Test-Driven Porting

Ralph Bohnet
ClearStream Consulting
3710 205 5th Ave. SW
Calgary, AB
T2P 2V7 Canada
1-403-264-5840
ralph@clrstream.com

Gerard Meszaros
ClearStream Consulting
87 Connaught Dr NW
Calgary, AB
T2K 1V9 Canada
1-403-560-2408
gerard@clrstream.com

Abstract. Traditional Test-Driven Development focuses on development of new units (classes) driven by programmer-facing unit tests. This paper describes our experiences when using business-facing tests (also known as "story tests") to guide the porting of a legacy application. Domain experts specified tests in a tabular format using Excel spreadsheets. Developers automated these spreadsheets in various ways over time: scripts, generation of JUnit source code, and Fit. These tests were run against the legacy system and guided the development of the newly ported system. We found test-driven porting to be an effective way to port a complex application.

Keywords: Porting legacy applications, Test-Driven porting

1 Introduction

Test-Driven development is a software productivity enhancing technique that claims to improve the quality of code and reduce unused code. In Test-Driven development the programmer writes an automated test before writing the code needed to make the test pass. Three steps are repeated over and over: write an automated test, write enough code to make the test succeed, and clean up the code. This sequence is known as "Red, Green, Refactor". It allows the programmer to focus on only writing code for what is needed to pass the test, thereby reducing the potential waste of adding extra features that are not needed. Test-Driven development typically applies to programmer unit tests. This process is described in detail in the various books on Test-Driven development [1].

Traditional Test-Driven development focuses on development of new individual units (classes) driven by unit tests. This paper discusses the experiences and lessons learned when using business-facing tests (AKA Story Tests) to drive the porting of a legacy application.

1.1 The Legacy Application

The legacy application was a complex billing system written in VisualAge Smalltalk during the late 1990s for a company involved in the energy transportation business. It calculated charges based on the allocation of transported energy to customer contracts and produced a monthly invoice for each customers' account. The main driver leading to the decision to port the

application was to replace obsolete technology that made the application expensive to support. Other factors included the need to accommodate minor business changes and to integrate with newer applications.

Support Issues

The legacy application used Toplink for Smalltalk to provide the relational/object database mapping but the Toplink vendor (Oracle) had discontinued support for this version of Toplink. Toplink for Smalltalk only worked with Oracle (8i) while the current company-wide standard was Oracle 9. This forced IT to support two different versions of Oracle in production.
The legacy application was complex with over a dzzen algorithms for charge calculations, and was comprised of approximately 1000 classes (140K lines of Smalltalk code). No automated regression tests existed, which increased the cost of introducing changes. In addition, it was difficult to recruit support staff with Smalltalk skills.

Business Changes

The company's business and government regulations had changed since the billing application was originally written. The impact of proposed business changes to the system implied restructuring it and making significant changes to the Smalltalk code base. The business was reluctant to endorse development of an entirely new application. Their stategy was to reengineer/port the application to replace the legacy technology as the first step before introducing business changes.

Integration and Performance Objectives

Since the application was built, a number of new systems and technologies had been introduced. Direct integration with these systems had been avoided, since there were no business benefits to support the integration. "Tactical" solutions had been used to minimize the impact on this application. Specifically, the objectives were:

- Direct integration with a new contracting system, replacing the "tactical" interface that provided the contract and rate data.
- Integration with the company-wide security model to eliminate the need for duplicate login.
- Improving throughput performance of the monthly billing run. To complete a monthly billing run took 12 hours.

2 Project Description

The overall project strategy that addressed the above issues was to deliver the reengineered system in multiple releases grouped into two major phases:

- Phase 1 was to port the existing Smalltalk code to Java. We estimated 6 months to complete phase.

- Phase 2 was to make business enhancements to the newly ported system. This phase could include multiple releases.

In Phase 1, our plan was to port the functionality incrementally and to regression test each increment of functionality by comparing the results with those produced by the existing Smalltalk system. Several small enhancements were also included in phase 1, so new functional tests had to be developed for them. We would replace the technology stack as required:

- Replace the fat desktop client with a browser-based user interface.
- Upgrade to latest company-wide technology stack: Oracle 9, J2EE, Toplink for Java.
- Replace the custom-built invoice rendering logic, based on an obsolete, proprietary reporting engine, with new PDF invoice generation logic.

The functionality to be ported was organized into "features" such as "Basic Charge Generation", "Prior Period Adjustment (PPA)" and "PPA on PPA". These features were then selected for an iteration based on which feature dependencies had already been satisfied and which ones had tests ready to go. Each iteration resulted in one or more features passing its functional tests.

The software development team consisted of 4-6 experienced Java developers and an average of 1.5 domain experts. The actual users of the system were available for discussions on an as-needed basis. The project was organized around 2-week development iterations, with a progress discussion and demo to the stakeholders at the end of each iteration. There were daily stand-up meetings for project team members.

3 Testing Challenges

The challenge was to ensure the business logic in the legacy system was ported correctly. The legacy application was difficult to test. No automated regression tests existed. All previous testing was done manually. There was no façade layer surrounding the domain model that a test could interact with. The fat client GUI was developed in the proprietary VisualAge for Smalltalk workbench (replace with correct name of the framework.) We discovered that testing tools and frameworks for executing the same test on 2 different systems do not exist.

4 Test-Driven Porting Technique Explained

Our approach to solving these challenges was to develop a suite of regression tests to run against the legacy application that identified the system behavior that needed to be ported.
The same tests would run against the new Java system as it was being ported. Usually at the beginning of porting some functionality, the programmer and domain expert would review the relevant domain concepts and the tests. This allowed the development team to learn the business domain and to use a common, ubiquitous language [2]. Next, the programmer would generate a JUnit test (more on this later) based on the spreadsheet. The programmer would find the relevant starting point in the Smalltalk code and begin to follow the call graph, translating into Java only what was needed to pass the test. This process would continue until the Java code compiled, and

the test could run. When the relevant tests passed, the ported functionality was considered complete and delivered to the tester. Figure 1 illustrates this concept of the tests driving or guiding the porting effort.

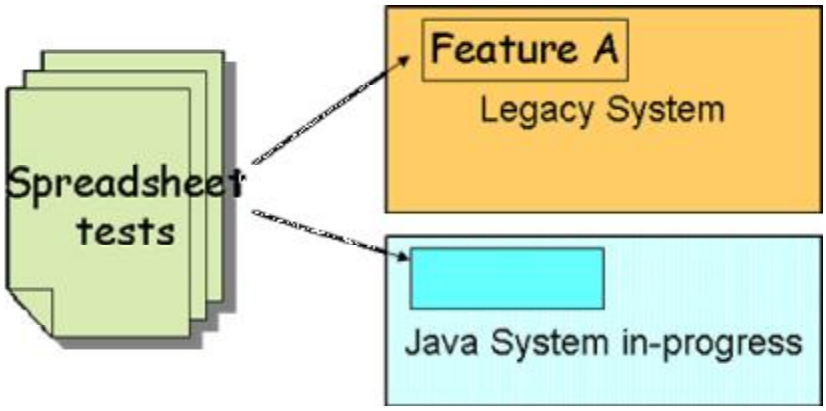

Figure 1. Spreadsheet-based tests guide the porting of each feature in the legacy system

A detailed discussion how the domain experts specified tests and how programmers automated these to run against the legacy system and the newly ported system is described below.

4.1 Domain experts specify business-facing tests in spreadsheets

The domain experts specified tests in a tabular format using Excel spreadsheets before the porting effort began. These spreadsheets were business-facing examples of what the system should produce (usually an invoice) under certain conditions. Domain experts were already adept at using Excel spreadsheets to specify calculated billing outcomes. The initial format of the spreadsheets followed an approach in a published article [3]. Figure 2 provides an example of a spreadsheet containing a test scenario. The test name and a brief description of the test conditions appear at the top of the spreadsheet. The *Prerequisite* section identified tests to run prior to the execution of the current test. The *Preconditions* section identifies the data required to run the test. The *Processing* section contains the user goal or event to initiate some interaction with the system. The *ExpectedResults* identifies the results the system produces in one or more tables.

UseCase	**Generate Schedule Charges**
TestScenario	**FT-R Service**
Description	Current Billing Period Charges DMD Charge is calculated for a contract that is in effect for part of the billing period DMD Charge is calculated for Price Point A

PreConditions

Customer			
ID	**Name**	**Mnemonic**	**TerminationDate**
2001	FT-R Test Customer	FTR	31-Dec-9999

Processing

GenerateCharges	
EnterpriseId	**Status**
2001	Complete

ExpectedResults

CustomerInvoiceForBillingPeriod			
BillingYear	**Month**	**Customer**	**InvoiceCreated**
2003	12	FTR	Yes

CoverPageInvoiceTotals	
TotalDescription	**Amount**
TOTAL BEFORE GST	2,422.25
GST	169.56
INTEREST ON PAST DUE AMOUNTS	0.00
TOTAL AMOUNT($ CDN)	2,591.81

Figure 2 - Example of business-facing test specified in Excel spreadsheet.

4.2 Legacy system results become the standard for the ported system

After running each test, the legacy system results were captured and became the standard for the newly ported system. The development team created a custom Excel macro that generated a XML representation of the test. Using XSL style sheets, developers transformed this XML representation into SQL statements that would be used to populate the database during the *Preconditions* step of the test.

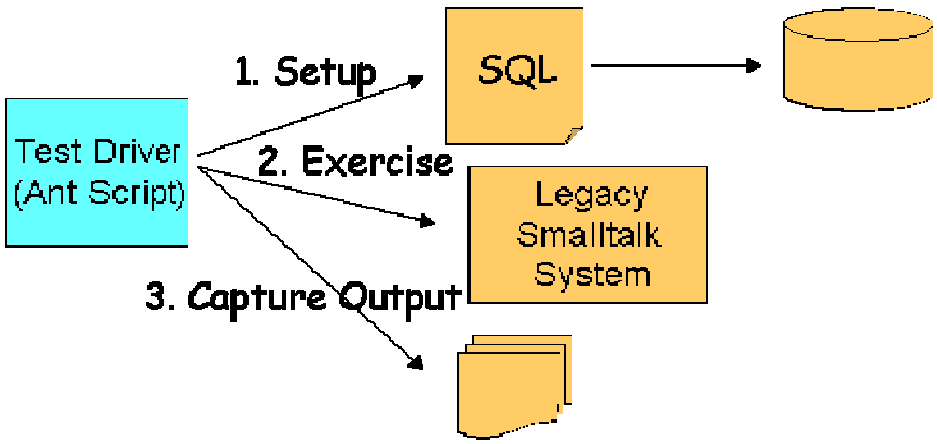

Figure 3 Capturing results from spreadsheet test

Developers wrote a test driver using ANT [4] that would feed the legacy system with input from the spreadsheet test, run the legacy system, and capture a database snapshot. This is illustrated in Figure 3. The tester would then run the same test against the ported system using the test driver and capture the database snapshot. Both snapshots were normalized to remove any system-generated unique keys that might differ between the legacy and ported systems. Then tester would compare the two database snapshots and log any differences between the two systems as defects in the ported system. This comparison would occur after each increment of functionality.

Although this method was used throughout the porting effort, some disadvantages were soon discovered. It was time-consuming to run both the systems and compare the results. Some manual intervention was required when comparing the database snapshots and also PDF files, which made it difficult to run by developers inside of their development IDE or an automated build process. However, the QA/tester used this approach throughout the project and it identified numerous defects in the ported system.

4.3 Automating spreadsheet-based tests by generating JUnit tests

Developers decided to hand-script some of the spreadsheet-based tests using JUnit to make it easier to run the test during the porting effort. After writing two or three tests manually, it was felt significant time could be saved if the JUnit tests could be generated directly from the spreadsheets instead of hand-scripting them. After approximately three man weeks of effort, we implemented a JUnit test generator that used POI [5] to read the spreadsheets and generate the corresponding JUnit test source code. These generated JUnit tests became part of the regression tests for the delivered ported system. As the spreadsheets are a kind of "domain specific testing language", the test generator had to be evolved throughout the project as new kinds of elements were added to the test spreadsheets for specific features.

The development team could now run these generated JUnit tests within their IDE and receive immediate feedback on their porting changes. These test suites became part of the check-in and build process. The automated build process also detected changes in the spreadsheets and generated the corresponding new JUnit tests. The benefit of automating over 300 individual spreadsheet tests for the developer certainly outweighed the cost of building a JUnit source code generator.

4.4 Automating spreadsheet-based tests using Fit

After we completed the port and started the next phase, we replaced the JUnit generated tests with Fit [6]. Running the generated JUnit tests required a Java IDE and the error messages were stack traces, geared to programmers. Hence, the domain experts or testers who created the spreadsheet-based tests did not run these tests. We wanted to provide the ability for domain experts and testers to run the spreadsheet tests themselves and understand the test results. Fit provides a visual color-coded view of the each output test result in a similar format as the original. FitNesse [7], which is built on top of Fit, provides the ability to easily select a single

test or a suite of tests and run them against the system inside of a Wiki. One advantage of this approach is the domain experts / testers can run a spreadsheet-based test against the system and detect errors in the spreadsheets by themselves without involving a developer. Automating the spreadsheet-based tests using FitNesse required minor changes in the spreadsheets, a custom FitNesse plug-in to display the spreadsheet file names and to translate the spreadsheet into HTML, and custom Fit fixtures to act as the glue code between the spreadsheet and the system.

5 Lessons Learned

Porting complex business logic is worthwhile

The legacy system contained complex algorithms and calculations in the business logic layer. Porting this layer rather than rewriting it from scratch proved to be a successful approach. It meant earlier delivery, lower risk, reduced cost, and consequently a greater return on investment. The parts of the application that had to be rewritten because of obsolete technology included the user interface and the PDF invoice generation logic. For the most part, there was very little business logic in these areas. Where business logic did exist, we moved that logic into the business logic layer rather than recreate it in the presentation layer.

Test-driven porting is an effective way to port code

Having an existing application as a reference point is valuable. However regression tests for the legacy application must be created or must exist in order to drive the porting effort. Without these tests, it is impossible to know what logic exists, what is current and what should be ported. This suite of automated regression tests became the safety net for developers during the second phase when significant business changes were made to the system. The maintenance staff also utilized these automated regression tests as they fixed bugs.

Test-driven porting results in less code to maintain because it leaves behind unused code

The legacy system had significant amounts of unused code. Some of the unused code probably existed from day of the original system and some of the unused code resulted from business changes over many years. Support staff were reluctant to cleanup unused code due to lack of time, resources, and no safety net of automated regression tests. If the tests did not invoke parts of the legacy code, it was not ported. As a result, there is much less code to maintain in the ported system.

Tests can have defects also.

Initially, developers assumed the tests were correct. When they couldn't get the tests to pass they spent a lot of time digging through the Smalltalk and Java code trying to find porting errors. Then, they would finally go to the domain expert who would look at the test error and immediately say, "Oh, the test is wrong. It should say …". We learned that writing correct tests

is almost as hard as writing correct code and that it was best to involve the domain expert as soon as we had any test failures that we didn't understand. Providing the domain experts / testers the ability to run the spreadsheet-based test against the system would have removed many of the test defects even before involving the developer.

JUnit/xUnit not most effective tool for business-facing tests.

JUnit, like other members of the xUnit family of unit test automation frameworks, raises an error and abandons execution of the current test when it encounters the first failure. Some tests had numerous errors in the expected results. This "stop on first failure" behavior was not very effective because developers would fix the bug in the test and re-run it, only to discover another error further down in the test.

FIT [6] is better suited to business-facing tests because it continues the test execution until completion and provides a visual color-coded view of the each output test result. This shows all the errors in a test run making it easier to see patterns in the failures. For example, each test used a different set of customer names to prevent conflicts. Many of the tests were built using a "clone & twiddle" approach. A common test bug was that some of the customer names were missed during the "twiddle" phase. So any expected results that referenced the old customer name were marked as failures. This was highly obvious when you could see the errors all at once.

6 Project Outcome

The ported Java system was successfully put into production on schedule with no significant bugs. The end users gave the system a 100% customer satisfaction mark. We are now working on Phase 2, which involves enhancing the system to support changes to the business.

7 Conclusions

Porting rather than rewriting complex business logic is worthwhile if you have a well-designed application. Having an existing application as a reference point is valuable, but regression tests are necessary to drive the porting effort. Without regression tests, it is difficult to know that logic exists or is current. Test-driven porting is an effective way to port the code. It results in less code to maintain because it leaves behind unused code. JUnit/xUnit is not most effective tool for business-facing tests. FIT [6] is better suited to these tests, because test execution continues until completion and provides a visual color-coded view of the each output test result.

8 Acknowledgements

We'd like to thank our client who graciously allowed us to write about this experience. The work described in this paper was significantly improved by the contributions of Lynne Ralston, Jennitta Andrea, Geoff Hardy, and Shaun Smith.

9 REFERENCES

1. Beck, Kent.Test-Driven Development:By Example, Addison-Wesley, 2003; ISBN 321-14653-0
   Astels, David. Test-Driven Development: A Practical Guide, Prentice Hall, 2003; ISBN 13-101649-0

2. Evans, Eric. Domain-Driven Design, Addison-Wesley, 2004; ISBN 321-12521-5

3. Andrea, Jennitta, "Generative Acceptance Testing for Difficult-To-Test Software", XP2004 Conference, 2004

4. ANT, http://ant.apache.org

5. POI, http://jakarta.apache.org/poi

6. FIT, http://fit.c2.com/

7. FitNesse, http://www.fitnesse.org