The Test Automation Manifesto

Gerard Meszaros
ClearStream Consulting
3710 205 5th Ave. SW
Calgary, AB
T2P 2V7 Canada
1-403-560-2408
gerard@clrstream.com

Shaun Smith
Sandbox Systems
Calgary, AB
Canada
shaun.smith@acm.org

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

ABSTRACT
Two key aspects of eXtreme Programming are automated testing and frequent refactoring. But is refactoring the best way to arrive at a set of tests that are both sufficient and maintainable? This paper builds on previously cataloged test smells, classifies these smells into two broad categories and introduces principles (or goals) for test automation. It also provides the start of a generative pattern language that helps guide the construction of automated tests that should not require extensive refactoring.

Keywords
Automated Testing, Maintenance, JUnit, XUnit, Patterns, Best Practices, Refactoring

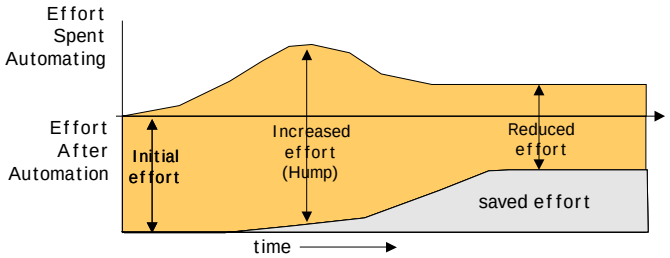
1 INTRODUCTION

Much has been written about the need for automated unit and acceptance tests as part of agile software development. But writing good test code is hard and maintaining obtuse test code is even harder. Since test code is optional (not shipped to customers), there is a strong temptation to give up testing when the tests becomes difficult or expensive to maintain. Once you have given up on the principle of "keep the bar green to keep the code clean", much of the value of the automated tests is lost.

Over a series of projects we have faced a number of challenges to automated testing. The cost of writing and maintaining test suites has been a particular challenge, especially on projects with hundreds of tests. Fortunately, necessity is the mother of invention and we, and others, have developed a number of solutions to address these challenges. We have also gone on to introspect about these solutions to ask ourselves why they are good solutions and what is the underlying test automation principle that they uphold? We called these collected principles Test Automation Manifesto. We believe that adherence to the principles of the Manifesto will result in automated tests that are easier to write, read, and maintain.

History

On our first test-first project, we encountered a number of problems: the cost of updating existing tests was beginning to become a major component of the overall cost to implement a new feature, the cost of writing automated tests for new features was increasing, and the effort required to run the test suite was growing. Changes to the software under test's (SUT) API would impact dozens of tests. For example, adding a parameter to a class constructor would mean revisiting every test that created an instance of that class. We found that as tests were developed for more complex requirements, the effort to setup and teardown test fixtures was becoming greater than the effort to exercise and verify the new behavior. And we found that we could no longer just press the "run" button to run the test suite; we would have to truncate all the tables in the database before we could run a test suite because a previous run had not cleaned up after itself. Test automation, which had seemed so simple at the beginning, was becoming a burden. We still enjoyed the benefits of



automated testing, but the investment cost was increasing. We had to find ways to reduce the cost while producing the valuable return we wanted.

Economics of Test Automation

Of course there is will always be a cost to building and maintaining an automated test suite. Ardent test automation advocates will argue that it is worth spending more to have the ability to change the software later. This "pay me now so you don't have to pay me later argument" doesn't go very far in a tough economic climate. And the argument that the quality improvement is worth the extra cost doesn't go very far in these days of "just good enough" software quality.

The goal should be to make the decision to do test

automation a "no-brainer" by ensuring that it does not increase the cost of software development. This means that the additional cost of building and maintaining automated tests *must* be offset by savings through reduced manual unit testing and debugging/troubleshooting as well as the remediation cost of the defects that would have gone undetected.

2 BAD SMELLS IN TEST CODE

At XP2001, van Deursen et al [6] introduced a number of "bad smells" that occur specifically in test code. They recommend a set of refactorings that can be applied to the tests to remove them. Many of our initial problems with test automation involved those smells as well a number of others that we identified and developed solutions for. We have also discovered that there are at least two different kinds of smells: "*code smells*" that must be recognized when looking at code, and "*behavior smells*" that manifest themselves when you least expect. The latter are much harder to ignore because tests are usually failing as you try to integrate your code and you must unearth the problems before you can "make the bar green".

Bad Smells—Code

Code smells are the "classic" bad smells as first described by Fowler in [3]. These smells must be recognized by the test automater as they maintain test code. Most of the smells introduced by Fowler are code smells. Code smells typically affect maintenance cost of tests but they may also be early warnings signs of behavior smells to follow.

Hard Coded Test Data—Lots of "Magic Numbers" or Strings used when creating objects. More likely to result in an *Unrepeatable Test*.

Test Code Duplication [6]—Same code sequences appear many times in many tests. More code to modify when something changes (causes *Fragile Tests*)

Mystery Guest [6]—When a test uses external resources such as a file containing test data, it becomes hard to tell what the test is really verifying. These tests often have a "lopsided" feel to them (either setup or verification of outcome is external to test).

Complex Test Code—Too much test code or *Conditional Test Logic*. Hard to verify correctness; more likely to have bugs in the tests

Can't See the Forest for the Trees—So much test code that it obscures what the test is verifying. The tests do not act as a specification because they take too long to understand.

Conditional Test Logic—Tests containing conditional logic (IF statements or loops). How do you verify that the conditional logic is correct? Does it always test the same thing? Do you have "untested" test code?

Complex Undo Logic—Complex fixture teardown code. More likely to leave test environment corrupted by not cleaning up correctly. Results in "data leaks" that may later cause this or other tests to fail for no apparent reason.

Bad Smells—Behavior

Behavior smells are smells you encounter while running tests.

Fragile Tests—Every time you change the SUT, tests won't compile or they fail. You need to modify lots of tests to get things "green" again. This greatly increases the cost of maintaining the system. Contributing code smells include *Test Code Duplication* and *Hard Coded Test Data*.

Fragile Fixture—Tests start failing when a shared fixture is modified (e.g., new records are put into the database). This is because the tests are making assumptions about the contents of the shared fixture. A contributing code smell is *Mystery Guest*.

Interdependent Tests—When one test fails, a number of other tests fail for no apparent reason because they depend on a previously run tests' side effects. Tests cannot be run alone and are hard to maintain.

Unrepeatable Tests—Tests can't be run repeatedly without manual intervention. Caused by tests not cleaning up after themselves and preventing themselves (or other tests) from running again. The root cause is typically *Hard-coded Test Data.*

Test Run War [6]—Seemingly random, transient test failures. Only occurs when several people testing simultaneously. Caused by parallel tests interacting with each other through a shared test fixture.

Beyond the Refactoring of Smells

In [6], the authors provided suggested refactorings for each of the bad smells. When we refactor production code, we rely on our automated tests to discover any problems introduced by the refactorings. But when we refactor our tests, what will alert us to broken tests? If a test fails when it used to pass, we can be certain that we have broken the test, but is "no news, good news"? Unfortunately not! The only way to verify that the tests haven't been broken by the refactorings is to modify the production code to introduce each of the bugs that the tests are designed to detect. Tools such as Jester [4] may help in this process but success is not guaranteed.

We believe there is an alternative to all this test refactoring. Many of the smells can be detected very early in test automation or avoided entirely. Rather than asking what refactoring you should apply to remove a smell, we prefer to ask what principle that is being violated when the smell is present.

Note that we are not advocating "big up-front design" of the tests. As consultants, we have seen many examples of testing frameworks built in anticipation of testing needs—needs that may or may not be real. These frameworks

usually end up causing more problems than they solve. What we *are* advocating is thoughtful application of test automation patterns that we have found help us *avoid* the smells. The patterns all support a small set of test automation principles that are being violated when the various smells are present. We propose these principles and patterns as a "Test Automation Manifesto".

3 TEST AUTOMATION MANIFESTO

Based on many years of experience building and maintaining automated unit and acceptance tests, we propose the following "Test Automation Manifesto".

Automated tests should be:

Concise—As simple as possible and no simpler.

Self Checking—Test reports its own results; needs no human interpretation.

Repeatable—Test can be run many times in a row without human intervention.

Robust—Test produces same result now and forever. Tests are not affected by changes in the external environment.

Sufficient—Tests verify all the requirements of the software being tested.

Necessary—Everything in each test contributes to the specification of desired behavior.

Clear—Every statement is easy to understand

Efficient—Tests run in a reasonable amount of time.

Specific—Each test failure points to a specific piece of broken functionality; unit test failures provide "defect triangulation"

Independent—Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.

Maintainable – Tests should be easy to understand and modify and extend.

Traceable—To and from the code it tests and to and from the requirements.

4 TEST AUTOMATION PATTERNS

Refactoring to eliminate smells is a good way to remove a problem once it has been created. "Generative" test automation patterns can be used to guide test automaters in avoiding the problems in the first place. In our experience, the following patterns can help ensure that automated tests comply with the Test Automation Manifesto.

Readability Patterns

Single Glance Readable
A test should visibly tie the expected outcome to the conditions that should cause it. A quick read of a test should be enough to understand what it tests. The test should fit in a single pane of the window without scrolling.

Intent Revealing Fixture
The part of the test that describes the fixture, the pre-conditions of test, should focus on what's relevant to this specific test. Anything irrelevant is hidden (encapsulated). This avoids the inclusion of objects and values that have no direct bearing on the condition being tested. Well-named *Finder Methods* and *Anonymous Creation Methods* are a common ways to do this.

Finder Methods
When reusing objects in a shared fixture, rather than using hard-coded object keys in your test, use clearly-named Finder Methods. This makes it easy to understand why the test is using specific objects and avoids the *Mystery Guest* smell.

Outcome Describing Verification Logic
The verification part of the test should make it very clear what the expect outcome should be. No "reading between the lines" should be required.

Single Condition Test
Tests should verify a single test condition (a single scenario). This makes them much easier to understand and maintain. They also make it easier to organize the tests in a way that makes it obvious which conditions are covered (and which ones remain to be tested.)

Declarative Style
All parts of the test should describe what is (fixture) or should be (expected results), rather than provide a recipe for how to create/verify it. Use of an *Expected Object* is one way to do this.

Robustness Patterns

Independent Tests
Each test is self-contained and makes no assumptions about what other tests have run before it or will run after it.

Clean Slate Fixture
Tests set up everything they depend on. Avoids depending on other tests, either on purpose or accidentally. Ensures the state of all objects is well understood.

Anonymous Creation Methods
Tests use common utility methods to create unique objects for each test and test run. Only the attributes of interest to the test are passed as "constructor" arguments. This ensures tests are repeatable and robust. It also prevents *Test Run Wars* since each instance of this test will create it's own, unique objects so it cannot "collide" with itself. These methods reduce the cost of writing tests by providing reusable building blocks.

Automated Test Cleanup
Eliminates complex (and untestable) "undo logic" in tests. Avoids test environment corruption ("data leaks"). Reduces the cost of writing tests by eliminating the most error-prone work.

SUT API Encapsulation
Reduces maintenance cost by isolating tests from unimportant changes to SUT API. Helps make test more readable by focusing on what *is* important.

Reuse Patterns

Reuse thru Test Building Blocks
Call building blocks rather than inheriting and overriding. Facilitates Single Glance Readable tests.

Anonymous Creation Method
Reusable (and testable) fixture setup logic (see Robustness Patterns).

Custom Assertions
Reusable object comparison logic that implements "test-specific equality". These are refactored using *Extract Method* when the same set of assertions appears in two or more tests. It simplifies the tests greatly yet avoids polluting production code with non-production object comparisons (which may need to vary from test to test anyway.) Non-trivial custom assertions (e.g. comparing XML) can and should be tested with unit tests of their own.

Parameterized Test
To apply the same test logic in a number of circumstances, write a test that takes a parameter that is used to determine which pair of inputs/expected-outputs to use. Either write a set of individual tests that just delegate to the Parameterized Test, or use a single Data-driven Test Suite that contains the values to be tested.

Templated Framework Tests
When testing framework plug-ins where every plug-in needs to be tested the same basic way, create a *Parameterized Test* that implements Template Method [GOF] which calls plug-in specific bits to setup the fixture and verify the outcome. Use a *Parameterized* Test to tell the Framework Test which plug-in to test.

Data-Driven Test Suite
When you have a large number of tests that require the same logic but different data, consider creating a data-driven test suite that reads the data and calls the appropriate *Parameterized Tests*. This allows tests to be created without "programming". The FIT framework [2] is a good example of this style of testing.

Other Patterns

Round-Trip Test
Avoid over-specification (and Fragile Tests) by testing inputs and outputs at same "black box" interface.

Stub Out Slow
Replace any slow component that is depended upon with a test stub. For example, stub out a database to speed up tests by orders of magnitude [5].

Stub Out Dependencies Beyond Control
Anything beyond your direct control should be stubbed out so it doesn't cause unexpected results or delays.

5 CONCLUSION

Over a series of projects we have learned not only to ruthlessly refactor our production code to keep it clean, but we have also learned to do the same with test code. But the principles of test code refactoring are not the same as those for production code refactoring.

The Test Automation Manifesto defines the principles that underlie highly effective tests. All test code refactoring activities should improve the alignment with these principles. Does a refactoring improve robustness? Does it make it more concise or clear? If not, it is probably the wrong refactoring.

When first writing a test, the Manifesto acts as a checklist of the qualities that lead to tests that are less likely to need refactoring. We have found that applying the generative test automation patterns leads us to produce clear, maintainable, robust automated tests that are much less likely to require refactoring to add these qualities after the fact.

6 REFERENCES

1. Appleton, Brad. Generative Patterns http://www.enteract.com/~bradapp/docs/patterns-intro.html#GenerativePatterns
2. Cunningham, Ward. FIT: Functional Integrated Test. http://fit.c2.com.
3. Fowler, Martin. Refactoring: Improving the design of existing code.
4. Moore, Ivan. Jester
5. Smith, Shaun; Meszaros, Gerard. Increasing the Effectiveness of Automated Testing. *The Second International Conference on eXtreme Programming and Agile Processes in Software Engineering, XP2001*. May 2001.
6. van Deursen, Arie (et al). Refactoring Test Code. *The Second International Conference on eXtreme Programming and Agile Processes in Software Engineering, XP2001*. May 2001.