Technical lessons learned turning the agile dials to eleven!

Paul King
ASERT
paulk@asert.com.au

Craig Smith
Suncorp
craig.smith@suncorp.com.au

Abstract

This report outlines technical lessons learnt by about 20 of Australia's most experienced agile specialists over several years across several projects within an organization which aggressively applied the agile practices with much success. In these projects the agile dials were cranked to eleven to achieve very high levels of quality. Most of the specialists involved believe that they produced the highest quality software of their careers with some of the highest productivity they have ever experienced.

1. Introduction

Agile methodologies such as Extreme Programming [1,5,22,23] help developers productively deliver high-quality features to their customers on an on-going basis. Most agile practitioners would agree that using more of the agile practices on their projects is likely to raise the quality of the developed software but there would be diverse opinion about which of the practices are the most important and also on the return on investment for some of the practices.

This paper discusses the experiences of a team that tried to commit fully to all of the practices but also tried to maintain or increase the short-term productivity expected by similar teams within the organization. The team brought together some of Australia's most experienced agile specialists who have now worked closely across numerous projects. This paper draws upon experiences from numerous projects; however most of the lessons learnt originated from one major project called EasyDoc. That project was a document generation and delivery system primarily coded using Java but also used numerous dynamic languages and various open source libraries. The system was required to integrate with several vendor systems; it incorporated web applications for administration purposes, and it integrated with numerous calling applications and third parties using web services.

2. Setting the Dials to Ten

The initial goals of the team included:

- 100% code coverage from unit tests
- All production code paired and test-driven
- Minimal design up front but an appreciation for when such design made sense
- Customer focused outcomes
- Full continuous integration
- Daily pair rotation
- Continuous improvement through retrospectives
- High levels of automation (including an installer that loaded and configured the approximately 20 tools developers used onto a freshly installed operating system)

We also set in place some light-weight metrics monitoring various project metrics. While it is difficult to accurately compare the metrics from this project with other projects Figure 1 certainly illustrates that velocity of the team remained constant over a long period of time even though code size and complexity continually increased over time.

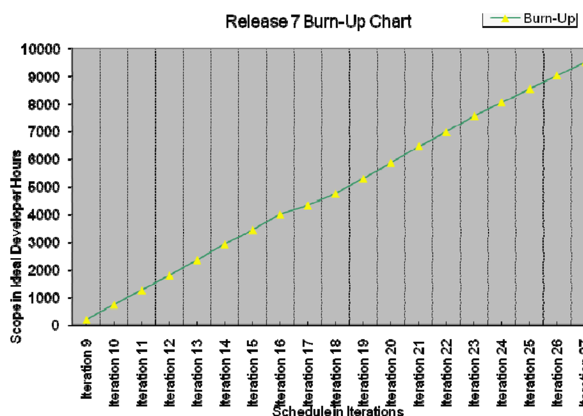

Figure 1 - Burnup chart

With these goals in place, we then set out to crank the agile dials even further.

3. Quality Metrics

Duplication

An on-going problem in some earlier projects in the organization was the amount of source code duplication. When a bug was found, it often meant changing code in several places with great confusion when not all places were fixed. To combat this, we used the Simian duplication detection tool [18] with a threshold of 4 lines. If any 4 consecutive lines in any production or test source file were duplicated, the build would break and we would refactor out the duplication.

The team was divided on whether this threshold was set too high. It certainly forced the team to be highly disciplined and after some teething troubles didn't impact our code or productivity to any significant extent. In hindsight, I suspect we could slightly raise this threshold without introducing too much unintended duplication.

One area which we have explored doing differently is applying this threshold to our acceptance tests. The developers on the team now naturally refactor any duplication in the acceptance tests into helper or library methods. However, we have found that sometimes intended duplication is preferable when explaining acceptance tests to the customer or business analysts. They don't always naturally think about the system with such refactoring in place.

Method and Class Complexity

Cyclomatic complexity was set to much lower than normal levels effectively prohibiting for instance nested looping statements or nested conditions or nested try catch blocks. This was mostly a worthwhile exercise but we did need to introduce a mechanism for excluding this check in a very small number of cases where a class inherently served a very special purpose that could not be coded in any other way using Java.

Method and Class Size

Common agile thinking is that large methods and large classes are hard to understand and refactor. Why not then strive for very small methods and very small classes? How small? Well, 7 lines will allow you to have a `try … catch … finally` block plus one other line. We tried that and 8 lines and eventually raised the value just a little further.

In the end, methods were limited to approximately 10 lines and classes to about 80 lines (we used slightly different metrics for test and production code). This (like the complexity settings) forced us to refactor any significantly complex class and because of our TDD approach and our 100% code coverage target, meant we had to create the accompanying unit tests. We made several observations resulting from having such stringent metrics in place:

- Due to code simplicity, it was always easy to understand what any individual class or method did.
- Our code base sometimes tended towards Ravioli style code [2]. This meant that while any class or method was easy to understand, the number of classes had increased dramatically and it was now sometimes hard to understand what all the classes did and where certain functionality was situated within the source files. Having great IDE support made this problem manageable.

A lesson we learnt was that it is best to try to minimize across all the various dimensions of size and complexity rather than just trying to minimize on any one individual axis. As a gross simplification of this concept, if you have 50 lines of business logic within a class, you are better off with about 7 methods each containing about 7 lines rather than two 25-line methods or twenty five 2-line methods.

We used our own metrics plugin [17] which enabled us to have exactly the same metrics rules in place and 'live' within the IDE as we did within our CI build. We will have more to say about these metrics in section 8.

4. Dealing with Boundaries

One of the more talked about features of the Java language is its support for checked exceptions. This feature allows class designers to force users of their class to handle any abnormal conditions which arise during execution of a class' methods. While this is debatably a very powerful feature when used correctly [6: Praxis 16-27] [3: Items 39-47], it does create additional work for agile teams if low-level libraries make extensive use of this mechanism.

Firstly, the production code must contain boiler-plate exception handling logic. This logic takes time to write and can obscure the main intent of the non error handling business logic, making it harder to read. Secondly, for agile teams striving for 100% code coverage, more work is required to test the added logic even though the approaches to doing so are well understood [4: Section 2.8] [8: p.89] [9: pp.25-31].

Fortunately, a fairly straight-forward approach to dealing with checked exceptions is frequently used. The delegation pattern [11] is used to encapsulate each checked exception in a runtime exception [12: Section 4.2.4]. This pattern is applied to each method of each class in the offending library. The resulting class files are frequently called boundary classes or edge classes [19] and should contain no logic but the exception wrapping logic. These classes are excluded from the

code coverage analysis but should be visually inspected.

As an example of this technique, consider using Java's `File` class and suppose we only were interested in using the `getCanonicalPath` method. We might create a boundary class as follows:

```
public class FileBoundary {
  private File delegate;

  public String getCanonicalPath() {
    try {
      return delegate
          .getCanonicalPath();
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

To crank up the dials, we tried a number of approaches:

- At first we used an IDE plugin to auto create boundary classes given the original file. This applied the pattern to all public methods. We could remove some of the methods if we didn't need them.
- We also explored using the Groovy language [21,24] as it integrates very smoothly with Java and automatically converts checked exceptions to runtime exceptions. At the time though, the IDE support for Groovy wasn't as good as it is now.
- We settled on using autoboundaries. With this approach, we simply created an interface containing the methods of interest and used a naming convention to associate it with the class we wished to wrap. We then wrote a custom instance provider for Spring [15] (the IOC container in use at the time) which automatically created a dynamic proxy class matching the original `File` class with all of the appropriate exception handling logic.

For the `File` example discussed above, our autoboundary interface would look similar to:

```
package autoboundary.java.io;

public interface File {
    String getCanonicalPath();
}
```

We found this approach to be very effective. We had no need to worry about impacts on coverage and we were saved from the need to explicitly create any boundary classes. The interface file itself also documented very clearly which parts of the external library were being used.

5. Easing the mocking burden

Another area which impacted team productivity given our goal of 100% code coverage was writing mock-based tests. Traditional approaches to writing such tests are well known [4,7,26,22]. As an example, consider the following test written using a traditional JMock2 style:

```
public class DocumentJMockTest {
  Integer count = 4;
  String line = "a dummy line";
  Mockery context = new Mockery();
  DocumentPropertiesImpl docProps;
  DocumentReader reader;
  WordCounter counter;

  @Before
  public void before() {
    reader =
      context.mock(DocumentReader.class);
    counter =
      context.mock(WordCounter.class);
    docProps = new DocumentPropertiesImpl(
                  reader, counter);
  }

  @Test
  public void countsWordsInOneLine() {
    context.checking(new Expectations() {{
      one(reader).hasMoreLines();
        will(returnValue(true));
      one(reader).readLine();
        will(returnValue(line));
      one(counter).count(line);
        will(returnValue(count));
      one(reader).hasMoreLines();
        will(returnValue(false));
      ignoring(reader).close();
    }});
    assertEquals(count,
      docProps.countWords());
    context.assertIsSatisfied();
  }
  …
```

The code here isn't too complex but there is a little bit of work to do setting up our mocks and test constants. For larger tests, this can be a more significant burden and it also reduced our ability to heavily automate creation of the test code. We will see later that by almost totally removing creation of dummy test constants and auxiliary mocks, we can almost fully automate mock creation for many of our classes.

We also looked at using Groovy tests but tool support for Groovy at the time was not as good as is currently available, so that approach was ruled out. The interesting part of those tests (the DSL for specifying the behavior of the mock) looked like this:

```
// Groovy
class DocumentGroovyTest {
  def count = 3

  @Test void countsWordsInLine() {
```

```
// set up counter mock … not shown …
// set up reader mock …
mock.demand.with {
  hasMoreLines { true }
  readLine { "a dummy line" }
  hasMoreLines { false }
  close {}
}
reader = mock.proxyDelegateInstance()

def docProps = new
  DocumentPropertiesImpl(reader,
                         counter)
assert docProps.countWords() == count
…
```

This allowed us to create more succinct tests with a much more DSL flavor but still involved more boiler plate code than we desired. We eventually moved to an approach where the testing infrastructure provided additional "magic" values to our test classes using various conventions. We mainly used naming conventions similar to those shown here:

```
String uniqueSurname;
Letter dummyLetter;
Mailer wiredMailer;
Summer mockSummer;
Sender stubSender;
```

When our test framework ran our tests, any field of our test class which followed these naming conventions and had a null value was automatically filled in. The prefix *unique* allowed us to use random values, *dummy* allowed us to provide a given known value with a well-defined instance creation framework including nested dependant (child) objects, *wired* would allow a bean to be provided as specified by an IOC wiring strategy and *mock* and *stub* created mocks and stubs respectively. We'll have more to say about dummy and random values in Section 6.

Before now looking at where this at first strange approach leads us, we should point out that some of these ideas are now in the Boost [19] and Instinct [14] open source projects which can use naming conventions or annotations to distinguish cases, e.g. for the case above, an Instinct version looks like:

```
@RunWith(InstinctRunner.class)
public class DocumentWithOneLineRemaining {
  @Subject(auto=false) private
    DocumentPropertiesImpl docProps;
  @Mock private DocumentReader reader;
  @Mock private WordCounter counter;
  @Stub private String fileName;
  @Stub private Integer count;
  @Stub private String line;

  @BeforeSpecification
  public void before() {
    docProps = new DocumentPropertiesImpl(
      reader, counter);
  }

  @Specification
  public void countsWordsInLine() {
    expect.that(new Expectations() {{
      one(reader).hasMoreLines();
        will(returnValue(true));
      one(reader).readLine();
        will(returnValue(line));
      one(counter).count(line);
        will(returnValue(count));
      one(reader).hasMoreLines();
        will(returnValue(false));
      ignoring(reader).close();
    }});
    expect.that(docProps.countWords())
      .isEqualTo(count);
  }
```

Now that we have seen that we can remove much of the clutter and boilerplate code in our test cases, we should consider additional implications. Let's consider test driving a `Book` class. A `Book` may have an `Author` and an `Author` may have a `String` property called `name`. If we would like a method `getAuthorName` on `Book` then we might create a test as shown in Figure 2:

```
public class BookTest extends PrimordialTestCase {
    private Book book;
    private String name;
    private Author author;

    protected void setUpFixtures() {
        book = new Book();
    }

    public void testGetAuthorName() {
        name = author.getName();
            Convert to an expectation                  getAuthorName());
            Move assignment to field declaration
}
```

Figure 2 - IDE support to create expectations

Using special IDE plugins [17], instead of writing traditional mock expectations, we write what we think the production code is going to be. Then we use the '*Convert to an expectation*' intention to convert that to a test, similar to that shown below in Figure 3:

```
public class BookTest extends PrimordialTestCase {
    private Book book;
    private String name;
    private Author mockAuthor;

    protected void setUpFixtures() {
        book = new Book();
    }

    public void testGetAuthorName() {
        expectOneCallTo("getName", mockAuthor)
                .will(returnValue(name));
        assertEquals(name, book.getAuthorName());
    }

}
```

Figure 3 - Resulting Test

Now to make this test green, we need to write the exact same line in our production code. We are effectively duplicating our production code in two places. The resulting production code will look like:

```
public class Book {
    Author author;
    String getAuthorName() {
        return author.getName();
    }
}
```

Figure 4 - Production Code

The implication of this approach is that for simple scenarios (like this example) we can write just one of either the test or the production code and generate the other. We'll have more to discuss about the implications of this approach in Section 8.

6. Instance providers

One of the features we spoke about in Section 5 was the ability to use dummy and random values. We elaborate on that idea here. Consider the following state-based unit test:

```
public class SummerTest extends TestCase {
  public void testSumStrings() {
    final Summer subject = new Summer();
    assertEquals("ab",
                 subject.sum("a", "b"));
    assertEquals("cd",
                 subject.sum("c", "d"));
  }

  public void testSumNumbers() {
    final Summer subject = new Summer();
    assertEquals(23, subject.sum(20, 3));
    assertEquals(32, subject.sum(30, 2));
  }
}
```

The test contains multiple String values used to triangulate the `sum` methods. We also need to decide whether such constants really belong as directly hard-coded values, class-level constants or shared constants across multiple tests? We can avoid these tricky questions by not introducing such arbitrary values into our test and instead just use random values. (Note: some testing frameworks allow a random seed to be provided to allow a repeatable sequence of randomly generated values to be used.) In such cases, triangulation occurs by running the test more than once (or by different pairs). Hence, we can write the test as follows:

```
public class SummerTest
    extends BaseTestCase {
  String uniqueStringA;
  String uniqueStringB;
  int uniqueIntA;
  int uniqueIntB;
  Summer subject;

  protected void setUpFixtures() {
    subject = new Summer();
  }

  public void testSumStrings() {
    assertEquals(
      uniqueStringA + uniqueStringB,
      subject.sum(uniqueStringA,
                  uniqueStringB));
  }

  public void testSumInts() {
    assertEquals(
      uniqueIntA + uniqueIntB,
      subject.sum(uniqueIntA, uniqueIntB));
  }
}
```

Immediately, we can see that clutter has been reduced in this test and hence more time has been devoted to writing valuable production code.

We should point out at this time that triangulating over multiple test runs will still yield 100% coverage but only if we don't have any branching or conditional logic in our class under test. In such cases, we still need to ensure that all paths are followed whether we make use of randomness or not.

7. Autochecking

Another practice we adopted was to increase the levels of automatic testing and checking of various properties. At the time, other teams were debating the merits of testing or test-driving simple getters and setters or debating the value of testing for nulls. Given our goal of 100% coverage, we felt we had no choice but to test all of these things, so we altered our testing framework to make such practices almost no work for developers.

By virtue of the base production classes and base test case classes that we used, as well as some simple conventions, we obtained a lot of checking almost for free. As an example, for every class, we automatically checked that all production code guarded against null values for each constructor parameter and all public method parameters. We distinguished between various categories of classes, e.g. *Data* (non-persistent POJO), *Domain* (persistent POJO), *Components* (remotable) and some variations such as *Immutable*, *Serializable*, etc. For each category we performed appropriate additional tests. As an example, we used Hibernate [14] for our persistent domain objects, so for each Domain class we checked that it was not final and that it had a protected `id` field – both Hibernate requirements. This concept is now also appearing in

other testing frameworks, e.g. JDave [16] has a similar concept called contract checks:

- *EqualsHashCodeContract*
- *SerializableContract*
- *CloneableContract*
- *EqualsComparableContract*

8. Atoms, Molecules and Disposable Tests

We wrap up our experiences by describing where our thoughts are heading. We are reasonably happy with the level of productivity that we are getting using our current quality metrics and testing approach but we often create unit tests which are in some sense disposable as discussed in Section 5. Currently we find it beneficial to think of our fine grained classes as atoms and our more coarsely grained classes as molecules. By analyzing atomic classes which do not store state and simply delegate through to other classes, e.g. using Complexion [20], we can dispose of any tests associated with those classes. If we are correctly doing TDD, we should still achieve 100% code coverage at the molecular level.

9. Acknowledgments

The authors wish to thank Suncorp management and members of the EasyDoc and EasySuite teams and other Suncorp staff who assisted the team or were involved in agile and other customer-focused initiatives within Suncorp. Thanks also to Ben Sullivan and Jeremy Wales for commenting on a draft of this paper.

Suncorp is one of Australia and New Zealand's largest diversified financial services providers, supplying banking, insurance and wealth management products to around 7 million customers through well-established and recognized brands such as AAMI, Australian Pensioners Insurance Agency, Shannons, Vero, Asteron and Tyndall, as well as Suncorp and GIO. Today, Suncorp is Australia's sixth largest bank and second largest domestic general insurance group, with over 16,000 staff. Suncorp has representation in 450 offices, branches and agencies throughout Australia and New Zealand.

10. References

[1] K. Beck and C. Andres, *Extreme Programming Explained*, 2nd Ed, Addison-Wesley, 2005

[2] Ravioli Code article on the C2 (Ward's) Wiki: http://c2.com/cgi/wiki?RavioliCode

[3] J. Bloch, *Effective Java*, Addison-Wesley, 2001

[4] J.B. Rainsberger, *JUnit Recipes*, Manning, 2005

[5] V. Subramaniam and A. Hunt, *Practices of an Agile Developer*, Pragmatic Bookshelf, 2006

[6] P. Haggar, *Practical Java*, Addison-Wesley, 2000

[7] A. Hunt and D. Thomas, *Pragmatic Unit Testing in Java with JUnit*, Pragmatic Bookshelf, 2003

[8] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005

[9] C. Beust and H. Suleiman, *Next Generation Java Testing*, Addison-Wesley, 2008

[10] G. Meszaros, *xUnit Test Patterns*, Addison-Wesley, 2007

[11] Wikipedia: *The Delegation Pattern*, http://en.wikipedia.org/wiki/Delegation_pattern

[12] B. Tate, M. Clark, B. Lee and P. Linskey, *Bitter EJB*, Manning, 2003

[13] *The Instinct Project*, http://code.google.com/p/instinct/

[14] *The Hibernate Project*, http://hibernate.org/

[15] *The Spring Framework*, http://springframework.org/

[16] *The JDave Project*, http://www.jdave.org/

[17] *The Agile Plugins Project*, http://code.google.com/p/agileplugins/

[18] *The Simian Code Duplication Detection tool*, http://www.redhillconsulting.com.au/products/simian

[19] *The Boost Project*, http://geekscape.org/daisy/geekscape/g2/128.html

[20] *The Complexion Project*, http://www.assembla.com/wiki/show/complexion

[21] *The Groovy Language*, http://groovy.codehaus.org/

[22] L. Koskela, *Test Driven: TDD and Acceptance TDD for Java Developers*, Manning, 2007

[23] J. Shore and S. Warden, *The Art of Agile Development*, O'Reilly, 2007

[24] D. Koenig, A. Glover, P. King and G. Laforge, *Groovy in Action*, Manning, 2007

[25] C. Smith and P. King, *Agile Project Experiences – The Story of Three Little Pigs*, Agile 2008

[26] J. Link, *Unit testing in Java, How Tests Drive the Code*, Morgan Hoffman, 2003