Retro-fitting Unit Tests with JUnit

by

Kevin Rutherford

Motorola UK

ABSTRACT
This paper documents the experiences to date of a small Java development team, who adopted a new unit testing regime in the midst of an ongoing medium-sized project. The team chose to develop the tests using an open-source framework called JUnit.

KEY WORDS
Stingray, Java, unit testing, module testing, regression testing, JUnit, make, JNDI, IP reuse.

1. Introduction and Background

This paper describes the experiences of a small Java team that attempted to retrofit a new software testing regime during the course of an ongoing development project. The paper emphasizes the impact of adding new tests to a live software development project, and is not intended to provide a detailed description of the JUnit testing framework or its underlying philosophy. The action described below took place throughout the last half of 1999 and continues today.

Stingray is a strategic software programme designed to produce an integrated end-to-end silicon design environment. It provides a metadata environment which promotes and encourages intellectual property (IP) reuse, and in which existing design tools and flows can be executed. Design tools are "encapsulated" by writing wrapper code that bridges the tool's expectations into the metadata environment of Stingray.

The project described in this paper is Stingray version 1.0, which comprises a Java framework called Corvette, together with an extensible suite of design tool wrappers known as "encapsulations", also written in Java. The Corvette framework provides the run-time environment for design tools and the metadata-based user workspace. The metadata service is provided by a module that implements the standard Java Naming and Directory Interface (JNDI); Corvette is therefore independent of the technology used to store and retrieve IP. Corvette also provides a range of utility classes for use by the encapsulation writer.

Stingray 1.0 is targeted to run on Solaris 2.5-7 and HP-UX. The software is being developed under Windows/NT using the JDK version 1.1.8 (JDK2 does not yet exist on HPUX-10). A small portion of the system (< 100 loc) is written in C++ and accessed via the Java Native Interface (JNI). The software build environment is based on Unix-style make, and uses the GNU Cygwin32 Unix emulation tool-kit.

The development team consists of 5 developers averaging about 1 year's experience of Java and totalling 45 years software development experience. The author joined the team in August 1999 and was given the task of fixing and completing the JNDI service provider module. The module already contained 40 classes and about 2000 lines of Java code. Some of it worked as required, some of it was incomplete, and some of it was complete but had bugs in it. There were two test cases, written in JPython, one of which didn't run.

We chose JUnit as our testing framework because the author had used it successfully before, and because the team had previously had little success with *ad hoc* approaches to unit testing. Our objective was to improve the quality of our software and of our development process.

In round numbers, this story begins with the system comprising about 180 Java classes in 15 packages, with half a dozen *ad hoc* unit tests written in JPython or embedded in makefiles. At time of writing the system comprises around 350 classes in 25 packages, with 85 tester classes implementing 350 test cases.

1.1 The JUnit Philosophy

JUnit [Beck98, Beck+99, Fowler99] was developed by Kent Beck and Erich Gamma as a Java version of Beck's successful Smalltalk testing framework. These test frameworks form a key part of the *eXtreme Programming* (XP) [Beck99] approach to software development, which has some implications for how tests are organised:

As seen from the outside, the functionality of a system such as Stingray 1.0 can be broken down into a (probably large) number of *user stories* (think of function points or, if you're used to use cases, think scenarios). The totality of these stories capture the entire system requirements – including the non-functional ones, such as distribution, performance, resilience etc (because these all impinge on the user in some way).

Therefore it seems reasonable that the primary set of tests for the system should be written in response to these stories. Thus each User Story will correspond to one or more System Tests. In XP these are (notionally) "owned" by the user/customer, because they demonstrate to him that the developers have fulfilled their contract (and that the system does everything he requested).

Internally, unseen by the user, programmers will divide the system into a number of "modules". Often a module will play a part in supporting more than one story, and conversely most stories will involve the functionality of several modules. (Thus modules and stories form orthogonal dimensions through which the developer and user see the system.) To speed development, each module should be accompanied by a number of tests (called Unit Tests) that demonstrate that it does the job it was designed to do.

These two testing dimensions are independent. Strictly speaking, only the System Tests are "required", while the Unit Tests significantly improve the speed and quality of development and support. In this paper, we concentrate on our efforts to retro-fit a set of Unit Tests for the Stingray 1.0 system.

With the XP approach, the Unit Tests are all completely automated. Anyone can run the unit tests for any part of the system, without needing to understand or interpret cryptic messages and logfiles. If any test fails, the programmer gets simple output of the form:

```
..............F......
22 tests, 1 failure, 0 errors
Failures:
1) ToolProcessTester.test_runDir: expected "hello" but was "goodbye"
```

JUnit's central abstraction is the `TestCase` class [Beck+99], which implements a `Test` interface. Each test case must be implemented by extending this class, which provides a range of methods for making assertions and recording the results. The JUnit framework executes a single `Test`, by calling its `setUp()` method, then running the test, then calling its `tearDown()` method. Thus each test instance creates and destroys its own fixture as required. A `TestSuite` is an alternative implementation of the `Test` interface, and forwards its `setUp()`, `runTest()` and `tearDown()` methods to the tests it contains. Thus tests can be organised into suites of arbitrary size.

In what follows, a "test" is a Unit Test, unless otherwise stated.

2. Experiences with JUnit

The first significant step we took was to decide that any tester class we wrote would be a member of the same package as the class under test. This arrangement had the major advantage that the tests could call package-private methods in the tested class, or indeed on any other class in that package. We could therefore, if necessary, provide methods specifically for use by the tests, without those methods becoming visible to our client projects.

We now checked JUnit into our source control system, so that it became an integral part of our build system.

The next task was to provide a simple way for developers to run JUnit on their current package version. Based on early experiments in unit testing the JNDI package, we declared that the tests should be run by invoking

```
java currentPackage.PackageTest
```

Thus we invented a local standard (the first of many!), stating that the complete set of unit tests for all the classes in a package must be in a class called `PackageTest`, which must have a `main()` method that invokes JUnit. These main() methods constituted a large number of copies of the same (small) piece of code; more on this later.

2.1 Integration with Make

At this point we also wished to add a "test" target to every existing makefile (one in each package source directory). Later, any new package would also require a makefile with an identical target, and since our makefiles were fairly standardized anyway, we replaced all of the makefiles by a single makefile generator called GenMake. GenMake creates a makefile in each Java package directory, and includes in it the desired "test" target. Now the unit tests for any package can be run by typing

```
mk test
```

and those for all packages below "here" can be run by

```
mkall test
```

This latter command simply recurses through the directory hierarchy calling 'mk test'. It makes no attempt to collect the unit tests it finds into a single suite. This means that the developer has to check the screeds of output for every line of the form

```
...............F......
```

looking for test failures. More on this later.

2.2 Growth and Complexity

This simple infrastructure allowed us to begin writing unit tests in earnest. Where sporadic unit tests already existed (usually written in JPython), these were quickly converted into a Java `PackageTest` class. Elsewhere, as developers got used to JUnit, a few additional `PackageTest` classes were written, particularly in parts of the system that were undergoing complex changes.

No specific test-writing activities were scheduled. The major impetus for adding a new unit test was when a bug was discovered in some part of the system that had been thought to be working correctly. Initially these tests were always added to the package's `PackageTest` class, which quickly became unwieldy as a result. After an initial spurt, progress adding new tests slowed, due mostly to the fact that many `PackageTest` classes were now too complex, and therefore too daunting to be easily extended.

Progress was fastest in the JNDI package, since the author had some prior experience of working with JUnit. Given the need to document the status of the existing code, we began writing unit tests to check that the JNDI implementation met the assertions in the specification.

Following [Fowler99] we created a `Tester` class for every class in the package's public API, and had the `PackageTest` class group these all together into a JUnit test suite, thus:

```
public final class PackageTest {
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestSuite(
        DistinguishedNameTester.class));
    suite.addTest(new TestSuite(
        ContextTester.class));
    suite.addTest(new TestSuite(
        ContextFactoryTester.class));
    suite.addTest(new TestSuite(
        SymbolicLinkTester.class));
    //...
    return suite;
  }
}
```

For each public feature in the API we added a corresponding `test` feature to the `Tester` class. Thus `context.bind()` is tested by the new method `ContextTester.test_bind()`. This general scheme is illustrated in Figure 1.

(This technique utilises a reflection-based feature of the JUnit `TestSuite` class, which will construct a test case instance whenever it finds a method whose name begins with `test` in a subclass of `Test`.)

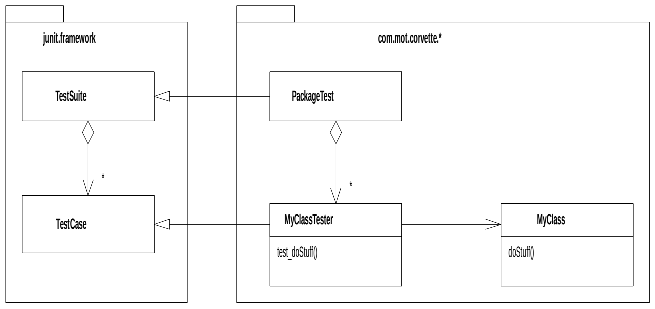

Figure 1 – Organisation of tests for MyClass

With only a few days' work we had a comprehensive list of passes and failures, and hence an automatically-generated work-list for fixing and completing the JNDI package. And as the package later came to be extended, these tests remained, serving as regression tests and providing confidence that the modifications had not broken any existing functionality.

A new practice started to emerge at this point. At the end of a development episode, the developer would run the entire set of unit tests before integrating his changes to the master sources. This gave him a simple way to check that he hadn't broken any other part of the system, thus boosting confidence and increasing the speed at which he was willing to proceed. However, `mkall test` can be slow, and the test reports can get lost in the reams of output generated by make's recursion through the directory tree.

2.3 Testing for Portability

Our product is targeted to run on various flavours of Unix, including Solaris 2.5, 2.6 and 2.7, but our developers all work on Windows/NT. At some point we began receiving bug reports on parts of the system that had tests, and for which the tests all passed. It turned out that certain aspects of the Java `File` class, for instance, behave differently on different platforms and on different filesystem types, so we found we needed to run our tests in the target environments, as well as in the development environment.

We therefore created an hourly batch job that built the entire system from scratch and then ran all the unit tests on the result. If the output contained any test failures these were immediately emailed to every member of the development team.

Now, if anyone integrated code that wasn't portable, the batch job discovered this and the problem could be quickly fixed.

2.4 Imposing Order

As the number of tests grew, we now started to find that the test classes themselves were becoming a little intrusive: in the source directory for some Java packages it had become difficult to distinguish production code from test code. This was exacerbated in cases where the tests required the development of additional utility classes (eg. a `TempFile` class that generates random test files and cleans up the filestore afterwards).

To help distinguish test code from production code, we took our second significant step: we adopted Fowler's [Fowler99] naming convention throughout the system. That is, the unit tests for a class `MyClass` should be in a class called `MyClassTester`. Following the approach used earlier for the JNDI tests, we went around and split up the other `PackageTest` classes into `Tester` classes. The small hit in development time was immediately repaid, as the unit tests were now less daunting again. Test coverage again started to increase steadily.

That helped to some extent, but we had another problem: As we approached the date of our first Alpha release, we needed a slick (ie. automated) way to collect together the production classes into a Jar (a zipped archive containing Java class-files), without including any test code in the release. We could now distinguish `Tester` classes, but is `TempFile` part of the product or part of the tests?

Here we took our third, and most radical, step: we created two parallel package trees in our source repository, one containing only production code and the other containing only tests and their support classes. Thus a package such as `com.mot.stingray` now appears as two filestore directories …`/packages/com/mot/stingray` and …`/tests/com/mot/stingray`. We then altered our makefile generator so that, in the …`/tests` package hierarchy, the java compiler and run-time had both …`/tests` and …`/packages` on their classpath (figure 2). Now these tools would see the two package trees as if they were overlaid. The end result is that the tests build with the production code in sight, while the production code cannot see the tests and is not contaminated with test class files. It is now a trivial matter to collect the production code (everything under …`/packages`) for release, and we have the added bonus that production code cannot accidentally refer to a test class.

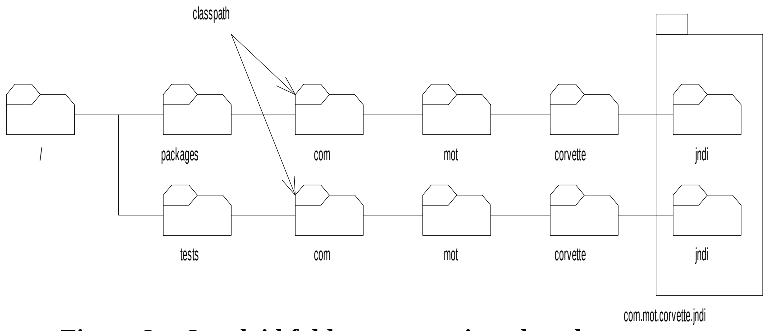

Figure 2 – Overlaid folders create virtual packages

2.5 Stability

Our development environment now suddenly seemed delightfully uncluttered, and to date we have made no further major changes.

As we continued to develop new tests, we discovered certain test utility classes cropping up time and again. We created a `test_support` package in the …`/tests` tree to hold these. This package now also seems to have become stable, perhaps indicating that our current unit tests span a large proportion of the range of ways to test our product.

One day we found we needed to issue an emergency patch release of one of our libraries. The library contains native C++ code and has a Java JNI wrapper to make it usable by the Java code in the rest of the system. One of our customers wanted to run on a platform we had never built on or tested on. Once we had access to such a machine, we quickly built the C++ library. Then we ran the unit tests for its Java wrapper – one failed because a compiler option in the makefile was incorrect for that platform. We fixed it and ran the unit tests again, and this time they all passed. Next the unit tests for the whole system – they passed. Finally, we ran the few system tests we currently had, which we knew exercised this library – they all passed. The whole episode had taken only an hour or so, yet we could think of nothing more that might go wrong. We issued the patch library and we have heard of no problems since.

At this point our test coverage seems to have stabilized at around 25% (that is, we have a `Tester` class for 1 in 4 of our production classes). This seems low, but in fact each test naturally exercises a class and its close collaborators. We therefore estimate that the code coverage through our unit tests is perhaps as high as 60% now.

The following table charts our progress in developing unit tests for the Corvette system:

Date	**Product Classes**	**Tester Classes**	**Test Cases**
1-Sep-1999	260	6	6
1-Oct-1999	312	6	6
1-Nov-1999	289	42	149
1-Dec-1999	297	57	221
1-Jan-2000	319	50	372
1-Feb-2000	325	52	285
1-Mar-2000	337	69	360
1-Apr-2000	344	85	428

The apparent decrease in system size during October – and the corresponding rapid increase in test numbers – reflects the introduction of the `Tester` naming convention and the conversion of other existing tests to JUnit. Since then, progress has been relatively steady, with the number of tests always rising faster than the size of the product.

3. Plus Points

This section describes some of the positive benefits of the approach we took to retrofitting unit tests in the Stingray development. Note that many of the benefits we discovered also reflect advantages claimed for XP [Beck99].

The unit tests serve as excellent documentation. A complete set of tests for a class is often a very good starting point for someone looking at that class for the first time and attempting to discover what it does. And because we know that the tests all compile and all pass, the reader can have more confidence in them than in source code comments, say, which may not have been maintained.

Well-named test cases can also communicate the design intent of a class. For example, we eventually learnt that a test case called `test_export()` could be replaced by smaller test cases called `test_export_0files()`, `test_export_1file()` and `test_export_nfiles()`, giving a clearer picture of the algorithm's scope and boundary conditions.

Separating the production source from that of the tests proved extremely worthwhile, and greatly simplified a number of processes that had become intricate or cumbersome.

The introduction of simple naming conventions for test classes and methods has gone some way towards generating "egoless" test source, thus making it easier for all team members to contribute to most parts of the system. The naming conventions have also made possible the generation of metrics showing our progress with testing, since it is now quite easy to count classes whose names end in "Tester" or methods whose names begin with "test".

Development tasks are generally finished more quickly in parts of the system that have comprehensive tests. The tests tell the developer when he is done. This is especially true when one adds new tests at the same time as one adds new functionality.

In those areas of the system in which large numbers of tests have been added, architectural change now seems less daunting – the tests give a full and precise report of one's progress. For instance, in the JNDI sub-system we were able to replace a single large class with six smaller classes implementing the Composite pattern [Gamma+94] surprisingly easily. After each refactoring step we ran the tests, and those that failed quickly told us where we had made an invalid assumption. The entire process took only a couple of hours – and we knew we were done because the tests all passed!

With our hourly batch test run, we often know immediately when a bug has been introduced into the product. As soon as the emailed failure report goes out, we quickly check any integrations done during the last hour (usually just one), and we know that that new code must have introduced a problem. Bugs are thus caught quickly, and the quality of the system remains high. This is only possible due to the fully automated style of testing encouraged by XP and JUnit.

Another positive side-effect of this is that we also know at all times that the system has high quality. So if we need to quickly issue a patch version for any reason, we can always do so from the most recent hourly build. Our release turn-around time, and indeed the risk to our customers, is thus greatly reduced.

Writing the first set of tests for an existing class often caused us to improve the design of the class itself. The main reason for this is that the tests often need to access smaller pieces of the class' functionality, so existing methods must often be split into "public" and "private" parts (see Fowler's Extract Method refactoring pattern [Fowler99]). A side-effect of this is that those classes and packages that have been tested heavily are now generally composed of smaller pieces, displaying more flexibility and embedding fewer assumptions about their clients. These parts of our system therefore now tend to be easier to evolve.

4. Negative Points

This section describes the down-side of the approach we took to retrofitting unit tests in the Stingray development.

As with any testing approach, any part of the system that has a low test coverage can lead to a "false positive". If I run the entire suite of unit tests at the end of a development episode, the absence of test failures does not imply the absence of bugs – I may have broken a part of the system that isn't well tested. The run of 300-400 dots in the test report, with no Fs or Es to indicate problems, encourages me to stop sooner and integrate what I have. Indeed this is one of our Plus Points above, because usually this does allow development to proceed more quickly. But occasionally, that untested area of the system is broken, and it may be a long time before we find out.

The existence of a `Tester` class does not imply that a class is fully tested. In fact, before the general adoption of our test-case method naming conventions above, a large and complex `PackageTest` class may easily have appeared thorough, while actually testing only a fraction of the necessary conditions. This could also lead to false positives ("that's a big test case, it surely checks everything"), and sometimes actually reduced our willingness to add new tests.

Our recursive make system makes it easy to run the tests for the package one is altering, but somewhat harder (slower) to run the tests for the entire system. Although this is quick and convenient during development, we often succumb to the temptation to test only one package and then integrate, thinking that our changes will probably have no side-effects elsewhere. This is frequently an invalid assumption, with the result that the system temporarily degrades while the updates are fixed.

Every so often an email arrives from the hourly batch test run, indicating that something in the system is now failing on the target platform. Every member of the team now takes a couple of minutes out to look at the log and check their recent changes. Thus one person's error disturbs everyone else in the team.

Adding new tests to an old class can be a slow and painful business. This is particularly true when the class to be tested is badly organised, or when it doesn't define a clear abstraction, so that its responsibilities leak out into its clients and suppliers. Such classes deter the would-be tester. It is usually far simpler to just add the required functionality to the class and leave it untested, than it is to bite the bullet and develop the necessary tests, knowing that the class and some of its collaborators must be refactored in the process. This is probably the main reason we have `Tester` classes for only 1 in 4 of our production classes.

Old habits die hard. Although all our old unit tests were "ported" to JUnit, many of them still used *ad hoc* reporting mechanisms, or required user interaction when they failed. Tests like this are a natural consequence of the process of gradually retrofitting new technology, and have now all but disappeared. But during the transitionary period, their presence helped to keep parts of the system a little more mysterious than necessary.

Unit tests for the GUI, and for the shell scripts that launch our system, are hard to write – so we don't have any. (Although there exist counterparts of JUnit for most popular programming languages, Bourne shell is not among them.) We have had bugs in both of these areas recently, which went undetected for some time due to the absence of unit tests.

One or two packages of tests came to rely heavily on large and complex test fixtures. For ease of development, these had often been simply extracted lock, stock and barrel from System Tests of the previous version of the product. These test fixtures were therefore often monolithic, and contained much more data or structure than was required for a unit test. When the time came to alter the test fixture for a new unit test, the fixture's enormous weight and complexity often made this a daunting task. The new test would not be written, usually meaning that new functionality was added to the system with no accompanying unit tests. Ultimately, one team member took a week out of normal development to slim down these fixtures and refactor the tests to be more lightweight.

5. Future Plans

This section describes some of the activities we plan to undertake in order to address the negative points listed above.

We need to develop tests for the more difficult parts of the system, especially the GUI. GUIs are notoriously hard to unit test, because of the difficulty of simulating external events. However, users of Java Swing are now starting to evolve JUnit-based testing techniques that largely remove this difficulty [Wake00]. We plan to adopt this approach to test the Corvette GUIs soon.

We need to reduce the frequency at which the unit tests for a single package are taken to be an adequate measure of the quality of the whole system. One way to do this would be to remove our ability to run the tests in this way, and to provide only a single test suite covering all of the production code; such a suite could even be assembled on the fly. We have a simple script that can do this, and it currently takes less than a minute to run, but some further work is required before running the Unit Tests for the entire system is quick and simple.

Although our hourly batch test runs can disturb the entire team, we can't do without them because our product must port to numerous platforms. However, we do need to find a way to include a fast test run into the integration process, so that new code is rejected if it fails on any of our target platforms. We are actively looking for a way to perform a complete build and test in under 10 minutes, and to automatically include that in our source code integration process. (The testing here would include both the Unit Tests and our growing suite of System Tests, which have been developed separately.)

Some team members are beginning to use unit tests in the debugging process. When a reported bug has been analyzed and the cause determined, it is almost always possible to write a new unit test that currently fails, but which would pass if the bug were fixed. Adding this test serves to guide the developer fixing the bug, and will prevent us from accidentally re-introducing the bug in the future. As the system evolves from alpha development to supported product, this should become a reflex part of our process.

As the system grows, the tests must continue to grow with it. We are past the "early adopter" stage with our unit tests now, and yet a number of classes in the system remain stubbornly untestable. One radical approach to both of these problems is that proposed in XP: whenever new functionality is to be added anywhere, the unit tests are developed *first*. Designing the tests equates to designing the interface and semantics of the new feature [Beck98, Wake00]. And when the tests all pass, development is done. Such an approach promises to improve development quality and speed, while simultaneously offering the benefits we found above from more testable (and hence adaptable) production classes. This seems to be an avenue worth pursuing.

Flushed with our success using JUnit, we intend to offer encapsulation developers a simple and rich test framework, to encourage them to write unit tests. The framework would consist of JUnit, together with a package of test utility classes, probably based on those we have developed for testing the main system.

6. Summary and Conclusions

We took an ongoing, medium-sized Java development project and gradually added a unit testing regime. The tests were developed over time, with little impact on development timescales, using an open-source Java framework called JUnit. Current test coverage is sufficient to have greatly improved our confidence in the software's quality.

The pain of moulding ourselves into new habits and new conventions seems to have paid off handsomely. Development speed is now improved in certain areas, as is the adaptability of much of the software. The JUnit approach to unit testing also seems to have speeded up some of our processes, including time to release.

We hope that the lessons we learned along the way will help others get to this point a little more quickly and painlessly.

7. References

[Beck99] Beck, *eXtreme Programming eXplained*, Addison-Wesley, 1999.

[Beck98] Beck, Gamma, *Test Infected*, Java Report, July 1998.

[Beck+99] Beck, Gamma, *JUnit – A Cook's Tour*, Java Report, May 1999.

[Fowler99] Fowler, *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gamma+94] Gamma, Helm, Johnson, Vlissides, *Design Patterns – Elements of Reusable Software*, Addison-Wesley 1994.

[Wake00] Wake, *Using JUnit to Unit-Test GUIs*, http://users.vnet.net/wwake/xp/xp0001/index.shtml