Increasing the Effectiveness of Automated Testing

Shaun Smith and Gerard Meszaros
ClearStream Consulting Inc.
1200 250 6th Avenue SW
Calgary, Alberta, Canada T2P 3H7
1-403-264-5840
{shaun|gerard}@clrstream.com

ABSTRACT

This paper describes techniques that can be used to reduce the execution time and maintenance cost of the automated regression test suites that are used to drive development in eXtreme Programming (XP). They are important because developers are more likely to write and run test, thus getting valuable feedback, if testing is as painless as possible. Test execution time can be reduced by using an in-memory database to eliminate the latency introduced by accessing a disk-based database and/or file system. This paper also describes how the effort of test development can be reduced through the use of a framework that simplifies setup and teardown of text fixtures.

Keywords
XP, Automated Testing, Database, Object/Relational Mapping, Test-first Development, Business System, Information System

1 INTRODUCTION

Automated testing is a useful practice in the toolkit of every developer, whether they are doing XP or more "traditional" forms of software development. The main drawing card is the ability to rerun tests whenever you want reassurance that things are working as required. However, for tests to provide this valuable feedback to a developer they have to run relatively quickly. On a series of Java projects building business systems, we have found that running large numbers of JUnit tests against a relational database is normally too slow to provide the kind of rapid feedback developers need. As a result we have developed a collection of techniques, practices, and technologies that together allow us to obtain the rapid feedback developers need even when a relational database underlies the system under construction.

2 DEVELOPER TESTING ISSUES

XP is heavily reliant on testing. Tests tell us whether we have completed the implementation of some required functionality and whether we have broken anything as a result. On an XP project without extensive design documentation, the tests, along with the code, become a major means of communication. It is not just a case of letting "the code speak to you". The tests must speak to you as well. And they should be listened to over and over again.

Slow Test Execution

Having established that we rely so heavily on tests to guide development and report on progress, it is no surprise that we want to be able to run tests frequently for feedback. We have found that it is essential for tests to execute very quickly; our target is under 30 seconds for the typical test run.

A purely economic argument is compelling enough by itself. Assuming a developer runs the test suite every 10 minutes while developing, they will have run the suite 24 times in a single 4-hour programming session. A 1-minute increase in test execution time increases development time by 10% (24 minutes)! But the result of slow tests is even more insidious than the economic argument insinuates!

If test execution is too slow, developers are more likely to put off testing and that delays the feedback. The preferred model of development is the making of a series of small changes, each time running the appropriate test suite to assess the impact of the change. If tests run slowly, developers will likely make a number of small changes and then take a "test timeout" to run the tests. The impact of this delayed use of tests is two fold. First, debugging becomes more difficult because if tests fail after a series of changes, identifying the guilty change is difficult. Developers may even forget they made some changes. Murphy's Law says that this forgotten change will most likely be the source of the failing tests.

Second, if tests take so long to run that a developer leaves her desk while a test suite runs, her train of thought may be broken. This routine of starting a test suite and then getting up for a stretch and a chat was observed regularly on one project. The lengthy test suite execution time was because of the large number of tests and the relational database access required by each test. This combination became a recurring pattern on a number of projects.

Expensive Test Development

Given the test-first approach taken by XP, the cost of writing and maintaining tests becomes a critical issue. Unit tests can usually be kept fairly simple, but functional tests often require large amounts of fixture setup to do even limited testing. On several projects, we found that the functional tests were taking large amounts of time to write and needed considerable rework every time we added new functionality. The complex tests were hard to understand in part because they contained too much necessary setup. We tried sharing previously setup objects across tests, but we also found it difficult to write tests that did not have unintended interactions with each other.

3 POSSIBLE SOLUTIONS

Business systems need access to enterprise data. In traditional business systems, queries are made against databases for such data. In testing such systems, we have experienced slow test execution due to the latency of queries and updates caused by disk I/O. The time to execute a single query or update may only be a fraction of a second, but we have often had to do several or many such queries and updates in each test as text fixtures are setup, tests are performed, and test fixtures are torn down.

Running Fewer Tests

One way to reduce the time required to obtain useful feedback is to limit the number of tests that are run. The JUnit Cookbook describes how to organize your tests so that subsets can be run. Unfortunately, this strategy suffers from the need for developers to choose an appropriate subset of the tests to use. If they choose poorly, they may be in for a nasty surprise when they run the full test suite just before integration. Nevertheless, there is value in these techniques. The key challenge is to know which tests to run at a particular time.

Faster Execution Environment

Another approach would be taken to increase test execution speed by using faster hardware, dedicated IP subnets, more/better indices for the database, etc. However, these techniques tend to yield percentage increases while we needed guaranteed improvements measured in orders of magnitude.

In-Memory Testing

Since the problem preventing the running of many tests is the time required to query or update a database, the ideal solution is to somehow avoid having to do this as much as possible. We have replaced the relational database with a simple in-memory "object database". To obtain rapid feedback from test suites, developers run against the in-memory database while coding. The development process focuses on the use of in-memory testing during development and then switches to database testing before a task is integrated and considered complete.

While our experiences are with relational databases, this approach could also be used to speed up testing with object databases or simple file-based persistence.

4 IN-MEMORY TESTING

The challenges for running tests faster by running them in memory are:

1. How do you eliminate the database access from the business logic? This requires ***Separation of Persistence from Business Logic***, including ***Object Queries***.
2. How do you know which tests can run in memory? We use standard JUnit test packaging conventions and aggregate tests capable of being run in-memory in a separate test suite from those that require a database.
3. How do you specify whether they should be run in memory on a particular test run? This requires ***Dynamic Test Adaptation***.
4. How do you deal with configuration specific behavior? This requires ***Environment Plug-ins***.

Separation of Persistence from Business Logic.

Switching back and forth between testing in-memory and against a database is only possible if application code and tests are unaware of the source of objects.

We have been building business systems using a "Business Object Framework" for about five years in both Smalltalk and Java. The objective of this framework is to move all the "computer science" out of the business logic and into the infrastructure. We have moved most of the "plumbing" into service provider objects and abstract classes from which business objects can inherit all the technical behavior. The technical infrastructure incorporates the TOPLink™ Object/Relational (O/R) mapping framework. TOPLink is primarily responsible for converting database data into objects and vice versa. It eliminates the code needed to implement these data/object conversions (which means no SQL in application code) and it does automatic "faulting" into memory of objects reached by traversing relationships between objects. This eliminates the need for having explicit "reads" in the business logic.

In essence, TOPLink makes a JDBC-compliant data source (such as a relational database) look like an object database. Our infrastructure makes persistence automatic, which leaves developers to focus on the business objects in an application and not on the persistent storage of those objects. By removing all knowledge of persistence from application code, it also makes in-memory testing possible.

This approach is described in more detail in [4].

Querying (Finding objects by their attributes)

If applications access relational databases, querying using table and column names, then replacing a relational database with an in-memory object database becomes problematic because an in-memory database contains objects, not tables. How do you hide the different sources of objects from the application, especially when you need to search for specific objects based on the values of their attributes?

Solution:
Object Queries—All queries are performed against the

objects and their attributes, not against the underlying database tables and columns. TOPLink's query facility constructs the corresponding SQL table/column query for queries specified using objects and attributes. Application code is unaware of where and how objects are stored—which provides for the swapping of the "where and how" between an in-memory and a relational database.

Our initial approach to querying was to move all database queries into "finder" methods on Home (class or factory) objects. But to support both in-memory and database querying we had to provide two implementations of these finder methods: one that used TOPLink's query facility against a relational database and the other that used the API of the in-memory database to perform the query.

We quickly tired of having to implement queries twice and have now implemented support for the evaluation of TOPLink's object/attribute queries against our in-memory database. With this technology, the same query can be used to find objects in our in-memory object database or translated into SQL to be sent to a relational database.

Configuration Specific Behavior
When testing in memory, how do you handle functional features of databases like stored procedures, triggers, and sequences?

Solution:
The functional features of databases can be implemented in memory by ***Environment Plug-ins*** (*Strategy* [3])**.** Each function provided by the database has a pair of plug-ins. When testing with a database, the database version of a plug-in simply passes the request to the database for fulfillment. The in-memory version of the plug-in emulates the behavior (side effects) of the database plug-in.

In a relational database, a sequence table may used to generate unique primary key values. The in-memory version of the plug-in keeps a counter that is incremented each time another unique key is requested.

Stored procedures can be particularly problematic when building an interface to an existing (legacy) database. On one project, we had to create an account whose state was initialized by a stored procedure. During in-memory testing, the state was not initialized so business rules based on the account's state would fail. We could have added code to set the state during the account object initialization but we did not want to have any code specific to testing in the production system. We were able to avoid this using an `InMemoryAccountInitializationStrategy` that performed the required initialization during in-memory testing and a *NullObject* [5] that did nothing when the database's stored procedure initialized the state.

Because it is possible that an in-memory plug-in behaves differently than the database functionality it replaces, it is still necessary to run the tests against the database at some point. In practice, we require a full database test before changes are permitted to be integrated.

Environment Configuration
How and when do you setup the test environment with the appropriate *Configuration Specific Behavior*? How do you decide which environment to use for this test run?

Solution:
Dynamic Test Adaptation – We use Test Decorators to specify whether the test environment should be in-memory or database. Using this technique we can choose to run a test in memory for maximum speed or we can run the test against the database for full accuracy. One hitch is the fact that one can choose to run all the tests, just the tests for a package, just the tests for a class or just a single test method. To ensure that the tests run in the right environment regardless of how they are invoked, we have added methods to our `TestCase` base class that push the decorators down to the individual test instance level.

On a recent project we created two Java packages: one containing tests that could only be run in memory (because the O/R mappings were not yet complete), and one for multi-modal tests (tests that could be run either in memory or against a database.) As the project progressed, tests were moved from the in-memory test package to the multi-modal test package. Combining this organizational scheme with ***dynamic test adaptation***, we were able to run the multi-modal tests against either the in-memory or the relational database.

5 OPTIMIZING TEST DEVELOPMENT

The JUnit test lifecycle specifies that one sets up a test fixture before a test and tears it down afterwards. But we found that many of our functional tests depended on a large number of other objects. A test can fail if it depends on a previous test's side effects and those effects can vary depending upon a test's success or failure. We divided the objects used by a test into three groups:

1. Objects referenced but never modified. These **shared objects** can be setup once for all tests.
2. Objects created or modified specifically for a test. These **temporary objects** must be created as part of each test's setup.
3. Objects created, modified or deleted during the test

We made it a hard and fast rule that tests cannot modify any shared objects because to do so makes test inter-dependent, which in turn makes tests much harder to maintain.

Shared Test Objects
In database testing, these shared objects would be the initial contents of database before any testing started. How do you ensure that the shared objects are available in both in-memory and database testing modes?

Solution

In-Memory Database Initializer—For in-memory testing, we define an object who's responsibility is to create all of the objects necessary to replicate the expected static database contents. The test infrastructure delegates to it to ensure that these objects are created before any tests need them.

If there is an existing legacy database, testing can be performed before the database O/R mappings are in place by manufacturing objects in memory that correspond to legacy data and placing them in the in-memory database.

When building applications that require a new database, an in-memory database can be populated with objects created by the *Initializer*. When the database schema is finally defined, the *Initializer* can be run with persistence enabled. The objects created by the *Initializer* are automatically written to the database to create the initial database content.

Temporary Object Creation

At any one time, several developers may be running the same tests. We need to ensure that the tests don't interact either with themselves or with other tests. How can we ensure that newly created objects have unique keys and contain all data required to make them valid? How can you ensure that several instances of the same test being run from several workstations aren't using the same values for keys thus causing transient test failures?

Solution:

Anonymous Object Creation— We created a `TestScenarioManager` that is the hub of all test object creation. Whenever a new kind of object is needed for a test, a `createAnonymousXxxx()` method is added (with any arguments required for customization.) These methods create a fully-formed object that may be used in tests without worrying about details like unique-key constraints and unintended test interactions. Inherited methods generate identifiers that are guaranteed to be unique.

Temporary Object Cleanup

Tests may create many new objects. Depending on where a test failed, the objects to be cleaned up could vary significantly. How can you ensure all the temporary objects are cleaned up properly without having to write complex teardown logic?

Solution:

Automatic Fixture Cleanup –To simplify teardown, each `createAnonymousXxxx()` method registers the newly created object as a transient object that needs to be deleted. Each test inherits a teardown method from our `TestCase` base class that automatically deletes all the registered test objects. This eliminates the need to write any test-specific teardown code. The test need only ensure that any objects it creates as part of the testing logic (as opposed to fixture setup) are also registered for automatic removal.

6 RESULTS

Reduced Test Execution Time

We have been able to run test suites of up to 260 tests in under a minute when running against an in-memory database. Those same tests run orders of magnitude slower when running against a relational database. On a project we are currently involved with, 53 tests are executing in memory in a time of around 10 seconds while the same 53 tests running against an Oracle database have an execution time of about 10 minutes. This is not surprising given the relative cost of memory access (measured in nanoseconds), compared with the cost of disk access (milliseconds.)

Reduced Testing Code

Through continuous improvement of our testing infrastructure, we have reduced the average size of our tests by 60%. Space does not permit inclusion of examples, however; these can be found on our website [4]. We estimate that this translates into an effort reduction of 70%. We also suspect that the quality of testing has improved but this is hard to measure directly.

7 CONCLUSIONS

The effectiveness of a test-first development process is inversely proportional to the execution time of the tests. The execution time of each test can be reduced by orders of magnitude by removing the latency introduced by disk-based I/O. This can be achieved by replacing the disk-based database with an in-memory database. This is most easily done if the application logic works exclusively with objects rather than interacting with the database via SQL. Test development and maintenance effort can be reduced significantly through improvement of the testing framework. There are a number of issues but each is surmountable.

8 ACKNOWLEDGEMENTS

The authors would like to thank the colleagues who provided comments on this paper and especially the clients who provided the opportunities for these experiences.

9 REFERENCES

1. K. Beck, E. Gamma, "*Test Infected: Programmers Love Writing Tests*",, Available on-line at: http://junit.sourceforge.net/doc/testinfected/testing.htm

2. E. Dustin, J. Rashka & J. Paul, *Automated Software Testing*, Addison Wesley, ISBN: 0-201-43287-0

3. E. Gamma et al, "*Design Patterns; Elements of Reusable Object-Oriented Software*", Addison Wesley, ISBN 0-201-63361-2

4. G. Meszaros, T. O'Connor, S. Smith, "Business Object Framework", *OOPSLA'98 Addendum*, available online at http://www.clrstream.com/papers.html

5. B. Woolf, "Null Object", *Pattern Languages of Program Design 3,* Addison Wesley, ISBN 0-201-31011-2 pp. 5-16