



Data-Driven Transaction¹ Based Unit Tests

Volkan Sevincok¹, Murat Ozemre¹ and Engin Yorgancioglu¹

¹Bimar Bilgi Islem Hizmetleri A.S.

Izmir, Turkey

(volkan.sevincok,murat.ozemre, engin.yorgancioglu)@bimar.com.tr

Abstract:

Unit testing is a fundamental complementary activity in robust software development. It promotes reliable code and drives testability into design, as well as helping discover defects at an early stage where they can be eliminated with relatively low cost compared to later stages. “Data-Driven Transaction-Based Unit Test”’s (DTUT) helps fast test development with increased test-case maintainability in applications based on similar architectures to that of application-under-test. Writing trivial unit-tests is reduced to inserting a row in the test database which is a result of adopting data-driven approach. Transaction-level scope of tests allows achieving high coverage quickly with the option of intensifying unit tests in critical areas of the system.

Keywords:

Unit Test, Data Driven, Transaction Based, Rota framework, EDRA

1 Introduction

Unit testing is a fundamental activity in most software development processes. It promotes reliable code and encourage testability into design, as well as helping discover defects at an early stage where they can be eliminated with relatively low cost compared to later stages. Automation of unit tests allows regression testing, by which functionality and quality can be continuously tracked.

2 Motivation

Unit testing, having the benefits mentioned above, is more effective when it is carried out together with development. However, that could not be the case in this project. The testing process was setup after the project completed its first major phase, and the existing software needed to be tested as well as the new modules that were being developed. This situation required a significant workload to be completed in a limited amount of time. Therefore our goal was to unit test old and new modules rapidly.

However, this goal could not be achieved in traditional ways. Creating unit tests for each class was impractical because of business constraints, namely workforce and time limitations.

¹ Transaction in this article refers to the specific transactional operations of the architecture used in the application-under-test, rather than database transactions.

In this context, we decided to construct a unit-testing framework which would help us achieve our goal. It was necessary to produce a specific solution which would enable exploiting specific properties of the application-under-test (AUT).

3 Application-Under-Test

The AUT is built on *Rota* framework, which has an architecture similar to Microsoft's Enterprise Development Reference Architecture^[2] (EDRA), using ASP.NET and C# on Microsoft Visual Studio 2008 using Oracle 10g database.

This architecture mainly operates on *Request* and *Response* objects communicated through pipelines. Requests are generated upon user interactions with the interface. They are sent to the business layer through pipelines where cross-cutting concerns such as security and logging are handled. In the business layer, *Transaction* objects process the incoming request and generate a Response object to be sent back to the user interface where it is rendered and displayed to the user.

Data-access layer classes are created by using an O/R mapping tool (LLBLGen).

The overview of Bimar-Rota Framework is described in *Figure 1*, along with sample metrics in *Table 1*.

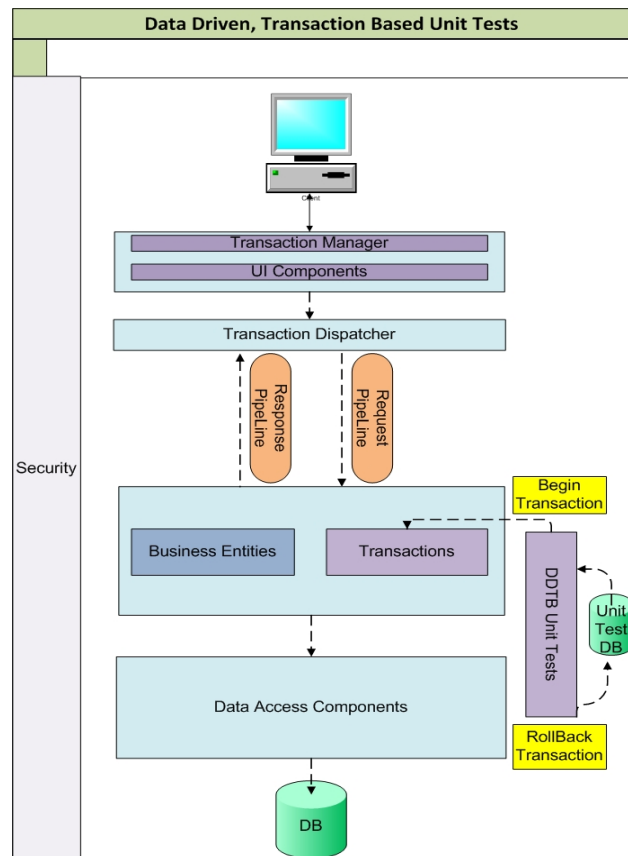


Figure 1. Overview of Bimar-Rota Framework.

Year(2008)	Lines of Code	Number of Screens	Number of Tables (DB)
March	810,985	744	814
April	859,897	780	820

Table 1. Sample metrics for the AUT.

4 “Data-Driven Transaction-Based Unit Test” (DTUT) Framework

DTUT infrastructure was designed with the motivations described above. It was observed that writing unit tests for each smallest unit was unfeasible due to our business constraints. In the on-going debate about the idea that test-units should be isolated and minimal, we chose IEEE Standard for Software Unit Testing^[1] as our reference point:

“A test unit may occur at any level of the design hierarchy from a single module to a complete program. Therefore, a test unit may be a module, a few modules, or a complete computer program along with associated data and procedures.”

Therefore the framework was designed in such a way that *Transaction* classes, which have database interactions, were accepted as our “test units”. This decision allowed our tests to have wider coverage than the simplest unit tests, but they still remained more focused than integration-level ones. With this approach, basic functionality of a wider range of classes could be tested within a limited amount of time, where for areas of higher concern additional resources could be expended.

Additionally, the state of the tested system should not be changed between tests. The transactional architecture facilitated this purpose as well. Before each unit test, the framework declares the beginning of a transaction and after each one, the transaction is rolled back. In this way, all the changes made on the target database are withdrawn and the state remains intact.

Another design decision was to make these tests data-driven. Since Rota provided a uniform transaction execution method all over the AUT, unit tests that execute these transactions could easily be written. An example transaction would look like this fragment:

```
public class AddCustomerTransaction
{
    // ...
    public void Execute(AddCustomerRequestMessage request,
        AddCustomerResponseMessage response)
    {
        // Transaction code
    }
    // ...
}
```

This uniform Execute method enabled the DTUT framework use reflection to execute transactions in a generic way. The new unit-test classes would simply extend this framework, adding trivial code. Test cases would be added as rows to test database. Once the test is run, Visual Studio automatically connects to this database and retrieves these rows one by one and executes them. Sample test code outline would be:

```
// Test class extends framework
public class AddCustomerTransactionTest : TransactionTestBase {
    // Database connection info here
    public void ExecuteTest() {
        // Visual Studio automatically reads rows
        // from test database.

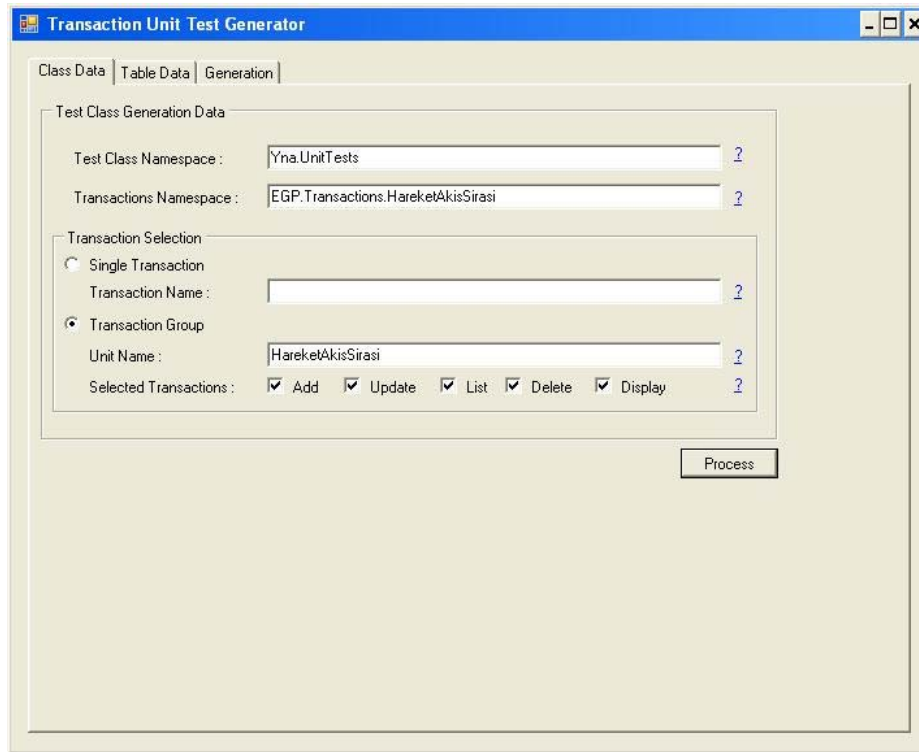
        // Here goes trivial code to get retrieved
        // parameters via UnitTestContext and invoke // Transaction
        // object with these parameters.
    }
}
```

For comparison, an existing unit-test class which was written without using the framework was converted to a data-driven one. This process resulted in a reduction from 18 methods in 560 lines of code to 2 methods in 50 lines in the test class. In return, a table with 18 rows was created in test database where each row corresponded to a test method. As a result, after the initial effort for creating the test class and the table for test data; creating new unit tests would be less expensive.

5 Code Generation For Framework

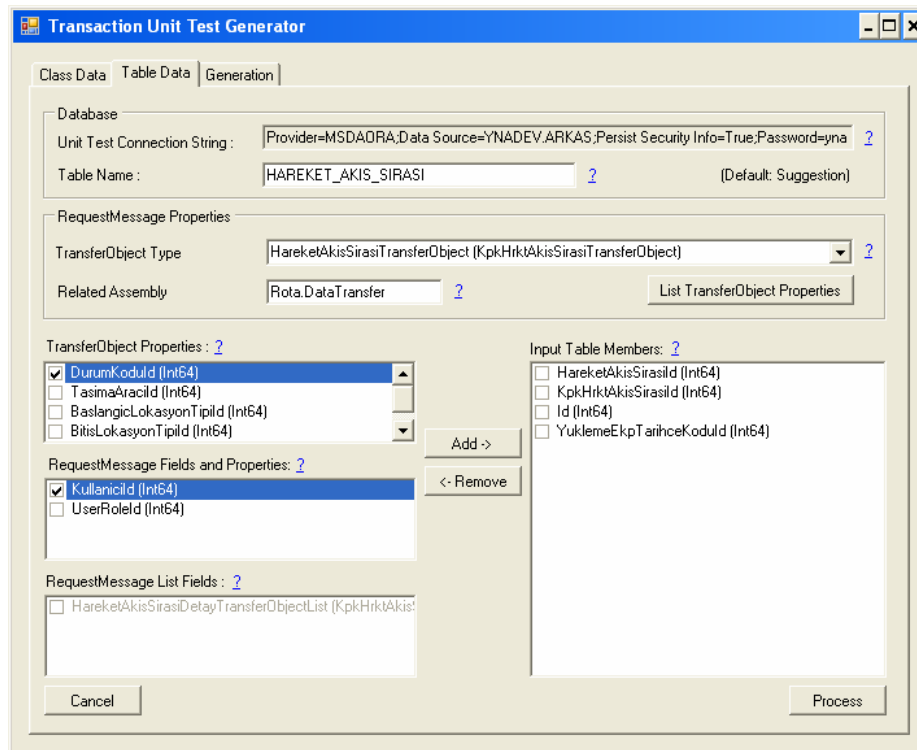
In spite of the gains with the framework, the overhead of creating a table in the database and a test class would be a drawback. Therefore a code generation tool is designed and implemented, which simply lets the user determine the parameters to be used in the unit tests. The generator produces the test class and creates the test data table. The whole process is finished in 3 steps, allowing the user to create the necessary artifacts for the unit test class in usually less than 1 minute. After this part is completed, test cases can be simply inserted as rows into the test database by the developers.

In the generation process, a single unit test or a group of tests can be produced. This is selected in *Screen 1*.



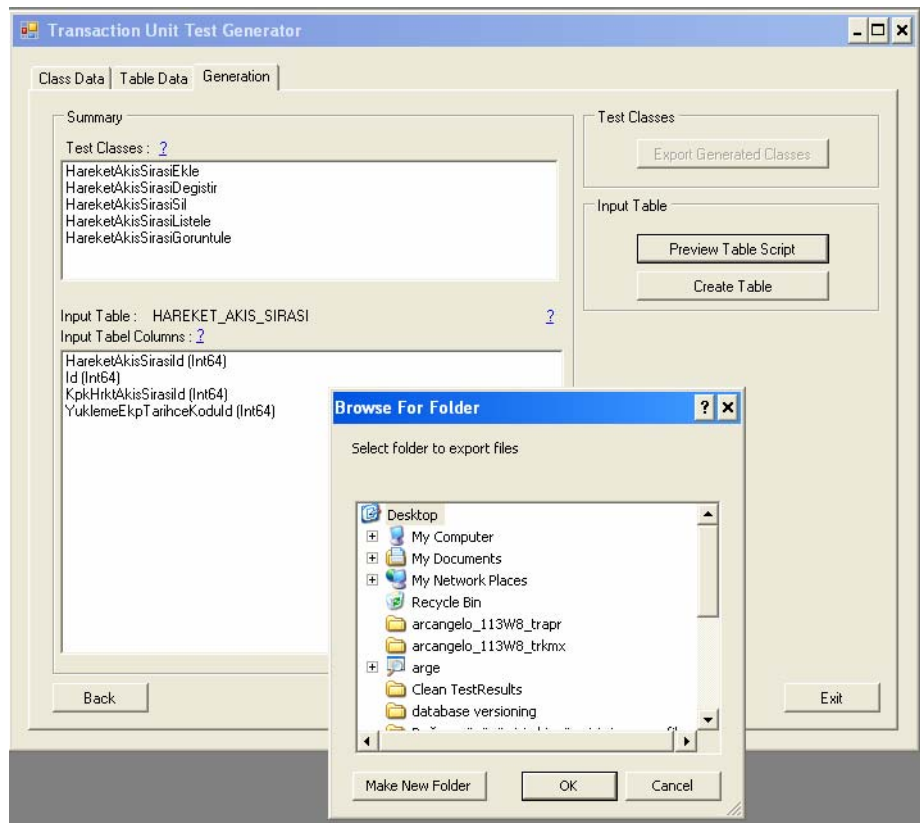
Screen 1. Single / Bulk Mode Generation

In the next step, the properties of the transaction which are going to be used as parameters in the unit test are selected, as in *Screen 2*.

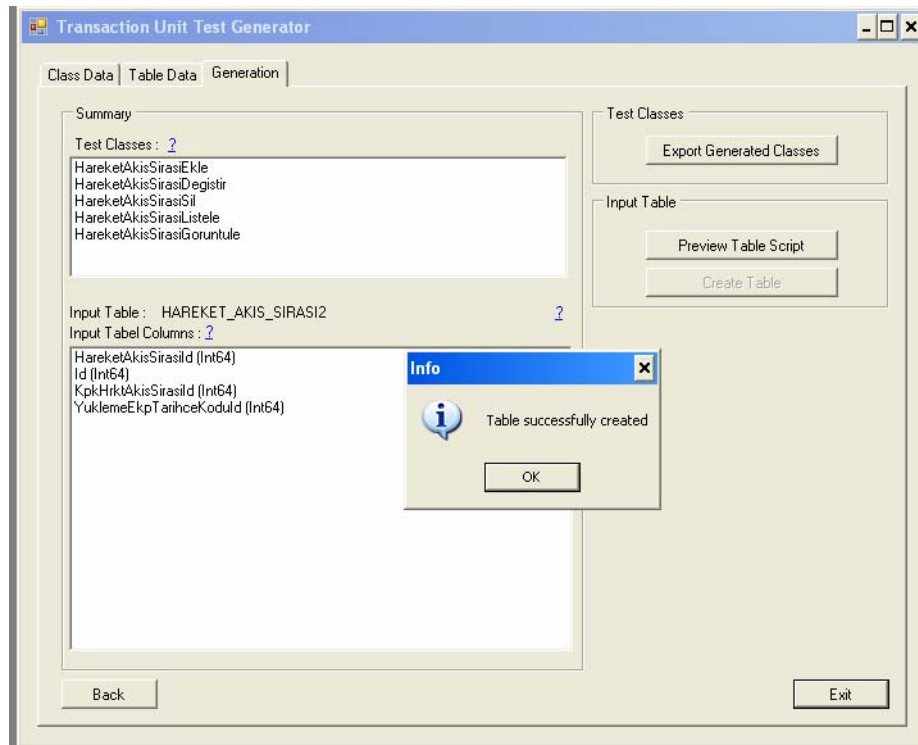


Screen 2. Parameter selection for test table

In the final step, the test class and the table are generated according to user inputs. The class or classes are exported to a user selected folder, as in *Screen 3*.



Screen 3. Class Generation



Screen 4. Table Generation

6 Usage and Testing Process

Using the unit-test framework and the generator, a unit-test creation cycle for a sample unit would be described as below:

1. Test suite design: The design is made so that it involves positive and negative test cases.
2. Test class and test-data table generation: The test class(es) for the specified unit and table(s) to hold test data are created using the generator.
3. Forming test cases: Each designed test case is entered in the test-data tables as a row.
4. Running tests: The tests are run using Visual Studio Team System 2008.
5. Reporting: Visual Studio reports successful and failed tests visually. Each test can be tracked down to its data rows and the reasons for unexpected results can be found easily. If a defect is discovered, it is entered into the defect-tracking system.
6. Defect resolution: The reported defect is fixed by the development team and the resolution is approved by the test team.

In addition, the unit-tests are scheduled to run after each build. This allows regression testing to be performed continuously.

7 Advantages

Data Driven Transaction Based Unit Test Generation reduces trivial unit test generation time to a simple user-interface selection effort and inserting rows to the test. This enables rapid creation of unit-tests which is especially important for testing projects that have sizeable amounts of existing untested code. The uniformity of generation also provides maintainability and it facilitates imposing standards on unit tests. Additionally, a high percent of code coverage can be obtained with relatively little effort, freeing up time for more critical parts of the AUT to be tested in a deeper way.

8 Limitations

The most important limitation of this unit test framework is that its applicable domain covers only projects based on Rota-like frameworks working on .NET architecture. It would also be expected work on EDRA-based applications which have the transactional architecture similar to the one described in this paper. Though this solution could be portable to other architectures as well, no study has been made by the authors for such purpose. Another limitation might be the framework being defined strictly and not allowing atypical unit tests. However it is not claimed to be suitable for all kinds of unit tests, since it is designed primarily to speed up development of unit tests.

9 Conclusion

As with all frameworks, DTUT framework can be useful in its own defined scope and domain. Although it is not a silver bullet, it can facilitate rapid production and standardization of unit tests.

References

- [1] American National Standards Institute, 1986: IEEE Standard for Software Unit Testing, ANSI / IEEE Std 1008-1987; 9.
- [2] Information about Enterprise Development Reference Architecture (EDRA) and Global Bank Reference Implementation (GBRI): Microsoft Help and Support. 28 August 2007. <<http://support.microsoft.com/kb/872836>>



Biography

PureConferences
www.pureconferences.com

Pure Conference Solution Pvt Ltd., A-108B, Sector 58, NOIDA 201301, India; Tel: +91 120 4621080

Page 10 of 10