The Design of a System for Testing Database-Centric Software Applications using Database Surrogates

Adrian W. Bonar
The Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

This paper presents the design of a system for testing database-centric software applications using surrogate databases. Traditional testing approaches, such as using test bed databases and using stub code or mock objects, are often either difficult to implement and manage, or do not provide entirely effective verification of the functionality of the application under test. Testing database-centric applications using database surrogates addresses these issues. A database surrogate is a data source, such as an XML file, which has the same structure as the backend database of the application under test. Database surrogates can be easier to implement and manage than actual test bed databases, but provide a mechanism for thoroughly testing the functionality of the application under test. The essence of the surrogate database testing system is to create a very lightweight library which provides an interface which is independent of the actual physical implementation of the underlying data source.

Keywords: Automatic testing, programming environments, software libraries, software quality, software testing.

1. Introduction

Consider the general problem of testing a software application which is based on a backend SQL database data store. Such applications are common. Examples include Web based shopping systems and corporate inventory systems. This paper presents an overview of an efficient system to test such applications. There are several approaches to testing database-centric applications. One common approach is for software test engineers to create a test bed database which exactly replicates the structure of the production database [1]. This test bed database can then be populated with rich test data (in the sense that the data is specifically designed to work with a test harness and particular set of test case inputs and expected values), and then a test harness can be written which exercises the application under test and the associated test bed database. This replica test bed database approach is effective but has several drawbacks. The approach requires significant time and effort, and requires that the test engineers using the approach have significant database development skills. Additionally, if on the one hand, a single replica test bed database is used for multiple test case scenarios, the approach is slow because each test case must reinitialize the test bed to a known state. On the other hand, if separate test bed databases are maintained for each test case, the approach can be difficult to manage because of the large number of databases.

An alternative to the replica test bed database approach for testing database-centric software applications is to use stub code [2]. Stub code is relatively simple code which partially simulates the behavior of a production database. Although many modern programming languages allow application code to query the application's database backend by directly embedding SQL language statements in the application code, in most situations application developers use some form of wrapper code, written in the application development language, to encapsulate SQL language queries. Stub code has the same interface to an application's database backend as the application wrapper code; however, instead of connecting to a backend database, stub code examines the calling code input arguments, and uses some form of code logic to return one or more values in the same format as an actual return from the backend database. When used in conjunction with unit testing, stub testing is sometimes called testing with mock objects [8]. Testing a database-centric application with stubs is generally quicker and easier than using the replica test bed database approach, but testing with stubs does not thoroughly verify the functionality of the application under test.

This paper presents a brief overview of a system which provides superior functional verification of a database-centric application than a stub-based testing approach, but is easier to implement and manage than a replica test bed database testing approach. The essential idea of the system is to use XML data files to replicate the structure of the application production database. Separate XML files can be maintained to store rich test case data. The approach described in this paper is called testing with surrogate databases to distinguish the approach from alternative techniques. Although the surrogate database testing system described here targets database-centric software applications which are written in the Microsoft .NET programming environment, and which use the LINQ (Language Integrated Query) code wrapping framework, the system principles are general and apply to alternative programming environments and wrapping frameworks.

2. Database-Centric Software Applications

In general terms, database-centric applications provide a user interface to perform CRUD (create, read, update, delete) operations on a backend data store [4]. Although application code can use embedded SQL statements, usually in the form of a simple string data type, to perform operations on the backend database, wrapping SQL code inside native application language code often provides superior security and maintainability. However, simple wrapping approaches enclose the application data tier in some form of interface not directly tied to the data source [3]. The LINQ framework enhances .NET languages such as C# and VB.NET by adding query syntax as a first-class language constructs. Compared to embedded SQL statements, advantages of using an integrated wrapping framework such as LINQ include enhanced debugging capabilities and design-time data type checking. Developing application software using an integrated wrapper framework such as LINQ can make the development process easier because the framework code bridges the gap between the application and the application's data source -- without the framework, a developer would need to write code with similar functionality.

Although the use of integrated wrapper frameworks such as LINQ provides many advantages, the underlying architecture of these frameworks still fails to reach part of the concept's potential usefulness. Consider that a single query is compatible with multiple data sources as long as the object type structures of the data sources are equivalent. However, LINQ provides similar, but slightly different, interfaces to different types of data sources. For example, even if a SQL database and an XML file had equivalent structures, the LINQ syntax for querying these two sources would be somewhat different. The essence of the system described in this paper is to create a higher level of abstraction which provides a single, uniform interface to equivalent SQL databases and XML files. Creating a thin abstraction layer encapsulated as a drop-in library allows testers to replace a SQL test bed database with a collection of XML files that can be dynamically swapped out or altered to accommodate individual test case scenarios. The architecture for this solution is outlined in Figure 1. The system described in this paper concentrates only on querying data sources; however, the technique can be extended to support other operations such as editing and deleting data.

3. The Development Environment

When creating a database-centric application using an integrated wrapper framework such as LINQ, a developer must create an entry point for LINQ-to-SQL in order to access the application's database. Such an entry point can be created by using a tool with a GUI interface hosted by an integrated development environment program, or the entry point can be created by using a separate shell-based utility tool [7]. Using either technique, a file is generated which contains a collection of classes which represent the architecture of the target backend database. In the case of the LINQ framework, the entry point into the database architecture is a child class of a parent DataContext class. This class represents the backend database and provides a programmatic interface to the database in a specified language. For example, when entered on a shell command line, the command:

```
> sqlmetal.exe /server:(local)
  /database:dbTarget /code:proxy.cs
```

uses a utility program named sqlmetal.exe to examine a SQL database named dbTarget located on the local host machine, and generates a file named proxy.cs which contains C# code, including classes derived from the DataContext class which can be used to perform operations on the database. In a development environment, developers use the mapping code inside the application logic code to perform operations on the backend database. The surrogate database testing system uses the mapping code as a basis for creating a thin bridge-like layer encapsulated in a code library. This approach enables the actual implementation of the underlying data source to be abstracted while exposing a common data source interface.

The bridge layer's primary task is to transform the DataContext subclass's interface from one which is SQL-specific to one which is data source agnostic, while still maintaining the idea that there may be a LINQ provider which is actually executing a query using the interface. There are two primary interface definitions that a LINQ query can leverage. The most abstract, and the only interface really needed for LINQ to query against, is IEnumerable. This interface is used when querying against

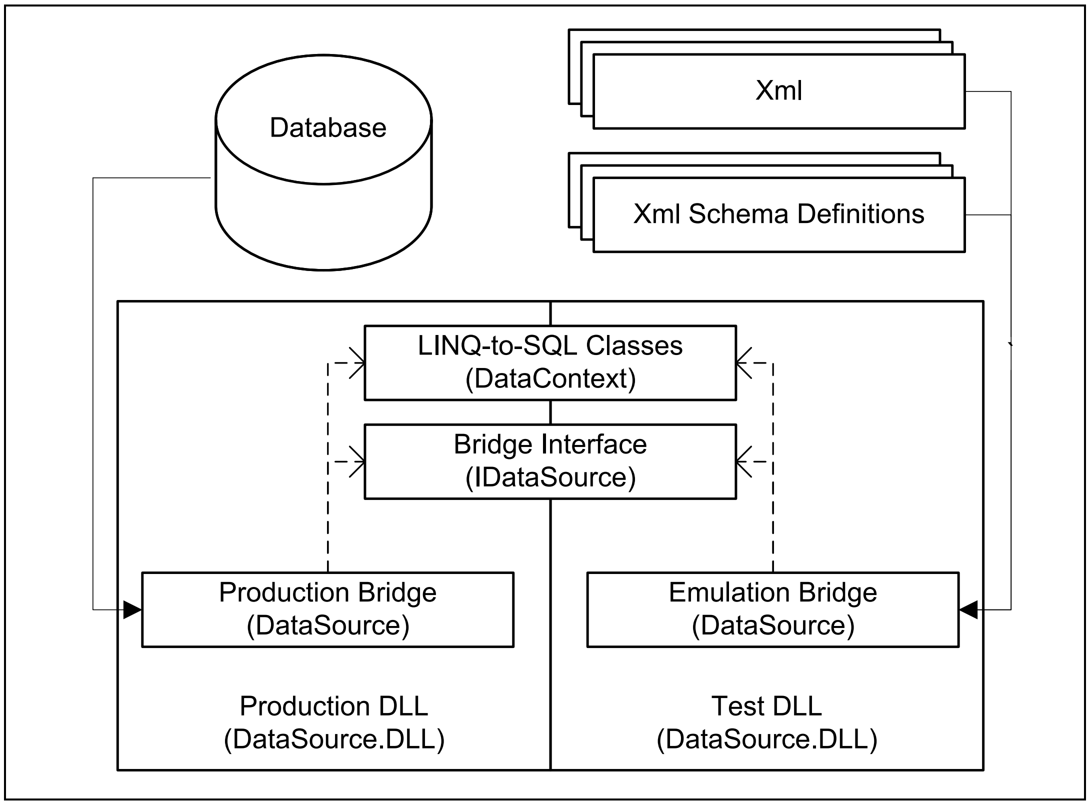
Database
Xml
Xml Schema Definitions
LINQ-to-SQL Classes (DataContext)
Bridge Interface (IDataSource)
Production Bridge (DataSource)
Emulation Bridge (DataSource)
Production DLL (DataSource.DLL)
Test DLL (DataSource.DLL)

Figure 1. Emulating a LINQ data source using a drop-in DLL architecture

most .NET Framework collection types and assumes the framework, rather than an external provider, is going to execute the query. The other interface is IQueryable. The IQueryable interface is intended to expose LINQ provider data sources. The API sets of the two interface definitions are almost identical, but an IQueryable interface assumes the burden of the query is placed upon the specific provider (LINQ-to-SQL, LINQ-to-Entities, LINQ-to-XML, and so on.) Since the intent of the surrogate database system is to emulate the LINQ-to-SQL data source and not necessarily hide this fact, the system exposes the IQueryable interface in the data source abstraction. For example, an interface to a data source named WidgetDataSource, which is realized as either a SQL database or an XML file, can be defined:

```
public interface IWidgetDataSource
{
  IQueryable<GetAllCategory>
    GetAllCategories {
      get;
    }
}
```

This interface defines a property named GetAllCategories which returns a query-able collection of objects of type GetAllCategory. The production implementation of this interface to a particular SQL database is a simple pass-through to the associated LINQ-to-SQL DataContext subclass. For example, if an object, derived from the DataContext class created by a mapping tool, is instantiated as wDataContext, then a possible implementation of the IWidgetDataSource interface to the associated database is:

```
public IQueryable<GetAllCategory>
  GetAllCategories
{
  get {
    return wDataContext.GetAllCategories;
  }
}
```

In short, the GetAllCategories property of the bridge layer component calls the GetAllCategories property defined in the DataContext definition. It is important that this implementation is placed into its own library to facilitate replacement with the emulating implementation. With the bridge to the LINQ-to-SQL data source in place, the emulation bridge pictured in Figure 1 can now be created. There are two primary components that the emulation bridge needs to share with the production implementation - the bridge interface and the LINQ-to-SQL classes. Access to the interface definition is

```
private IQueryable<T> Get<T>()
{
   // Retrieve source paths from settings file
   string XMLFullPath = Path.Combine(Settings.Default.XMLPath, typeof(T).ToString() + ".xml");
   string XSDFullPath = Path.Combine(Settings.Default.XSDPath, typeof(T).ToString() + ".xsd");

   // Enforce schema validation
   XMLReaderSettings xrs = new XMLReaderSettings();
   xrs.ValidationType = ValidationType.Schema;
   xrs.Schemas.Add(String.Empty, XSDFullPath);

   // Deserialize the XML file into an IQueryable collection
   using (FileStream fs = new FileStream(XMLFullPath, FileMode.Open, FileAccess.Read))
   {
      using (XMLReader xr = XMLReader.Create(fs, xrs))
      {
         IList<T> rowList = new List<T>();

         XMLSerializer xs = new XMLSerializer(rowList.GetType());
         rowList = xs.Deserialize(xr) as IList<T>;

         return rowList.AsQueryable<T>();
      }
   }
}

public IQueryable<GetAllCategory> GetAllCategories
{
   get { return Get<GetAllCategory>();
}
```

Figure 2. De-serializing an XML file into a typed Queryable interface

necessary to expose an API identical to the production database implementation. Access to the class definitions are necessary to support the typed queries required by LINQ-to-SQL code. There are several different methods for accomplishing this sharing of source files. One approach is to insert pre-build events into the emulation project and copy the current implementations of the required files from the production project to the emulation project. Another approach is to rely on the file system or source management to maintain mirrors of the required shared files in both projects. A third approach is to place both the production project and the emulation project in the same file directory. Regardless of which sharing mechanism is employed, in order to enable transparent drop-in functionality, both projects should output identically named libraries (e.g. WidgetDataSource.dll).

4. The Get Meta-Method and Creating Surrogate Data Sources

With the production and emulation projects set up, the surrogate database entities can be created. In the case of emulating a production database, using XML files is a convenient design choice to store both the structure and data of the application's backend database. The emulation implementation uses a combination of XML serialization and generics to simplify this common interface. The surrogate database system defines a meta-method named Get() as shown in Figure 2. The definition of the Get() method allows test engineers to write code which uses an XML file as a surrogate for a SQL database, using identical code as in the case when the data source is the production database.

There are two approaches to creating an XML file which has an equivalent structure and equivalent data to a SQL database. In addition to a fundamental XML file, the surrogate testing system also requires XML schema definitions for use in data source validation. One approach is to first create an XML file by serializing the target SQL database, and then to use the resulting XML file to infer an XML schema definition. A second approach is to analyze the target backend database to produce a SQL schema definition of the database, then use the database schema to produce an equivalent XML schema definition, and then use the XML schema to create an XML data file which conforms to the XML schema. The first approach is generally easier but somewhat less reliable than the second approach for complex databases. Implementation of the first approach might resemble:

```
var categories = from c
  in wDataContext.GetAllCategories
  select c;

IList<GetAllCategory> categoryList =
    categories.ToList();

XMLSerializer xs = new
  XMLSerializer(categoryList.GetType());

XMLWriter xw =
  XMLWriter.Create(XMLFilename)
xs.Serialize(xw, categoryList);
```

The second approach to generating surrogate XML data files and schema definitions is non-trivial. In practice, a reasonable approach is to use any one of several commercially available tools to convert SQL database schema to XML schema in the form of XSD (XML Schema Definition) files.

5. References

[1] Carsten Binnig, Donald Kossmann, and Eric Lo, "Towards Automatic Test Database Generation", *IEEE Data Engineering Bulletin*, 2008, vol. 31, no. 1, pp. 28-35.

[2] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. "A Framework for Testing Database Applications", *Proceedings of the 7th International Symposium on Software Testing and Analysis*, August 2000, pp. 147-157.

[3] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. "An AGENDA for Testing Relational Database Applications", *Software Testing, Verification, and Reliability*, 2004, vol. 14, no. 1, pp. 17-44.

[4] B. Daou, R. A. Haraty, and N. Mansour, "Regression Testing of Database Applications", *Proceedings of the ACM Symposium on Applied Computing*, 2001, pp. 285-289.

[5] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. "Quickly Generating Billion-Record Synthetic Databases", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994, pp. 243-252.

[6] F. Haftmann, D. Kossmann, and A. Kreutz, "Efficient Regression Tests for Database Applications", *Proceedings of the Conference on Innovative Data Systems Research*, 2005, pp. 95-106.

[7] James D. McCaffrey, "Testing SQL Stored Procedures using LINQ", *MSDN Magazine*, April 2008, vol. 23, no. 5, pp. 99-104.

[8] David Saff and Michael D. Ernst, "Mock Object Creation for Test Factoring", *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004, pp. 49-51.

[9] D. Willmor and S. M. Embury, "An Intensional Approach to the Specification of Test Cases for Database Applications", *Proceeding of the 28th International Conference on Software Engineering*, 2006, pp. 102-111.

[10] J. Zhang, C. Xu, and S. Cheung, Automatic Generation of Database Instances for Whitebox Testing", Proceedings of the 25th Annual International Computer Software and Applications Conference, October 2001, pp. 161-165.