public MyBaseAccessor Create() {
  object obj = Invoke(nameof(Create));
  if (MyDerivedAccessor.AccType.ReferencedType.IsAssignableFrom(obj.GetType()))
    return new MyDerivedAccesor(obj);
  return new MyBaseAccessor(obj);
}
```

5.7 Generic Classes with Public Type Arguments

The AccessorBase makes it significantly easier to implement generic classes. A non-public class may be implemented as:

```
namespace RJCP.Assembly {
  internal class GenericClass<T> {
    public void Push(T item) { ... }
    public T Pop() { ... }
  }
}
```

Writing the generic Accessor is similar to earlier examples, except now a list of all the type arguments are provided in the constructor.

```
namespace RJCP.Assembly {
  public class GenericClassAccessor<T> : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.GenericClass`1";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName, new Type[] { typeof(T) });

    public GenericClassAccessor()
      : base(AccType,
        new Type[] { },         // Constructor signature to use.
        new object[] { })       // Parameters
    { }

    public void Push(T item) {
      Invoke(nameof(Push), item);
    }

    public T Pop() {
      return (T)Invoke(nameof(Pop));
    }
  }
}
```

The name of the type uses the back tick (`` `1 ``) describing the number of type arguments in the type. The default constructor in the Accessor must be defined to properly instantiate the class under test.

5.7.1 Generic Classes with Non-Public Type Arguments

One can easily see in this example why type arguments are limited to public types.

In the example previously, a function returns an object of the type argument T. If this is a public type, it is easy to test, the return type is public and can be directly used in code as the type is known to the compiler and .NET without reflection. If it is required to test a class with generic types, and the type argument must be an non-public type, it is recommended to create a specific Accessor class for that non-public type and not to expose the type arguments via the Accessor class at all.

```
namespace RJCP.Assembly {
  public class GenericClassPrivateTypeAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.GenericClass`1";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName,
        new Type[] { OtherAccessor.AccType.ReferencedType });

    public GenericClassPrivateTypeAccessor()
      : base(AccType, new Type[] { }, new object[] { })
    { }

    public void Push(OtherAccessor item) {
      Invoke(nameof(Push), item.PrivateTargetObject);
    }

    public OtherAccessor Pop() {
      obj result = Invoke(nameof(Pop));
      return new OtherAccessor(new PrivateObject(result));
    }
  }
}
```

5.8 Nested Types

Nested types occur commonly within the .NET framework. The following examples show how to test static methods in nested classes (whether those classes themselves are static or not).

5.8.1 Simplest Nested Type

The simplest kind of nested type is not using generics. Let's have the following code:

```
namespace RJCP.Assembly {
  internal static class NestedStaticTypes {
    internal static class NestedStaticType {
      public static int NestedMethod() {
        return 42;
      }
    }
  }
}
```

The equivalent nested classes for the two types mentioned would be written as:

```
namespace RJCP.Assembly {
  public static class NestedStaticTypesAccessor {
    private const string AssemblyName = "Assembly";
```

```
    private const string TypeName = "RJCP.Assembly.NestedStaticTypes";
    private static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public static class NestedStaticTypeAccessor {
      private static readonly PrivateType AccType =
        NestedStaticTypesAccessor.AccType.GetNestedType("NestedStaticType");

      public static int NestedMethod() {
        return (int)AccessorBase.InvokeStatic(AccType, nameof(NestedMethod));
      }
    }
  }
}
```

The top level class is defined with a `PrivateType` object. The nested classes reference the top level class `PrivateType` and request a new private type for the nested class with the function `GetNestedType`. One can repeat the pattern to have further nested classes.

5.8.2 Generic Nested Class

In the case that the inner nested class uses generics, but the outer class doesn't, it is expected that only the inner nested class needs to provide details for the type arguments.

```
namespace RJCP.Assembly {
  internal static class NestedStaticGTypes {
    internal static class NestedStaticGType<T> {
      public static string Name() {
        return typeof(T).ToString();
      }
    }
  }
}
```

The Accessor classes would be written as:

```
namespace RJCP.Assembly {
  public static class NestedStaticGTypesAccessor {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.NestedStaticGTypes";
    private static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public static class NestedStaticGTypeAccessor<T> {
      private static readonly PrivateType AccType =
        NestedStaticGTypesAccessor.AccType.
          GetNestedType("NestedStaticGType`1", new Type[] { typeof(T) });

      public static string Name() {
        return (string)AccessorBase.InvokeStatic(AccType, nameof(Name));
      }
    }
  }
}
```

5.8.3 Top Level Generic Type with Nested Class

If the outer class uses generics, the inner class also needs to be instantiated with the type argument, even if in C# it isn't explicitly mentioned.

```
namespace RJCP.Assembly {  xcv bn
  internal static class NestedStaticGTypes<T> {
    internal static class NestedStaticGType {
      public static string Name() {
        return typeof(T).ToString();
      }
    }
  }
}
```

The Accessor classes would be written as:

```
namespace RJCP.Assembly {
  public static class NestedStaticGTypesAccessor<T> {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.NestedStaticGTypes`1";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName, new Type[] { typeof(T) });

    public static class NestedStaticGTypeAccessor {
      public static readonly PrivateType AccType =
        NestedStaticGTypesAccessor<T>.AccType.
          GetNestedType("NestedStaticGType", new Type[] { typeof(T) });

      public static string Name() {
        return (string)AccessorBase.InvokeStatic(AccType, nameof(Name));
      }
    }
  }
}
```

One should note how the nested class also needs to be instantiated with the parameter type for the parent class.

5.8.4 Generic Method in Nested Class

This example is not really specific on showing how to access a generic method, the same example as for the non-generic classes apply, except now `AccessorBase.InvokeStatic(PrivateType, string, Type[], Object[], Type[])` allows parameter types for generics.

```
namespace RJCP.Assembly {
  internal static class NestedStaticGTypes {
    internal static class NestedStaticGType {
      public static string Name<T>() {
        return typeof(T).ToString();
      }
    }
  }
}
```

The Accessor code is therefore:

```
namespace RJCP.Assembly {
  internal static class NestedStaticGTypesAccessor {
```

```
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.NestedStaticGTypes";
    private static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    internal static class NestedStaticGTypeAccessor {
      private static readonly PrivateType AccType =
        NestedStaticGTypesAccessor.AccType.
          GetNestedType("NestedStaticGType");

      public static string Name<T>() {
        return (string)AccessorBase.InvokeStatic(AccType, nameof(Name),
          new Type[] { },
          new object[] { },
          new Type[] { typeof(T) } );
      }
    }
  }
}
```

The parameters `parameterTypes` and `arguments` must be provided and may not be `null`.

5.8.5 Nested Classes all with Generic Types

The most general example for types that are generic are provided:

```
namespace RJCP.Assembly {
  internal class NestedStaticGTypes<T> {
    internal class NestedStaticGType<U> {
      public static string Name<V>() {
        return String.Format("{0}+{1}+{2}",
          typeof(T), typeof(U), typeof(V));
      }
    }
  }
}
```

The Accessor classes would be written as thus:

```
namespace RJCP.Assembly {
  public class NestedStaticGTypesAccessor<T> {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "Namespace.NestedStaticGTypes`1";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName, new Type[] { typeof(T) });

    public class NestedStaticGTypeAccessor<U> {
      private static readonly PrivateType AccType =
        NestedStaticGTypesAccessor<T>.AccType.
          GetNestedType("NestedStaticGType`1",
            new Type[] { typeof(T), typeof(U) });

      public static string Name<V>() {
        return (string)AccessorBase.InvokeStatic(AccType, nameof(Name),
          new Type[] { }, new object[] { }, new Type[] { typeof(V) });
      }
    }
```

```
  }
}
```

Interesting in this example is that the nested type shows it has one parameter type with the back tick notation of `NestedStaticGType`1`, but it itself requires two generic type arguments.

5.9 Instantiations (objects) of Nested Types

The next logical step from Nested Types is to be able to instantiate objects of nested types and test the methods of those instances.

5.9.1 Nested Types

The simplest case to test for are non-generic classes:

```
namespace RJCP.Assembly {
  internal class NestedTypes {
    public int MethodA() { return 42; }

    internal class NestedType {
      public int MethodB() { return 64; }
    }
  }
}
```

In the case above, both classes have a default constructor. The Accessor classes would therefore be:

```
namespace RJCP.Assembly {
  public class NestedTypesAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.NestedTypes";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public NestedTypesAccessor() : base(AccType) { }

    public int MethodA() {
      return (int)Invoke(nameof(MethodA));
    }

    public class NestedTypeAccessor : AccessorBase {
      public static readonly PrivateType AccType =
        NestedTypesAccessor.AccType.GetNestedType("NestedType");

      public NestedTypeAccessor() : base(AccType) { }

      public int MethodB() {
        return (int)Invoke(nameof(MethodB));
      }
    }
  }
}
```

In this example, we see a more universal pattern for `PrivateType` and `PrivateObject` forming, where a class is beginning to support static and non-static methods, with or without nested classes.

5.9.2 Nested Types with Generics

The next example covers the case that the parent and nested classes uses generics.

```
namespace RJCP.Assembly {
  internal class NestedGTypes<T> {
    private T m_Value;
    public NestedGTypes(T initialValue) { m_Value = initialValue; }
    public string Value() { return m_Value.ToString(); }

    internal class NestedGType<U> {
      private U m_Value;
      public NestedGType(U initialValue) { m_Value = initialValue; }
      public string ValueNested() { return m_Value.ToString(); }
    }
  }
}
```

The Accessor code now creates the `PrivateType` defining the generics and uses that to instantiate the base `AccessorBase` class to create instances of the specific generic types. You'll note in the example code, we don't need to specify the generics details to the `AccessorBase` as this is encapsulated by the `PrivateType`.

```
namespace RJCP.Assembly {
public class NestedGTypesAccessor<T> : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.NestedGTypes`1";
    private static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName, new Type[] { typeof(T) });

    public NestedGTypesAccessor(T initialValue)
      : base(AccType,
        new Type[] { typeof(T) }, new object[] { initialValue })
    { }

    public string Value() { return (string)Invoke(nameof(Value)); }

    public class NestedGTypeAccessor<U> : AccessorBase {
      private static readonly PrivateType AccType =
        NestedGTypesAccessor<T>.AccType.
          GetNestedType("NestedGType`1",
            new Type[] { typeof(T), typeof(U) });

      public NestedGTypeAccessor(U initialValue)
        : base(AccType,
          new Type[] { typeof(U) }, new object[] { initialValue })
      { }

      public string ValueNested() {
        return (string)Invoke(nameof(ValueNested));
      }
    }
  }
}
```

5.10 Non-Public Type Return Values

The factory pattern often results in one class creating instances of other classes. The `AccessorBase` can be used to test private instances of the factory pattern. Take the simplest factory pattern:

```
namespace RJCP.Assembly {
  internal class RelatedClassTest {
    public RelatedClassTest(int initialValue) { Value = initialValue; }
    public int Value { get; private set; }
  }

  internal class RelatedClassTestFactory {
    public RelatedClassTestFactory() { }
    public RelatedClassTest Create() {
      return new RelatedClassTest(42);
    }
  }
}
```

The simplest case here is that the factory object creates a specific instance having a value of 42. The Accessor versions of the two classes would be:

```
namespace RJCP.Assembly {
  public class RelatedClassTestAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.RelatedClassTest";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public RelatedClassTestAccessor(PrivateObject obj)
      : base(obj)
    { }

    public RelatedClassTestAccessor(int initialValue)
      : base(AccType,
        new Type[] { typeof(int) }, new object[] { initialValue }) { }

    public int Value {
      get {
        return (int)GetFieldOrProperty(nameof(Value));
      }
    }
  }

  public class RelatedClassTestFactoryAccessor: AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.RelatedClassTestFactory";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public RelatedClassTestFactoryAccessor() : base(AccType) { }

    public RelatedClassTestAccessor Create() {
      object obj = Invoke(nameof(Create));
      return obj ?? new RelatedClassTestAccessor(new PrivateObject(obj));
    }
  }
```

```
}
```

The `RelatedClassTestAccessor` requires a constructor that takes a `PrivateObject` instance that can be used to provide a way to access the methods of the object. It is not expected that a non-public class have a constructor taking a type of `PrivateObject`.

The `RelatedClassTestFactoryAccessor` returns an object type, which is then used to wrap around the equivalent object accessor class `RelatedClassTestAccessor`. If the return value is `null`, then `null` is also returned by the accessor, else the wrapped object.

5.11 Non-Public Type Input Parameters

A slightly more complex case is if private types are used as part of signatures for other private types. Such situations might occur with private collections of private objects.

```
namespace RJCP.Assembly {
  internal class RelatedItemClass {
    public RelatedItemClass(string value) { Value = value; }
    public string Value { get; private set; }
  }

  internal class RelatedCollectionClass {
    private HashSet<string> m_Set = new HashSet<string>();

    public void Add(RelatedItemClass item) {
      if (m_Set.Contains(item.Value)) {
        throw new ArgumentException("Item already in collection", nameof(item));
      }
      m_Set.Add(item.Value);
    }

    public bool IsInCollection(string value) { return m_Set.Contains(value); }
  }
}
```

In this example, the `RelatedItemClass` is a private type and is used with the private collection type `RelatedCollectionClass` for maintaining if an object has already been added to a collection or not. The equivalent Accessor classes would look like:

```
namespace RJCP.Assembly {
  public class RelatedItemClassAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.RelatedItemClass";
    public static PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public RelatedItemClassAccessor(string value)
      : base(AccType, new Type[] { typeof(string) }, new object[] { value })
    { }

    public string Value {
      get {
        return (string)GetFieldOrProperty(nameof(Value));
      }
    }
  }
```

```
  public class RelatedCollectionClassAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.RelatedCollectionClass";
    public static PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public RelatedCollectionClassAccessor() : base(AccType) { }

    public void Add(RelatedItemClassAccessor item) {
      Invoke(nameof(Add),
        new Type[] { RelatedItemClassAccessor.AccType.ReferencedType },
        new object[] { item.PrivateTargetObject });
    }

    public bool IsInCollection(string value) {
      return (bool)Invoke(nameof(IsInCollection), value);
    }
  }
}
```

The `RelatedItemClassAccessor` has to provide a reference to the actual private type so that `RelatedCollectionClassAccessor.Add` can call the private type method with the correct type (not the accessor type). When invoking, it retrieves the actual referenced type and the actual underlying object via `PrivateTargetObject`.

As the `AccessorBase` must maintain a reference to the underlying object, but expose this for cases such as `Invoke` when it's needed, that a class that is under test may not have a property with the name `PrivateTargetObject`. The name of the property is such to reduce the likelihood of a name conflict for the class being tested.

5.12 Input Parameters Reference and Out Types

Handling of reference types and output types with Accessor classes is no different to how it's done with reflection.

5.12.1 Out Types

Let's say the non-public implementation looks as such:

```
namespace RJCP.Assembly {
  internal MyClass {
    public void OutMethod(out int value) {
      value = 42;
    }
  }
}
```

The corresponding Accessor method would be:

```
namespace RJCP.Assembly {
  public MyClassAccessor : AccessorBase {
    private const string Assembly = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyClass";
    public static readonly PrivateType AccType =
      new PrivateType(Assembly, TypeName);

    public MyClassAccessor() : base(AccType) { }
```

```
    public void OutMethod(out int value) {
      int internalValue = 0;
      object[] args = new object[] { internalValue };
      Invoke(nameof(OutMethod),
        new Type[] { typeof(int).MakeByRefType() },
        args);
      value = args[0];
    }
  }
}
```

For an `out` type, one needs to make a temporary variable which will be converted to the reference type. This allocates space on the argument stack. The non-public implementation then overwrites the value of the argument, which on return from the `Invoke` is read and placed into the Accessor methods `out` variable.

5.13 Non-Public Enumerations

Enumerations appears to be a special case for testing, that they are types, but not classes. To use non-public enumerations, make a copy of the enumeration in the Accessor code that is public. A second class is requiired that encapsulates the type of the non-public enumeration.

```
namespace RJCP.Assembly {
  internal enum MyValues {
    None = 0,
    One,
    Two
  }

  internal class MyClass {
    public void DoSomething(MyValues value) { ... }
  }
}
```

A copy of the enum and an Accessor class to hold the type would be then:

```
namespace RJCP.Assembly {
  public enum MyValues {
    None = 0,
    One,
    Two
  }

  public static class MyValuesAccessor {
    private const string Assembly = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyValues";
    public static readonly PrivateType AccType =
      new PrivateType(Assembly, TypeName);
  }

  public class MyClassAccessor : AccessorBase {
    private const string Assembly = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyClass";
    public static readonly PrivateType AccType =
      new PrivateType(Assembly, TypeName);
```

```
    public MyClassAccessor() : base(AccType) { }

    public void DoSomething(MyValues value) {
      Invoke(nameof(DoSomething),
        new Type[] { MyValuesAccessor.AccType.ReferencedType },
        new object[] { value });
    }
  }
}
```

The enumeration itself derives from the system type `System.Int32`. In inputs, the runtime system will do the typecast from the test `enum` to the non-public `enum`. Returning an `enum` (not shown) would perform an explicit typecast to the test `enum`.

5.14 Non-Public Events with Public Event Handlers

Events can be passed to base classes by using the `add` and `remove` keywords. Let the code to be tested look something like:

```
namespace RJCP.Assembly {
  internal class MyEventClass {
    public event EventHandler<EventArgs> MyEvent;

    protected virtual void OnMyEvent(EventArgs args) {
      EventHandler<EventArgs> handler = MyEvent;
      if (myEvent != null) handler(this, args);
    }

    public void RaiseEvent(int value) {
      if (value == 42) OnMyEvent(new EventArgs());
    }
  }
}
```

To implement an Accessor class allowing the event handler to be tested:

```
namespace RJCP.Assembly {
  public class MyEventHandlerAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "MyEventClass";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyEventHandlerAccessor() : base(AccType) { }

    public event EventHandler<EventArgs> MyEvent {
      add { AddEventHandler(nameof(MyEvent), value); }
      remove { RemoveEventHandler(nameof(MyEvent), value); }
    }

    public virtual void OnMyEvent(EventArgs args) {
      Invoke(nameof(OnMyEvent), args);
    }

    public void RaiseEvent(int value) {
      Invoke(nameof(RaiseEvent), value);
```

```
    }
  }
}
```

It is not required to provide an accessor method for `OnMyEvent` as this is a private implementation feature of the class being tested. It is not possible to test that `OnMyEvent` will be called by raising the call to `RaiseEvent` (as it is not possible to create a public class for testing that derives from the non-public class).

5.15 Non-Public Events with Private Event Handlers

The information is based on Microsoft Documentation - Hook up a Delegate using Reflection. One can expand on the examples to produce a generic implementation with Events and the associated private delegates.

```
namespace RJCP.Assembly {
  internal class MyPrivateEventArgs : EventArgs {
    public MyPrivateEventArgs(int value) {
      Value = value;
    }

    public int Value { get; private set; }
  }

  internal class MyEventClass {
    public event EventHandler<MyPrivateEventArgs> MyPrivateEvent;

    protected virtual void OnMyEvent(MyPrivateEventArgs args) {
      EventHandler<MyPrivateEventArgs> handler = MyPrivateEvent;
      if (handler != null) handler(this, args);
    }

    public void RaiseEvent(int value) {
      OnMyEvent(new MyPrivateEventArgs(value));
    }
  }
}
```

The creation of the Accessor `EventArgs` object is very special. C# does not allow to inherit from two different classes, so it is not possible to inherit from both `EventArgs` and `AccessorBase` - a decision needs to be made. Hence, the Accessor `EventArgs` object therefore derives from `EventArgs` so that it can be used as the type parameter to `EventHandler<>`. Inside the Accessor `EventArgs` is a private implementation of the real accessor.

```
namespace RJCP.Assembly {
  public class MyPrivateEventArgsAccessor : EventArgs {
    private class Accessor : AccessorBase {
      public Accessor(PrivateObject obj) : base(obj) { }

      public int Value {
        get {
          return (int)GetFieldOrProperty(nameof(Value));
        }
      }
    }
```

```
    private Accessor m_Accessor;

    public MyPrivateEventArgsAccessor(PrivateObject obj) {
      m_Accessor = new Accessor(obj);
    }

    public int Value { get { return m_Accessor.Value; } }
  }
}
```

```
namespace RJCP.Assembly {
  public class MyEventClassAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyEventClass";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyEventClassAccessor() : base(AccType) { }

    public event EventHandler<MyPrivateEventArgsAccessor> MyPrivateEvent {
      add {
        EventHandler<MyPrivateEventArgsAccessor> handler = value;
        AccessorEventHandler ieh = (s, a) => {
          handler(s, new MyPrivateEventArgsAccessor(new PrivateObject(a)));
        };
        AddIndirectEventHandler(nameof(MyPrivateEvent), value, ieh);
      }
      remove {
        RemoveIndirectEventHandler(nameof(MyPrivateEvent), value);
      }
    }

    public void DoWork(int value) {
      Invoke(nameof(DoWork), value);
    }
  }
}
```

In studying the implementation, `MyPrivateEventArgsAccessor` derives from `EventArgs` and has an internal Accessor that does the work. `MyPrivateEventArgsAccessor` only has a single constructor taking a `PrivateObject`, as it is expected this object is only instantiated from the code generating the event (the code under test).

The registration of the event is done through a lambda expression that is responsible for taking the private `MyPrivateEventArgs` from the code being tested and wrapping it in a `MyPrivateEventArgsAccessor`. Thus the test code for the event looks very much similar to the original code.

As the delegate `ieh` is what is really associated with the event hander in the code under test, it has to be "remembered" when removing the event. One cannot simply recreate the lambda and expect it to be removed. Thus, the usage of `AddIndirectEventHandler` which takes the name of the event and the delegate the user provided to map to the lambda created. So in `RemoveIndirectEventHandler`, the name of the event and the user delegate can be used to get back the delegate trampoline created in `add`.

The delegate `AccessorEventHandler` is a convenience that has a compatible signature (.NET 2.0 and later) with event handlers in private code. So long as the delegate for an event is derived from `EventHandler` or `EventHandler<T>` where `T` is an `EventArgs`, one can use the

`AccessorEventHandler`.

The test code could look like:

```
[Test]
public void RaisePrivateEvent() {
  EventClassAccessor accessor = new EventClassAccessor();

  int count = 0;
  EventHandler<MyPrivateEventArgsAccessor> handler = (s, e) => {
    count += e.Value;
  };

  accessor.MyPrivateEvent += handler;
  accessor.DoWork(45);
  Assert.That(count, Is.EqualTo(45));

  accessor.MyPrivateEvent -= handler;
  accessor.DoWork(10);
  Assert.That(count, Is.EqualTo(45));
}
```

5.16 Non-Public Delegates

Probably one of the most complex cases of testing internal types are delegates (events are covered separately). Implementing the use case of non-public delegates is possible if done carefully.

The example provided here is for the case that a non-public delegate is passed to a non-public method. The non-public delegate will be executed at a later time by other logic in the code. This is a different use case to events (and would commonly be found with non-public P/Invoke implementations)

```
namespace RJCP.Assembly {
  internal class Fd { ... }

  internal enum FdMonEvents { ... }

  internal class FdMon {
    public delegate void FdEvent(FdMon sender, Fd fd,
      FdMonEvents occurred, object userData);

    public void RegisterEvent(Fd fd, FdMonEvents events,
      FdEvent callback, object userData) {
      ...
    }
  }
}
```

The usage of lambdas to act as trampolines significantly simplifies the implementation.

```
namespace RJCP.Assembly {
  public class FdMonAccessor : AccessorBase {
    private const string Assembly = "Assembly";
    private const string TypeName = "RJCP.Assembly.FdMon";
    public static readonly PrivateType AccType =
      new PrivateType(Assembly, TypeName);

    public FdMonAccessor() : base(AccType) { }
```

```
    private readonly static PrivateType FdEventType =
      AccType.GetNestedType("FdEvent");
    public delegate void FdEventAccessor(FdMonAccessor sender, FdAccessor fd,
      FdMonEvents occurred, object userData);
    private delegate void FdEventInternal(object sender, object fd,
      int occurred, object userData);

    public void RegisterEvent(FdAccessor fd, FdMonEvents events,
     FdEventAccessor callback, object userData) {
      FdEventInternal cb = (csender, cfd, coccurred, cuserData) => {
        callback(
          new FdMonAccessor(new PrivateObject(csender)),
          new FdAccessor(new PrivateObject(cfd)),
          (FdMonEvents)coccurred,
          cuserData);
        };
      Delegate d = Delegate.CreateDelegate(FdEventType.ReferencedType,
        cb.Target, cb.Method);

      Invoke(nameof(RegisterEvent),
        new Type[] {
          FdAccessor.AccType.ReferencedType,
          FdMonEventsAccessor.AccType.ReferencedType,
          FdEventType.ReferencedType,
          typeof(object)
        },
        new object[] {
          fd.PrivateTargetObject,
          events,
          d,
          userData
        });
    }
  }
}
```

In the above real world example, a temporary "trampoline" delegate is created (called `cb`) which has a signature compatible with the non-public delegate. Since .NET 2.0, a delegate matches if the types are compatible (in .NET 1.1 and 1.0, it was required that the types match exactly).

- Non-Public Original Delegate

  ```
  internal delegate void FdEvent(FdMon sender, Fd fd,
    FdMonEvents occurred, object userData)
  ```

  Both `FdMon` and `Fd` are classes and derive from object, `FdMonEvents` is an enumeration type (derived from `System.Int32`) and `object` is a public type.

- Trampoline Delegate can therefore be

  ```
  private delegate void FdEventInternal(object sender, object fd,
    int occurred, object userData)
  ```

The lambda notation significantly reduces overhead in writing code by allowing the accessor callback `FdEventAccessor callback` being used in our trampoline callback `cb`. Inside our callback there is boilerplate code which simply instantiates Accessor instances from the objects of non-public types.

The main trick is now creating the delegate which can be passed to the non-public method expecting

the non-public delegate:

```
Delegate d = Delegate.CreateDelegate(
  FdEventType.ReferencedType, cb.Target, cb.Method);
```

Because the delegate has context (it is not static), creating the delegate requires the form where the callback target is required (`cb.Target`). Not providing this will result in a type mismatch when invoking the non-public method `RegisterEvent`. The `FdEventType.ReferencedType` is the type of the delegate we're creating, which is the non-public delegate.

The lifetime of the delegate is for the lifetime for which there is a reference. The `FdEventInternal cb` lives for as long as `Delegate d` lives, which is determined by the duration for which the non-public class `FdMon` keeps a reference to `Delegate d`. Thus, special consideration is not required to pin, or keep the delegate alive. The .NET garbage collector will do the work for us.

5.17 Exceptions

Exceptions raised by the Accessor classes will be the same type as raised by code under test. Any instances of the `TargetInvocationException` are again raised with the `InnerException`. Thus no special code is required if the exception being raised is a public type.

5.17.1 Checking Non-Public Exceptions

The `Type` class is required to test type equality for raised exceptions, where those exceptions are non-public.

```
namespace RJCP.Assembly {
  using System.Runtime.Serialization;

  [Serializable()]
  internal class MyException : System.Exception {
    public MyException() : base() { }
    public MyException(string message) : base(message) { }
    public MyException(string message, System.Exception inner)
      : base(message, inner) { }
    protected MyException(SerializationInfo info, StreamingContext context)
      : base(info, context) { }
  }

  internal class ClassException {
    public void RaiseException() {
      throw new MyException("Test");
    }
  }
}
```

The following classes show the accessors and the test case to test that the exception was raised. Note that the MyExceptionAccessor isn't really intended to be instantiated (but has a constructor in case we want to wrap the raised exception in the Accessor class).

```
namespace RJCP.Assembly {
  public class MyExceptionAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.MyException";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public MyExceptionAccessor(PrivateObject obj) : base(obj) { }
```

```
  }

  public class ClassExceptionAccessor : AccessorBase {
    private const string AssemblyName = "Assembly";
    private const string TypeName = "RJCP.Assembly.ClassException";
    public static readonly PrivateType AccType =
      new PrivateType(AssemblyName, TypeName);

    public ClassExceptionAccessor() : base(AccType) { }

    public void RaiseException() {
      Invoke(nameof(RaiseException));
    }
  }
}
```

```
namespace RJCP.Assembly {
  [TestFixture]
  public class ExceptionTest {
    [Test]
    public void CheckException() {
      ClassExceptionAccessor ce = new ClassExceptionAccessor();
      Assert.That( () => { ce.RaiseException(); },
        Throws.TypeOf(MyExceptionAccessor.AccType.ReferencedType));
    }
  }
}
```

The test code uses the `PrivateType` referenced type to perform a comparison of exactly the correct exception that is raised.

Chapter 6

PrivateType and PrivateObject

The `PrivateType` and `PrivateObject` are important for the implementation of Accessor classes. The RJCP.CodeQuality project has provided a copy of the classes for portability, with portions decompiled from the Microsoft implementations. This provides the following benefits:

- The `PrivateType`/`PrivateObject` API has changed from VS2015 to VS2017, now .NET 4.0 can use VS2017 semantics
  - VS2015 would sometimes raise the `TargetInvocationException` raised by the Reflection system in .NET, sometimes it would catch the exception and raise the inner exception - the one raised by your code.
    - The RJCP.CodeQuality project is compatible with the VS2017 implementation and raises always `TargetInvocationException`, the AccessorBase implementation always raises the inner exception to make it easy to exchange test code against real classes with test code against Accessor classes.
  - VS2015 can target .NET 4.0 and earlier. VS2017 uses a different assembly and can target .NET 4.5 and later.
    - The RJCP.CodeQuality project targets .NET 4.0 for the highest level of compatibility.
- Extension of Functionality:
  - Generic types are now supported by providing `Type` parameters
  - Instantiation of Nested Types, which may also be generic

Chapter 7

Further Work

7.1 Type Casting System

The work to convert objects from one type to another as described in the section Type Conversions is significant and can be repetitive. One could investigate a system to register `PrivateType` objects with `AccessorBase` objects that can be used to:

- Instantiate an Accessor class of the correct type given an object of a non-public type. One can check the type against a `PrivateObject` and if they match, then instantiate the Accessor class (with `CreateInstance` from reflection). This can assist with the factory use case to create the correct type object. Examples where this is useful may be:
  - Implementation of code for non-public return types (Non-Public Type Return Values).
  - Implementation of trampoline callbacks (Non-Public Delegates) to instantiate the correct member from derived Accessor classes (Derived Classes).
- Perform explicit type conversions. Such a static helper class to instantiate new Accessor classes of types from other Accessor classes can automatically check if the type conversion is valid by checking the object inheritance tree of the Accessor classes, also allowing for programmatic reduction of errors in test code.

7.2 Non-Public Interfaces

The usage of interfaces occurs often when implementing against SOLID principles. One common model is to define interfaces for classes and use dependency injection to specify behaviour of classes. The usage of dependency injection and interfaces makes it theoretically simpler to test behaviour, for example by abstracting behaviour by implementing a test class against an interface and injecting that into a class under test. It should be investigated how it might be possible for a test class to be implemented against a non-public interface to inject into a non-public class for testing.