BrianHetrick.com
Application Note AN-4
Unit Testing Inherited Interfaces

Goal

Object oriented languages encourage inheritance. Inheritance, by design, creates classes which share substantial subsets of behavior. Unit testing these classes requires substantial subsets of duplicated test activity testing the shared behavior. This gives rise to copy and paste unit tests. This in turn increases maintenance risk and effort.

The goal of this Application Note is to describe techniques for unit testing classes which inherit common behavior or interfaces from parent classes or interface declarations without replicating test code.

Introduction

Inheritance can isolate and implement behavior common to a category of classes. When a class implements a property or method, all classes inheriting from that class automatically have that property or method. Unless the inheriting class replaces it, the parent class's implementation of the property or method is available through the inheriting class.

Inheriting classes can replace an inherited property or method in one of two ways: by hiding it or by overriding it. In C#, hiding an inherited property or method requires the redefinition use the `new` keyword, and overriding an inherited property or method requires that the original definition use the `virtual` keyword and the redefinition use the `override` keyword. Java does not permit hiding inherited methods, and methods which are not defined using the `final` keyword are virtual or overrides.

Inherited properties and methods are part of an object's public interface. Unit testing best practice calls for testing these inherited properties and methods in all classes having them. This effectively requires duplicating the base class's unit tests in the unit tests for each class inheriting from the base class. A common but incorrect approach to avoiding this duplication limits tests in a unit test to properties and methods introduced or replaced in the class under test. The behavior inherited from a base class, it is thought, will be tested in the base class's unit test.

This reasoning is incorrect. Inheritance involves more than the object's public interface. Inheriting from a class permits access to its elements declared `protected` in addition to its elements declared `public`. This permits inheriting classes to change the object state relied upon by inherited code. This in turn affects the results the inherited properties and methods produce, even without redefining them. Nor is this a problem, a mistake, or a defect in the language: this is an intended and very powerful aspect of inheritance. Therefore, depending on the base class's unit test to automatically validate classes inheriting from the base class is simply insufficient.

Furthermore, it might not be possible to unit test the base class. Unit tests by definition exercise code, and code is exercised by executing it. Most object oriented programming languages provide facilities for writing code which is not intended to be executed as-is, and in fact cannot be executed as-is. These facilities include abstract classes, generic (templated) classes, and interfaces. These code constructs are intentionally skeletal and incomplete: they are intended to be completed by other code which inherits from the constructs. In these cases, then, there is no base class unit test to depend upon.

A simple solution to this problem is for the unit tests to form an inheritance hierarchy similar to that formed by the tested classes. Unit tests for derived classes would thus inherit tests from the unit tests for the parent classes. With suitable design, the inherited unit testing functionality can ensure the derived object sat-

isfies the contracts of its parent classes. This approach can be used even if the parent class is abstract, generic, or an interface.

This Application Note explores techniques to avoid duplication of effort in constructing unit tests for classes arranged in inheritance hierarchies. It includes techniques to test classes inheriting both from concrete classes and from non-executable code constructs. The discussion in this Application Note uses C# and NUnit for examples. The principles discussed are of course more widely applicable.

Unit Testing

Unit Testing Definition

Microsoft describes unit testing this way:

> The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.[1]

This Application Note adopts the definition that unit testing as a software verification technique in which the smallest independent software artifacts are exercised in isolation to ensure they satisfy expectations. All of these components of the definition are important.

Unit testing is a *verification* technique. Software quality measurement has two aspects: validation, ensuring the correct software is built; and verification, ensuring the software is built correctly. Achievement along one of these dimensions gives little information regarding achievement along the other. Unit testing does not address the correctness of the translation from market opportunities into requirements, or from requirements into specification, or from specification into design. Instead, it addresses only the correctness of the translation from design into implementation.

Unit testing deals with *smallest independent software artifacts*. In object oriented programming, the smallest software artifact that exists independently is the class. Unit testing in object oriented languages therefore tests classes. Similarly, unit testing in procedural languages tests individual functions or procedures.

Unit testing deals with software artifacts *in isolation*. The class under test is the only part of the deliverable software that participates in the test. The unit test provides all stimuli to the unit under test through invoking properties and methods. The unit test also provides all resources required by the unit under test. Should the unit under test create or use other objects, these other objects are ones provided by the unit test. This may require the unit under test to provide for dependency injection.

Unit testing deals with *satisfaction of expectations*. The expectations on a software artifact may be formalized as a set of assertions about the artifact. The unit test then consists of a sequence of activities that exercise the software artifact and observe that each assertion is correct. The unit test effectively specifies the software "as built;" in agile or informal software development, this is the only specification the software has.

Finally, unit testing *exercises* software, and examines its behavior. The unit test creates an environment in which the software artifact is to be executed, then executes the software artifact. The artifact's behavior results from its execution. The software artifact must therefore be executable. This has important implications to the techniques used to implement the unit tests. These are discussed later in this Application Note.

Unit Testing Goals

The goal of unit testing is not to catch bugs. Unit tests do indeed catch bugs, when first run. However, that purpose is served by running the tests once. Retaining the unit tests for future runs, and running the unit tests at every build (a typical use), does not serve to catch bugs. So what does it do?

Unit testing is a check on the development process, not on the developed code. Most importantly, unit testing documents and verifies the contract offered by a class. This allows bugs to be ascribed a location. A bug is code not executing "as expected." The unit test allows the developer to determine if the expectation is justified. If the behavior is not exercised by the unit test, the behavior is not guaranteed by the class under test, and expecting the behavior is a bug in the code using the class. If the behavior is exercised by the unit test, the behavior is guaranteed, and failing to provide the behavior is a bug in the class under test. Generally, then, if the unit tests succeed, "the bug" may be regarded as in the code using the class.

1 Microsoft Corporation. "Unit Testing," in *Developing with Visual Studio .NET*. Retrieved from http://msdn.microsoft.com/en-us/library/aa292197%28VS.71%29.aspx

Now, it may be that the class is intended to provide the depended-upon behavior, and both the class and its unit test are deficient. This commonly occurs in agile development practices, where the expected behavior of the code base changes over time. It also occurs with waterfall development practices, where the behavior of the code base is extended over time. The appropriate response is to modify the class and add tests for the now-expected behavior to the unit test. Existing tests for the class should continue to pass.

Failures of existing tests indicate an incompatible change in the contract offered by the class. Incompatible changes to one class may introduce bugs into other classes, classes that have not changed at all. Recall that unjustified expectations as to code behavior are bugs: incompatible changes by definition change the expected behavior of code. This removes the justification from code expecting the former behavior, and by definition creates a bug wherever the former behavior is expected. As with any bug, observable symptoms may exhibit far removed in location and time from the actual bug.

This bug creation effect can be partially contained by manually examining all uses of the class undergoing the incompatible change. This examination attempts to determine if the client (using) code depends on the formerly guaranteed behavior. When such dependencies are found, they must be repaired.

This containment activity is notoriously difficult, both in the sense of requiring a great deal of effort and in the sense of being error prone. Detecting the need for repair involves recreating the chain of logic that resulted in the production of the original code, and then determining if that logic is still valid. Making the repair involves generating an alternate chain of logic not involving the former behavior, and creating replacement code resulting from this alternate logic. The latter activity is essentially normal programming; the former, though, resembles debugging without the benefit of a debugger.

The purpose of unit tests is therefore to document and verify a set of assumptions product code is permitted to make about the major constituent of its environment: other product code.

All code is intended to be used in a specific context and embodies a great many assumptions which are valid in that context. The primary context of deliverable code is the remainder of the deliverable code.

Unit tests enumerate and verify the assumptions that may be made about deliverable code. Unit tests therefore require a great deal of frequently repetitious code. Much unit test code can be automatically generated or copied and pasted from templates or other unit tests. Still, unit tests typically require at least as much development effort as the deliverable code tested, and are frequently several times larger than the deliverable code by any code volume measure. The effort required by unit tests should be regarded as an investment, rather than an expense: as with most software quality processes, unit testing saves effort over the product lifetime and frequently accelerates the product development cycle.

Unit Testing Best Practices

Unit testing is far more of an art than a science. There is, however, some degree of consensus on what constitutes "best practice" in unit testing. The following points are generally agreed upon:

- Each deliverable class has a dedicated unit test class. Each unit test class exercises a single deliverable class. Automatically generated code which is identical across a set of tests is acceptable. A code artifact that exercises a behavior across multiple classes is not.

  For example, consider the method:

  ```
  public override string ToString()
  ```

  This method is inherited from the `System.Object` class, and so exists in every class. The project may require `ToString()` to produce "something useful." It is entirely possible to create a code artifact which creates an instance of every class, invokes the `ToString()` method on each, and ensures the string returned contains the name of the class. Such a code artifact might well be a useful auditing tool, but it is not a unit test. It would not replace tests of the `ToString()` method in the unit test for each class. The unit test for each specific class would test the class's implementation of `ToString()` and ensure it returned the string expected for that class.

  Similarly, no deliverable class other than the class under test is instantiated in a unit test. If the class under test requires resources, those resources must be provided by the unit test. Otherwise, the test is an integration test: quite useful in the appropriate context, but not a unit test.

- Tests are independent. Each test is responsible for the entire life cycle of the object under test. This involves creating an environment in which the object under test exists, creating the object under test, creating a state for the object under test, exercising the behavior to be tested, disposing of the object when the test is complete, and disposing of the environment in which the object under test existed. Tests do not depend on the results of other tests.

  For some objects, creating the environment may be a particularly expensive operation. For example, an object may require a populated backing database. In such a case, it may be acceptable for the test fixture (the class containing tests for the class under test) to set up and tear down an environment used by all tests in the test fixture.

  Test independence requires that tests not depend upon changes made to the environment by other tests. This can be achieved by tests restoring the environment to its initial state when changes are made as part of the test, or by reloading the contents of the environment.

- Tests test a single assertion. Each test is responsible for checking a single fact about the object under test. A typical simple property would have several associated tests: one to ensure the property accepts and retrieves a series of acceptable values, several to ensure the property rejects each of a series of unacceptable values, one to ensure the property publishes a `PropertyChanged` event when an assignment to the property value changes the property value, one to ensure the property does not publish a `PropertyChanged` event when an assignment to the property value does not change the property value, and perhaps others.

- Tests should not contain logic. Ideally, a test has no loops and no decisions. It sets up the environment and object under test, exercises a single specific behavior, and tears down the object and environment.

- Every error condition has a test. Error handling is like the fire sprinkler system in a building: it is needed only when something goes wrong, but then the need is desperate. Every condition giving rise to any exception should be exercised.

- Tests are first-class development objects. They are documented, under change control, planned into the schedule, reviewed identically to deliverable code, and required to be complete and uniformly successful before the tested class is considered complete.

- Test data sets are first-class development objects. If the product uses a database, there are test databases under change control constructed specifically for unit tests. If the product accepts a file, there are test files under change control constructed specifically for unit tests.

- Tests use literal expected values. Repeating the code being tested to compute the expected value is senseless. If the object property under test returns a value that results from a computation, the unit test should compare the return value to a literal constant (such as `3`, `"Blue"`, or `new int [] {0, 1, 3}`) rather than the result of a computation. The value of the literal, of course, should be consistent with the environment created by the unit test, or the change controlled test data used by the unit test. If necessary, the unit test should include a separate test for the test data itself.

The following points are less universally accepted, but are frequently recommended:

- Tests test essential behavior rather than accidental behavior. A serial number generator, for example, might increment a durable counter by 1 and return its new value. That sequentially returned values are observed to differ by 1 is accidental behavior. It depends on a property of the test environment: only one thread is accessing the generator. The essential behavior, expected in practice, is that sequentially returned values are observed to be increasing. The unit test should therefore be written to accept any sequence of strictly increasing numbers.

- Tests exercise only the client interface. Most classes have public properties and methods that constitute their interface to clients. This client interface is generally the only way to cause the object to do anything. Testing non-public properties and methods checks the implementation details, not the implementation correctness. Non-public methods and properties are the ultimate in accidental behavior.

- Tests exercise all public behavior. A popular saying is that simple property getters and setters are “too simple” to fail. But property getters and setters at least retrieve from and store into a variable, and the specification of variable might be erroneous. Setters generally have other behavior as well: rejecting `null` values, rejecting out of range values, rejecting

empty or over length strings, rejecting strings of an incorrect format, publishing `PropertyChanged` events, and so forth. Finally, the simplicity of a property is an implementation detail and hence an accidental behavior: the next revision of the object may replace the simple property with one that does substantial computation. No code is too simple to fail.

Constructing Inheritable Unit Tests

There are several key requirements to successfully creating a unit test class hierarchy. This section describes these requirements.

Virtual Object Creation Function

Unit test classes contain a number of tests. Best practice calls for these tests to be independent: the tester should be able to run any subset of tests, in any order. Each test is therefore responsible for any per-test setup and tear down, and each test fixture (in NUnit, a test class) is responsible for any per-fixture setup and tear down. Unit tests therefore create instances of the class under test.

Suppose class `B` specializes class `A`. The unit test `TestB` should therefore specialize unit test `TestA`. The methods of unit test `TestA` must create an instance of `A` to test; however, the methods of `TestB`, including the inherited methods of `TestA`, must create an instance of `B` to test. The use of class `B` must somehow be injected into the existing tests in `TestA`.

A simple way to do this is to delegate test object creation to a single virtual function. The root of the unit test class hierarchy should have a function:

```
protected virtual object NewObject()
{
    return new object ();
}
```

Each unit test must override this function to create an instance of the appropriate class under test. For example, the unit test for class `A` would contain:

```
protected override object NewObject()
{
    return new A ();
}
```

while the unit test for class B would contain:

```
protected override object NewObject()
{
    return new B ();
}
```

References to the `NewObject` function will, through the virtual function mechanism, resolve to the definition of `NewObject` provided by the most specialized class. In the class `A` and `B` example, the tests of `TestA`, when inherited by `TestB`, will request instances of `B` and treat them as instances of `A`.

When only `NewObject` is defined, any use of it must be cast to the appropriate class under test. It is frequently convenient to define a convenience function that obtains an object to test from `NewObject` and casts it to the appropriate type. The unit test for class `A`, `TestA`, would then contain the convenience function:

```
protected A NewA ()
{
    return (A) NewObject ();
}
```

As this function obtains its object from `NewObject`, it will return an object of the most derived class. In the class `A` and `B` example, in `TestB` it will return an object of class `B` when it is invoked in functions inherited from `TestA`. Abstract unit test classes (such as used for unit testing abstract classes or interfaces) might well return `null` as the value of NewObject.

Virtual Test Functions

Methods and properties are marked as `virtual` in C# when they are intended to be overridden. Methods are overridden when their implementations, and hence frequently their detailed semantics, change. Unit tests frequently test detailed semantics. Therefore, unit tests exercising `virtual` methods and properties should themselves be `virtual`, allowing the unit test function itself to be overridden.

Suppose once again that class `B` specializes class `A`. A virtual function `VFunc` in class `A` can then be overridden by an identically named function in class `B`. The unit test for `A` will have a `TestVFunc` function which exercises the `VFunc` as it is known in class `A`; this function will be inherited by the unit test for `B`. The unit test for `B` must then override this function with one that exercises the `VFunc` as it is known in class `B`.

Unit Tests for Abstract Classes

Abstract classes have the peculiar property that instances of them cannot be created. They therefore cannot be directly unit tested: a unit test requires an instance of a class to exercise.

There are two approaches that have proved fruitful in writing unit tests for abstract classes. The first is to have a unit test as if the class under test were a concrete class, but to withhold the unit test class from the unit test suite. Failing to mark the unit test class with the `TestFixtureAttribute` attribute will cause NUnit to recognize the class is not an executable unit test, and NUnit will not attempt to execute it. Individual test functions retain their `TestAttribute` attribute even when the `TestFixtureAttriute` attribute is withheld from the unit test class. Classes inheriting from the abstract class under test would have unit tests inheriting from the abstract unit test for the abstract class under test. Just as the concrete classes would inherit functionality from the abstract class, the concrete unit tests would inherit testing functionality from the abstract unit test. The abstract class's functionality would then be tested in every concrete class descending from it.

This first approach is frequently sufficient. However, when the abstract class is available for use outside its defining project, independent tests of the abstract class are well advised. A second approach provides these independent tests. In this second approach, the unit test class defines a concrete test subclass of the abstract class under test. The test subclass has minimal functionality in addition to that inherited from the abstract class. The `NewObject` function of the unit test class returns an instance of this test subclass, which is then tested as an instance of the abstract class.

Test classes for subclasses of the abstract class inherit from the test class for the abstract class in the usual way. The inheriting test classes replace the `NewObject` function with their own, so instances of the test subclass (sometimes known as *mock objects*) are generated only in the unit test for the abstract class.

Either of these approaches are adequate for unit testing abstract classes. The second approach has the advantage that the abstract class can be unit tested in the absence of deliverable concrete classes inheriting from the abstract class. However, this advantage comes at the cost of implementing a concrete testable class used only in the unit test.

Unit Tests for Generic Classes

Like abstract classes, generic classes (classes with type parameters) cannot be directly created. They also therefore cannot be directly unit tested. Only constructed generic classes (where the type parameters are replaced by actual types) can be created and unit tested.

Unit testing generic classes is somewhat similar to unit testing abstract classes. However, the unit test for a generic class must have type parameters that parallel those of the generic class under test. To omit such parametrization would be to limit the specializations of the generic class that could be tested. Because it must be a generic class, instances of the unit test class cannot themselves be directly created. Therefore, the unit test class for a generic class cannot create and test mock objects.

Any number of concrete specializations of the generic class can be created and tested. Each such specialization would require a separate unit test, inheriting from the generic unit test for the generic object. As with the corresponding approach to abstract classes, the value of such tests is dependent on whether the generic class is available outside the creating project.

Unit Tests for Interfaces

Interfaces share with abstract classes the property that instances of them cannot be created. Interfaces also therefore cannot be directly unit tested.

The two approaches useful for unit testing abstract classes are also fruitful in unit testing interfaces. A unit test class can exercise the methods and properties exposed by the interface as if the interface were a concrete class. Withholding the `TestFixtureAttribute` attribute from the unit test will ensure NUnit does not attempt to execute it. However, the unit test class is available for constructing unit tests for classes or interfaces inheriting from the interface in question.

It is also possible for the unit test to create a concrete test class implementation of the interface, and to subject this concrete class to the unit test. The functionality exposed by this concrete test class would reflect the desired semantics of the interface. As with the abstract class unit test, this concrete test class would not participate in tests of classes inheriting from the interface.

The interface unit test is intended to be inherited by unit tests for types inheriting from the interface. However, C# (and Java) allow implementation (code) inher-

itance from only a single type, yet allow interface inheritance from an unlimited number of interfaces. It is not at all uncommon for a class to inherit from a fairly large number of interfaces, perhaps in addition to inheriting from an abstract or concrete class.

If a class inherits from only one source (a class or interface), the unit test for that single source can be used as a base class for the unit test of the inheriting class. If a class inherits from multiple sources, the single implementation inheritance limitation of C# (and Java) becomes troublesome for the unit test.

If the interfaces are simple enough, repeating the appropriate unit test functions in all inheriting classes is not terribly burdensome. However, many interface types are too complex for this approach to be viable. In such cases, the unit tests for the interfaces are substantial. Repeating the entire text of the unit test for the interface is both unwise and undesirable.

One approach in this case is to relax the restriction that a unit test be composed of a single class. Instead, a unit test has a class for each complex interface exposed by the class under test. Each unit test class inherits from the unit test for one of the exposed interfaces. Aside from an appropriate override of the `NewObject` function, only one unit test class need have any new content, exercising the class under test.

Unfortunately, the multiplicity of unit test classes propagates down the unit test class hierarchy. Due to the single implementation inheritance limitation, classes inheriting from a class with multiple unit test classes have at least as many unit test classes in their own unit tests. Each of the unit test classes of the parent class must be inherited by a unit test class of the child class.

Example

The Classes Under Test

Consider for a moment the problem of printing checks. Although perhaps a rarity in these electronic money transfer times, checks were formerly a common way of relatively securely and relatively rapidly transferring funds from one entity to another. A check is a negotiable draft containing, among other things, a description of the amount of money to be transferred. This description has two parts: a numerical part and a textual part. The numerical part uses digits: for example, "$105.05." The textual part uses words: for example, "One Hundred Five and 05/100 Dollars." The numerical part is easily produced with built-in functionality in almost every system. Producing the textual part generally needs specific programming.

Text, of course, is in a language, and language is culture-specific. When designing a program that can potentially be used in multiple cultures, it is useful to separate the general concept of "produce a string" from the specific concept of "produce a string in the language associated with a specific culture." In other words, it is useful to define an interface used by all number to word translation classes. Call this interface `IWordNumber`.

Suppose further that there are two languages of immediate interest, English and Japanese. An initial design might have two concrete classes, `JapaneseWordNumber` and `EnglishWordNumber`, each implementing the `IWordNumber` interface. Unfortunately, English has two widespread dialects which represent numbers differently. Confusingly, a US English "billion" is a British English "thousand million," and a British English "billion" is a US English "trillion." However, the dialects have the same descriptions for numbers up to 999,999,999. The number to word translation classes for these two language can therefore share a large amount of data and logic. This suggests the use of an abstract class with two concrete descendants. An adjustment to the design to handle these two dialects of English suggest an abstract class called `EnglishWordNumberBase`, and concrete classes `AmericanWordNumber` and `BritishWordNumber`.

The author's personal utility library contains these classes, derived in this way. The coding standards for this library also require all objects to be serializable and to implement value-based semantics for the methods `Equals()`, `GetHashCode()`, and `ToString()` inherited from `System.Object`. The coding standards also require all functional classes to implement an `IComponent` interface consisting of the following:

- The `System.ComponentModel.IComponent` interface. This interface, defined by .NET, allows the object to be manipulated by Visual Studio and by client application run-time GUIs. This interface inherits from `IDisposable`, which requires the class implement a `Dispose()` function. This interface also defines a `Disposed` event, which the object is obliged to publish when `Dispose()` is invoked.
- The `System.ComponentModel.INotifyPropertyChanged` interface. This interface, also defined

by .NET, requires the object to publish `PropertyChanged` events when a property value changes. This allows visual representations of the component to update themselves when the component changes.

- An `IsDisposed` property. The `IDisposed` interface requires the object to implement the `Dispose()` method. After being disposed, an attempt to access the object throws an exception. This property allows a query against even a disposed object to determine if it is disposed.

The utility library provides both an `IComponent` interface (distinct from that provided by .NET) and a `Component` abstract class providing partial support for the interface.

The result of all this is the class hierarchy shown in Figure 1. This is a small portion of the overall class hierarchy provided by the library.

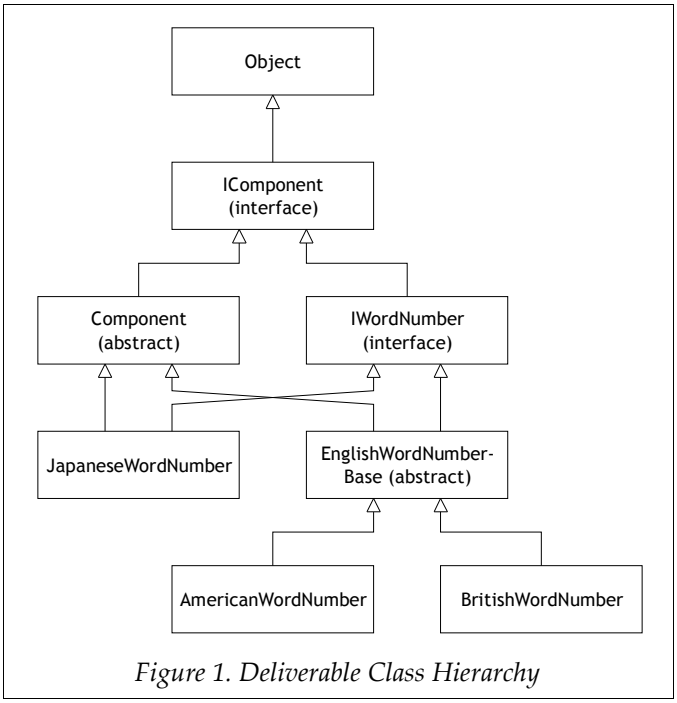

Figure 1. Deliverable Class Hierarchy

The class `AmericanWordNumber`, as an example, descends from two abstract classes and two interfaces, and has imposed constraints on the properties and methods inherited from `Object`. Even though the class's function is conceptually simple, the class is in actuality moderately complex. Its unit test, therefore, will be similarly complex. Its unit test will inherit from such unit tests for `Object`, `IComponent`, `IWordNumber`, and `EnglishWordNumberBase` as exist.

The Unit Test Classes

The unit test classes for these word to number related classess are shown in Figure 2. `TestObject`, the root class of the inheritance hierarchy, is an abstract unit test class. Its class declaration is as follows:

```
namespace BrianHetrick.Util.UnitTests
{
    [Serializable]
    public abstract class TestObject
    {
        …
    }
}
```

Its `NewObject()` function is:

```
protected virtual object NewObject ()
{
    return null;
}
```

Because it is not known whether the object under test requires disposition, TestObject disposes of objects conditionally, as follows:

```
protected void DisposeOf (object o)
{
    IDisposable d = o as IDisposable;
    if (d != null)
    {
```

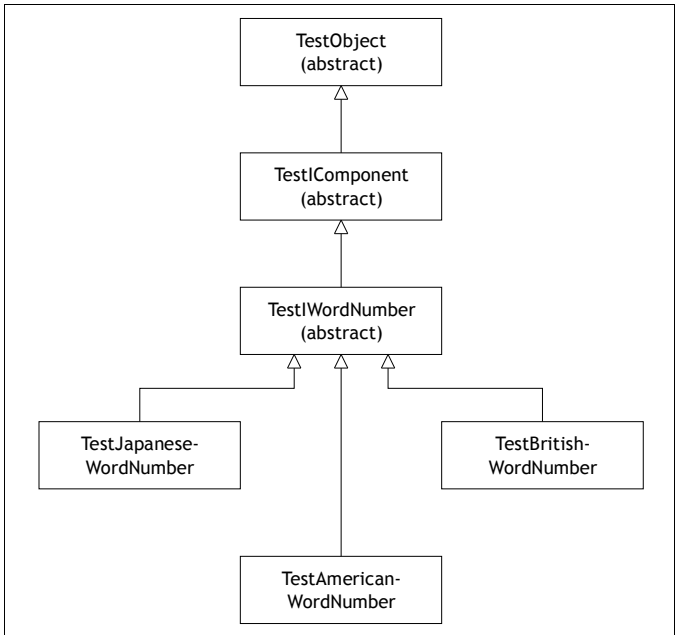

Figure 2. Unit Test Class Hierarchy

```
            d.Dispose ();
        }
    }
```

Note that this function is `protected`, not `private`. This function is available to all descendant classes.

The assertions tested by this class are:

- The `NewObject()` function is overridden.
- The `Equals()` function returns `true` when an object is compared to itself, `false` when an object is compared to `null`, and `false` when compared to an object of a different type.
- The `Equals()` function compares object values, not object references. (The only reliable way to "duplicate" an object (as the object need not implement ICloneable) is to serialize and deserialize it.)
- `GetHashCode()` is implemented and returns the same value for two objects that compare `Equals()`. (The only reliable way to "duplicate" an object (as the object need not implement `ICloneable`) is to serialize and deserialize it.)
- The object is `Serializable`.
- `ToString()` exists and returns something.

The next class in the hierarchy, `TestIComponent`, is again an abstract unit test class. It checks the class's implementation of the `IComponent` interface. It exercises `Dispose()` and `IsDisposed`, and ensures disposition publishes the `Disposed` event and the `PropertyChanged` event for the `IsDisposed`.

Similarly, the next class, `TestIWordNumber`, once again an abstract unit test class, tests that the word to number translator under test returns something for the `Culture` property and returns different values for 0 through 1999.

Finally, the class `TestBritishWordNumber` (as an example) is a concrete class and actually executes as-is. The assertions explicitly tested by `TestBritishWordNumber` are:

- The `Culture` property of an `BritishWordNumber` is "en-UK".
- `BritishWordNumber` uses thousand, million, thousand million.
- `BritishWordNumber` handles the entire `int` range. (This checks the results for -2147483648, -2147483647, -2, -1, 0, 1, 2, 2147483746, and 2147483647 against the literal strings "Negative Two Thousand One Hundred Forty Seven Million Four Hundred Eighty Three Thousand Six Hundred Forty Eight" and so on).

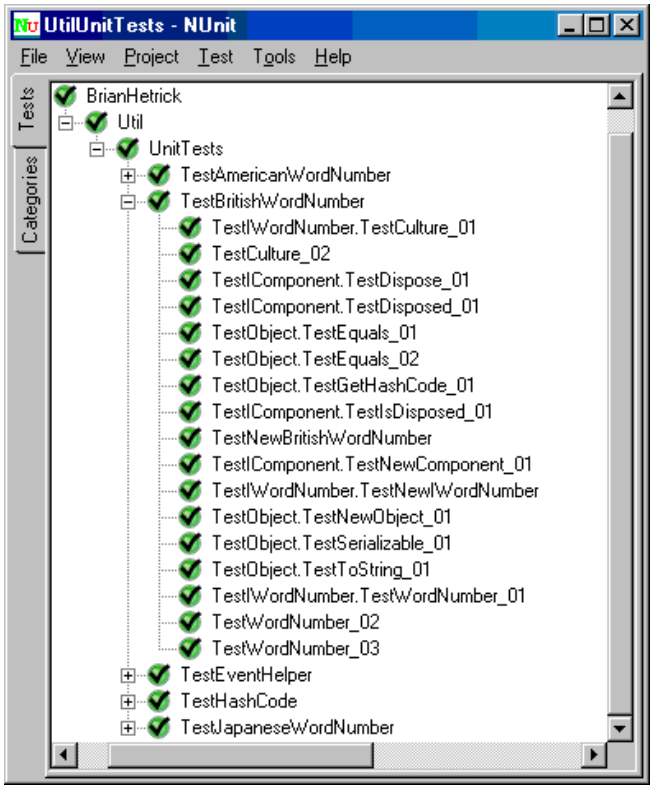

Figure 3. NUnit Output Showing Inherited Tests

The abstract unit tests from which `TestBritishWordNumber` inherits do not execute separately. However, as shown in Figure 3, this unit test runs the tests defined in the abstract unit test classes from which it inherits. Thus, even though there are only three assertions explicitly checked in `TestBritishWordNumber`, and these assertions are all specific to `BritishWordNumber`, the `BritishWordNumber` class is tested against 17 assertions in total. The assertions checked include those of `TestIWordNumber`, `TestIComponent`, and `TestObject` — the tests for the classes and interfaces from which `BritishWordNumber` inherits.

Further Information

This Application Note and the source code for the classes and unit test classes described herein are available at http://www.brianhetrick.com/an/AN-4.zip.

The NUnit testing framework and documentation is available at http://nunit.org/index.php.

The Visual C# 2008 Express Edition development environment used to develop the code described in this Application Note and documentation is available at http://www.microsoft.com/express/vcsharp/.

The techniques described in this Application Note are not new. The earliest description the author can find is http://www.ryanlowe.ca/blog/archives/000365_unit_-testing_multiple_interface_implementations.php, from 2003. The techniques, particularly that of using a test concrete class (mock object), appear to be more widely known in the Java community than in the .NET community. However, even in late 2009, the question of how to test abstract classes or interface implementations commonly appears in internet forums devoted to both Java and .NET development.

Disclaimer

The author believes all content of this Application Note and the software it describes is a straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes or any portion of it, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License