Using Model-Based Testing for API Testing in C#

November 2010

Author: Peter Shier, Microsoft

1. Expectations:

1. Learn fundamentals of the C# Language
   a. Expressions, control flow, data types, operators, exception handling
   b. Namespaces, interfaces, events, delegates
   c. Program structure – Main() entry point, processing command line arguments
   d. Basic utility classes: collections, iterators
2. Learn fundamentals of ADO.NET
   a. API Architecture
   b. Database connections
   c. Query execution
   d. Processing query results
3. Learn the fundamentals of model-based testing and the SpecExplorer tool.
4. Design and implement a model for an aspect of the ADO.NET API of your choice, generate tests, run them, and report results (extra credit for any bugs found in the API☺).

2. Learning and Tools

Visual Studio

Visual Studio is Microsoft's GUI development environment. It covers all of our programming languages and most of our APIs. You will need it for all the development aspects of this project. If you do not have access to Visual Studio then you can download a trial version at http://www.microsoft.com/visualstudio/en-us/download that will work for 90 days.

C#

If you are not already familiar with the basics of the C# language then start by learning that. The Microsoft Developer Network (MSDN) website has a good reference and there are also many books available on the subject.

Start with the MSDN C# home page at http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx. The topics you'll need are listed under "Expectations" above. There is no need to become an expert in the language – you'll need just enough to become a user of the ADO.NET API. This exercise will also help you learn to acquire a new programming language efficiently. After acquiring the basics listed above and writing a sample program or two you'll know enough to move on to your work in ADO.NET. Note that there is no need for any GUI programming – everything can be done with simple command line invocation and console output so don't waste any time on it. Also note that multi-threaded programming is not necessary for this course.

ADO.NET

ADO.NET is Microsoft's .NET API for data access. It is very powerful and can do many things but for the purposes of this course it will suffice to know how to connect to a database, query or insert to a table, and print the results out to a console window. That may not sound like much but there are numerous concepts that you'll need to acquire to reach that point. The MSDN home page for ADO.NET is at http://msdn.microsoft.com/en-us/library/e80y5yhx(v=VS.71).aspx and it has all the documentation and some samples. There is more good info at http://msdn.microsoft.com/en-us/data/aa937722.aspx.

As with C# part of this exercise is learning to acquire knowledge of a new API quickly and efficiently. When learning a new API get the general concepts first: what does it do, how is it structured, how do basic operations work. Then review all of the functions at a shallow level to learn what is available within the API. Don't get bogged down with details of any particular function. When you need to use a function you'll learn it in detail at that point.

SQL Server

ADO.NET is only useful if it has some data to process. For the database you can use SQL Server Express (download at http://www.microsoft.com/downloads/en/details.aspx?FamilyID=58ce885d-508b-45c8-9fd3-118edd8e6fff&DisplayLang=en). This is a free edition of Microsoft's enterprise database product and it can run on the same machine as your client code. There is a tutorial at http://msdn.microsoft.com/en-us/library/ms167593(v=SQL.100).aspx. In particular, it describes the SQL Server Management Studio tool which is a GUI that enables exploring the structure and contents of your databases.

There are code samples and sample databases at http://sqlserversamples.codeplex.com/. You may choose to use one of the sample databases for your modeling work or you can simply connect to the master database (contains the metadata about all of the settings and databases on the server). In SQL Server API testing we often use the master database because it is sufficient to reach the desired code paths.

The SQL Server documentation for ADO.NET access is at http://msdn.microsoft.com/library/system.data.sqlclient(v=VS.90).aspx.

Model-Based Testing and Spec Explorer

SpecExplorer is a model-based testing tool that works with Microsoft .NET languages such as C#. It actually uses C# as its model definition language and it can generate tests from the model that are implemented in C#. The SpecExplorer home page is at http://visualstudiogallery.msdn.microsoft.com/en-us/271d0904-f178-4ce9-956b-d9bfa4902745. There are some excellent training videos at http://social.msdn.microsoft.com/Forums/en-

US/specexplorer/thread/e9d70290-a529-4142-987b-a866fcc19c34. These videos will teach you the fundamentals of modeling as well as how to use the tool.

3. OK, I learned all that stuff. Now what?

The major goal of this project is to apply the technique of model-based testing to the problem of API testing. An API like ADO.NET has about 25 classes with 10-15 methods/properties per class and lots of rules governing input domains, call sequences, thread usage, and so on. Parameters to methods often accept the full range of a data type (e.g. a 32 bit integer or an arbitrary English string). If we assume an average of 3 parameters per method you can already see the combinatorial explosion. The problem gets worse when the API involves a connected entity such as a database server because all possible pairs of operating systems, client versions, and server versions must also be taken into account.

If it is not possible to test every possible permutation then how do you choose what to test? One approach is to choose a particular aspect of the API (e.g. database connection) and possibly some sub-aspect (e.g. connection string specifications). Looking at the documentation for ADO.NET connection string specifications for SQL Server (http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring(v=VS.90).aspx) there is a very complex set of rules that define all the different possibilities for connecting to a SQL Server database. Those rules are encoded in English and comprise the functional specification for that aspect of the API. If you read them as a software tester you'll undoubtedly have dozens of test cases that come to mind. In truth there are probably a virtually infinite number of cases just for this one function.

Modeling an aspect of a software system is equivalent to using an airplane model in wind tunnel testing. The airplane model looks like very much like the real thing and in fact its aerodynamic surfaces are exactly the shape of the actual aircraft. Everything else about the model is either fake or simply not there. For example, there are no glass windows, no seats, no avionics, no baggage compartment, etc. Even without all that detail it is still possible to check the properties of the aerodynamic surfaces in great depth. Software modeling is based on the same principle – describing one or more aspects of the system under test in fine detail while ignoring the rest. This description is then used to test those aspects comprehensively without being bogged down by the remainder of the system.

With model based software testing a tool like SpecExplorer translates the English specification into a machine-readable form. SpecExplorer uses C# as its model specification language but it could have used any other existing programming language or even invented a new one for that purpose. C# has a couple of very cool features called 'attributes' and 'reflection' that essentially allow the user to extend the language in a custom fashion and then explore those customizations programmatically on a given body of code.

Attributes enable associating arbitrary properties with a language object such as a class or a method. For example, SpecExplorer defines an attribute called 'StateInvariant'. When applied to a Boolean object property it indicates that the property must be true for any state of the object.

Reflection enables querying a code object to determine its custom attributes.

SpecExplorer uses these features to enable the user to annotate the C# code with a set of rules that it uses to encode a model of the system under test. In the StateInvariant example SpecExplorer can check an object property for this attribute and if it is present use that knowledge during model verification to ensure that the invariant is not violated in any state.

Once the model is constructed, SpecExplorer can verify it in great depth and breadth. For the SQL Server connection string example that would mean checking that all the rules written in English make sense and do not conflict with each other. For example, when modeling an API a design bug might be that two different functions acquire the same lock and can be called in succession. The second call will of course wait forever. With modeling you can specify that a function acquires a lock and also the combinations and possible sequences of function calls. If you did not notice this potential deadlock but encoded the functionality in your model then SpecExplorer would find it during verification. This is very valuable because a bug like that in an English specification would be easy to miss with human review. In working code it would be missed if you didn't create a test case for it and a customer might be the first one to encounter it. Also, fixing a bug so early in the development process is always the cheapest path to quality and modeling enables a larger number of bugs to be found early.

The other thing SpecExplorer can do with a model is generate tests from it. The generated test code is in C# and it can be run directly against the system under test. Of course a model of even a small portion of a large system may still produce a virtually infinite number of test cases. SpecExplorer deals with this state space explosion in two ways: carving out a portion of the model and random test generation. By carving out a portion of the model you tell SpecExplorer to generate a set of tests that relate only to a subsection of particular interest. Once you have made that decision then SpecExplorer will generate a set of tests that making some intelligent choices to achieve useful test coverage. These choices can also be narrowed down further to restrict their inputs.

SpecExplorer also has a feature called "on the fly" testing that does not generate the test code but instead generates tests and runs them on the spot. This has the advantage of generating different tests each time it is run and ultimately achieving broader exploration of the state space of the SUT. It has the disadvantage that there is no single body of test code that can be run repeatedly to check for regressions in future changes to the SUT. Both approaches are useful and might be used for different circumstances. For example, a build verification test would probably be best done with a repeatable set of tests so that every build is known to pass a minimum standard set of functionality. On the other hand, a stress test that runs for days or weeks would benefit from on the fly testing that never repeats itself.

4. Modeling and Testing the API: The Heart of the Project

The final assignment for the project is to pick a subset of the ADO.NET API, write a model for it, verify your model and then generate and run the tests. As you can see from the example above, even a very narrow aspect such as connection strings can result in a rather large and complex model so choose your API subset carefully.

Your final presentation should include the following:

1. An overview of the API subset you chose – what it does, how it works, and examples of key rules governing its usage.
2. A tour of your model and the rules it encodes. You can use SpecExplorer's visual model explorer as an aid for this.
3. A tour of a generated test case.
4. Your thoughts on the value of model-based testing for API testing vs. other approaches.
5. Your suggestions for new features you'd like to see in the API and in SpecExplorer.

<End of project description>