IMPROVING TESTABILITY WITH STATELESS METHOD EXTRACTION

by

Olivier Dagenais

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
April, 2012

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-91492-2

Our file Notre référence
ISBN: 978-0-494-91492-2

Abstract

The unit testing of object-oriented classes is complicated by the presence of hidden mutable state, since the state of an instance may depend on its entire history of input. This difficulty is known as "the state problem" and causes unit tests to contain a lot of test set-up code, making them more fragile and complex. This thesis presents the *stateless method extraction* "testability refactoring", which is designed to overcome the state problem by extracting simple methods that operate only on their parameters, communicate results only through their return values, have no side-effects and whose functionality can be directly verified by unit tests. An empirical evaluation compares our approach to three other strategies, using three open-source projects and measuring three metrics for each combination. Our approach produces the best results on average, improving the testability of the classes under test while keeping their public interface intact.

Acknowledgements

I would like to thank Marie-Noëlle Houston, Eric Nadeau, James Hague and Keith Bell for their insightful review comments and suggestions.

I would also like to thank Dr. Dwight Deugo for his guidance and help with distilling my research and findings into a suitable thesis format.

Last, but not least, I would like to thank my lovely wife Natalie for all her support and help.

Table of Contents

List of Tables

Listings

Chapter 1

Introduction

1.1 Introduction

Our thesis is that testability of business logic can be improved with stateless method extraction. In this chapter, we introduce the "state problem" and the motivation for solving it. We also introduce how we will be evaluating our stateless method extraction approach and the metrics we will use to evaluate and measure testability improvements using it. We conclude with an overview of the rest of the thesis.

1.2 Problem

Software has advanced considerably in the last several years, but has software quality kept up? We do not always know how good a program or component is, or if it is free of defects, and if those defects will affect the program, directly or otherwise. Software defects can range from the benign, such as incorrect colours and typographical errors in the interface or output, to the very serious, such as causing the destruction of a rocket carrying four satellites [55, 2] or the death of several patients [53].

Programming languages and compilers have evolved to add features that make it easier to write code, although those features do not always make it easier to test the same code. Object-oriented programming (OOP), for example, has introduced constructs and paradigms designed to make abstraction, inheritance and polymorphism easier, but at the same time has introduced a feature that can make testing more

difficult: encapsulation (information hiding) in the form of instance state. Encapsulation allows an object to hide its implementation details so as to present a simpler interface to callers. Indeed, Alan Kay describes this simpler, higher-level interaction between objects as "goals", replacing low-level assignments: "Again, the whole point of OOP is not to have to worry about what is inside an object." [49]

Hidden and mutable instance state poses a problem for automated unit testing because that instance state may come into play when exercising a specific scenario in business logic [19]. Since the instance state is mutable, the values may not be initialized at instance construction or can change as instance methods are called. Because the instance state is hidden, there exists, by definition, no way to manipulate this instance state directly. Separately, hidden (but immutable) instance state or mutable (but visible) instance state do not provide too many difficulties for test writing. It is the combination of the hidden and mutable properties that makes it difficult to write tests.

1.2.1 Hidden Mutable State, Described

Hidden mutable state (HMS) manifests itself in different forms:

- Instance state that cannot be set from the constructor or a factory method.
- Instance state that cannot be directly set from a setter method or property.
- A constructor or direct factory method is not available. For example, an instance can only be created as a by-product of a method call on another class.
- Instance state points to objects with their own hidden mutable state.

In short, a class with hidden mutable state has variables or fields that must be set or modified indirectly as a result of calling a method that, although not necessarily directly related, modifies one or more variables or fields.

Hidden mutable state results in difficult-to-test code, which means it remains untested, despite the many automated test frameworks and systems available [15]. This is likely due to a trade-off between ease of maintenance and testability, where testability was ultimately not favoured.

We still want to test the code, but we must first surmount the difficult-to-test obstacle that hidden mutable state introduces:

> "It is often difficult to control the pretest state of the [Implementation Under Test (IUT)]. The instance variables of the IUT are encapsulated and often composed of still more uncontrollable objects. The interface of the IUT is typically insufficient for testing purposes. (...) Although existing IUT methods can be used to control state, they often do not provide all access needed and can produce spurious test results if they are buggy. They may not even exist, if their development has been deferred to a later increment or they are part of an abstract class. (...) Observing the post-test state of the IUT is often difficult, for the same reasons that controlling the pretest state is difficult." [19]

1.2.2 Business Logic, Defined

Meszaros defines business logic as "The core logic related to the domain model of a business system." [61] For the purposes of this thesis, business logic will refer to the conditions and computations involving state changes, such as code that prevents the deposit of a negative amount into a bank account or code that converts degrees into radians before performing a rotation. Such business rules and processes often read their inputs from and write their outputs to instance state.

In the process of testing the business logic in a method that uses hidden instance state (as input, output or both), it usually becomes necessary to write the test such that it will indirectly write to this state to implement specific scenarios.

Listing 1.1: C# showing the `SmokeDetector` class with HMS

```
1  public class SmokeDetector {
2    public const double Level = 0.3;
3    public const int Danger = 5, WaitTime = 20;
4    private int time = 0, offTime = 0, detected = 0;
5    private bool alarmOn = false, waiting = false;
6
7    public int Cycle(double level) {
8      int signal = 0;
9      time++;
10     if (level > Level && detected < Danger)
11       detected++;
12     else if (detected > 0)
13       detected--;
14
15     if (!alarmOn && detected == Danger) {
16       alarmOn = true;
17       signal = 1;
18     }
19
20     if (alarmOn) {
21       if (!waiting && detected == 0) {
22         waiting = true;
23         offTime = time + WaitTime;
24       }
25       if (waiting && detected == Danger)
26         waiting = false;
27       if (waiting && time > offTime) {
28         waiting = false;
29         alarmOn = false;
30         signal = -1;
31       }
32     }
33     return signal;
34   }
35 }
```

This makes the test method difficult to maintain because the test will be more unwieldy and less obvious—especially in the "arrange" phase [70], also called the "setup" phase [61]—which may discourage the creation of such tests.

An example of the state problem can be seen in listings 1.1 and 1.2. They demonstrate how having all the instance fields marked as `private` forces any test wishing

Listing 1.2: C# showing the test class for the `SmokeDetector`, with the repeated calls to the `Cycle()` method

```
1  using NUnit.Framework;
2
3  [TestFixture]
4  public class SmokeDetectorTest {
5
6    [Test]
7    public void OnAfterFiveThenOffAfterFivePlusTwenty () {
8      var detector = new SmokeDetector();
9
10     // 4 seconds at level of 0.6 accumulates detection
11     for (int i = 0; i < 4; i++) {
12       Assert.AreEqual (0, detector.Cycle (0.6));
13     }
14
15     // the 5th second at level 0.6 triggers alarm
16     Assert.AreEqual (1, detector.Cycle (0.6));
17
18     // alarm sounds while detection decays
19     // through 5 seconds at level 0.1
20     for (int i = 0; i < 5; i++) {
21       Assert.AreEqual (0, detector.Cycle (0.1));
22     }
23
24     // alarm sounds for another 20 seconds
25     for (int i = 0; i < 20; i++) {
26       Assert.AreEqual (0, detector.Cycle (0.1));
27     }
28
29     // 20 seconds without breaching level means
30     // it is safe to turn off alarm
31     Assert.AreEqual (-1, detector.Cycle (0.1));
32   }
33 }
```

to bring the object to a certain state—such as "alarm sounds while detection decays"—to repeatedly call the `Cycle()` method with various values provided to the `level` parameter. By the same token, the only observable state for verification is the return value of the method, although, in this particular instance, this should be sufficient for most test scenarios.

1.3 Motivation

Lest we could somehow perform the practically impossible "complete testing" (in other words, exhaustively trying all possible inputs), testing can only show the presence of defects and not their absence [28]. A good set of automated tests, on the other hand, can catch software regressions that may be introduced through the addition of a feature or the removal of a defect. Finding—and fixing—regressions as early as possible ensures the fixing cost is minimized [75, 65]. A good set of automated tests is henceforth defined as one that can catch most software regressions. Meszaros calls this goal "Bug Repellent" [61].

A metric commonly used to represent confidence in a set of automated tests is code coverage percentage, usually in the form of statement coverage [74]. Code that is covered might be—but is not necessarily—tested, but uncovered code is definitely untested code [20]. Shipping software with untested code is like serving a dish that has not yet been tasted.

HMS does not just present trouble to tests written by programmers—henceforth referred to as human-generated tests (HGT)—but also to computer-generated tests (CGT), such as those generated through Evolutionary Testing (ET), a form of Search Based Software Testing. Indeed, McMinn reported on the difficulty HMS brought to ET:

> "States can cause a variety of problems for ET, since test goals involving states can be dependent on the entire history of input to the test object, as well as just the current input." [60]

If HMS presents an obstacle to unit testing some business logic, it is possible to work around the HMS with a noteworthy "arrange" (or "setup") phase, but this is not always desirable as it can increase the maintenance overhead and costs to write the unit tests, both for human- and computer-generated tests. There are other obvious

options available, such as increasing the visibility of the mutable state or converting the state from mutable to read-only. While it is possible to perform the former without changing the public interface of the IUT, the latter would very likely change the public interface, not to mention the associated risk in performing non-trivial modifications without the safety net of sufficient tests. Fortunately, there exists an option with better results.

One perhaps less obvious option is the extraction of a subset of the state*ful* functionality in the form of a state*less* method which has all its inputs and outputs available to the test driver, while sourcing those inputs from and outputs to the very instance fields they represent within the implementation itself. An example of this can be seen in listings 1.3 and 1.4. The original implementation of the `Cycle()` method has been trivially extracted into a static overload that accepts all its input and emits all its output through parameters and the method return value[1]. The `AccumulateDetection()` test method can then be written to supply values for all parameters, while keeping the fields that the parameters would be initialized from, safe from modification. Indeed, the previous test is still there, unchanged, since the `SmokeDetector`'s public interface did not change as a result of performing this transformation.

If we could overcome the unit testing obstacles brought on by HMS, we would be able to write more and better tests, thus enhancing our ability for catching regressions as our software projects evolve and saving time and effort.

[1]The `ref` keyword in C# causes the method argument to be passed by reference, such that "any change to the parameter in the method is reflected in the underlying argument variable in the calling method." [10]

Listing 1.3: C# showing the SmokeDetector class with the static Cycle() method overload extracted

```
 1  public class SmokeDetector {
 2    public const double Level = 0.3;
 3    public const int Danger = 5, WaitTime = 20;
 4    private int time = 0, offTime = 0, detected = 0;
 5    private bool alarmOn = false, waiting = false;
 6
 7    public int Cycle(double level) {
 8      return Cycle(level, ref time, ref offTime,
 9        ref detected, ref alarmOn, ref waiting);
10    }
11
12    internal static int Cycle(double level,
13      ref int time, ref int offTime, ref int detected,
14      ref bool alarmOn, ref bool waiting) {
15      int signal = 0;
16      time++;
17      if (level > Level && detected < Danger)
18        detected++;
19      else if (detected > 0)
20        detected--;
21
22      if (!alarmOn && detected == Danger) {
23        alarmOn = true;
24        signal = 1;
25      }
26
27      if (alarmOn) {
28        if (!waiting && detected == 0) {
29          waiting = true;
30          offTime = time + WaitTime;
31        }
32        if (waiting && detected == Danger)
33          waiting = false;
34        if (waiting && time > offTime) {
35          waiting = false;
36          alarmOn = false;
37          signal = -1;
38        }
39      }
40      return signal;
41    }
42  }
```

Listing 1.4: C# showing the test class for `SmokeDetector` exploiting the new `Cycle()` method overload

```
1  using NUnit.Framework;
2
3  [TestFixture]
4  public class SmokeDetectorTest {
5
6    [Test]
7    public void AccumulateDetection() {
8      int time = 0, offTime = 0, detected = 0;
9      bool alarmOn = false, waiting = false;
10     for (int i = 1; i <= 4; i++)
11     {
12       int actual = SmokeDetector.Cycle (0.6,
13         ref time, ref offTime, ref detected,
14         ref alarmOn, ref waiting);
15       Assert.AreEqual (0, actual);
16       Assert.AreEqual (i, time);
17       Assert.AreEqual (i, detected);
18     }
19   }
20
21   [Test]
22   public void OnAfterFiveThenOffAfterFivePlusTwenty () {
23     var detector = new SmokeDetector();
24
25     // 4 seconds at level of 0.6 accumulates detection
26     for (int i = 0; i < 4; i++) {
27       Assert.AreEqual (0, detector.Cycle (0.6));
28     }
29
30     // the 5th second at level 0.6 triggers alarm
31     Assert.AreEqual (1, detector.Cycle (0.6));
32
33     // alarm sounds while detection decays
34     // through 5 seconds at level 0.1
35     for (int i = 0; i < 5; i++) {
36       Assert.AreEqual (0, detector.Cycle (0.1));
37     }
38
39     // alarm sounds for another 20 seconds
40     for (int i = 0; i < 20; i++) {
41       Assert.AreEqual (0, detector.Cycle (0.1));
42     }
43
44     // 20 seconds without breaching level means
45     // it is safe to turn off alarm
46     Assert.AreEqual (-1, detector.Cycle (0.1));
47   }
48 }
```

1.4 Goal

If we are to overcome the difficulties of HMS when testing business logic, can we do so with minimal changes to the public interface and with minimal risk of introducing regressions? Can the unit tests be kept parsimonious, yet cover all the important edge cases? Is there something we can do to make code containing hidden mutable state more easily testable?

The goal of this thesis is to improve the testability of business logic in classes that suffer from the "state problem"—because they contain HMS. This is to be done by increasing the maintainability of the accompanying tests while minimally affecting the functionality and the public interface of the code under test. The goal is to be validated according to the following metrics:

1. Complexity of the unit tests.
2. Number of changes to the public interface of the class under test.
3. Percentage of branches covered using concolic testing [52].

1.5 Objectives

To fulfill the general goal of improving testability of business logic in HMS-afflicted code, we have the following objectives:

1. Design and describe a testability refactoring [40] called "stateless method extraction" (SME).

   The SME technique would transform the source code of the implementation under test (IUT) to introduce a functional, or "externally state-free", software layer to specifically work around many of the testability problems of HMS and to generally increase the testability of the code under test. This technique will

be applied manually by the author, although it should be possible to automate it.

2. Design and implement an automated evaluation.

   The evaluation will measure the three metrics of the goal on 4 competing strategies (including SME), using 3 open-source projects.

3. Implement the Public Interface Comparer tool.

   As a suitable tool to enumerate the changes in public interface between two versions of a .NET project output could not be found, one will be written.

4. Independently apply techniques on the open-source projects.

   SME and 3 other techniques will be applied to evaluate their relative effectivenesses. After the techniques have been applied to their respective copies of the IUT, unit tests will be written by the author for each copy, exploiting their relevant features in order to write the simplest tests possible.

5. Run the evaluation and collect the results for comparison and discussion.

1.6 Outline

The remainder of the thesis is organized as follows: chapter 2 introduces the problem domain, describes related concepts, related work and concludes with how the work fits in context. Chapter 3 covers the design, suitability and applicability of the proposed testability refactoring, including examples. Chapter 4 introduces a mechanism by which to evaluate the proposed testability refactoring, followed by the results of the evaluation and their correlation to the goal. The document ends with chapter 5 where the work is summarized, the results discussed and further work is proposed.

1.7 Summary

In this chapter, we introduced the state problem, the specific difficulties related to hidden mutable state and our approach as a possible solution. We also described the three metrics we will be using to evaluate our approach along with specific objectives for the design of the approach and its evaluation. We concluded with an overview of the rest of this thesis.

Chapter 2

Background

2.1 Introduction

This chapter presents an overview of the related concepts from object-oriented (OO) programming to automated testing to computer-generated tests, introduces related work, establishes context within that work and defines the scope of the thesis.

2.2 Object Oriented Programming

This thesis focuses on testing programs written with OO languages—as opposed to procedural or functional languages—because the intent is to solve a problem specific to many OO languages: hidden mutable state. Although procedural languages (such as C) allow programs to have mutable state, the state is not usually hidden. Functional languages (such as Lisp) on the other hand, only allow programs to be written using immutable state through the use of set-once variables. Pure procedural and functional languages are therefore excluded from consideration.

2.2.1 Terminology

OO languages will generally have a basic set of features, the most important of which are described here for clarity:

class The definition for an object.

instance A run-time manifestation of a class, each of which has its own scope.

static A scope for the definition of a class.

field A variable associated with a class (static field) or with an instance (instance field).

method A set of operations attached to a class (static method) or to an instance (instance method).

constructor A special method that is called when an instance of a class is created.

factory method A method whose purpose is to create instances of a class, but not necessarily an instance of the class in which it is found.

overload Two (or more) methods that have the same name, but different parameter numbers or types are said to be *overloads* of the same method.

interface A contract (usually in the form of a list of methods) for classes to implement which enables their instances to be interchangeable with other implementations of the interface.

property A method whose purpose is to allow the reading or reading and writing of a field's value. Some languages support this directly while others offer a convention of "getter" and "setter" methods, instead.

2.3 Automated Testing

A proper introduction to automated testing, especially as it pertains to OO programs, would fill a book and indeed it has. The reader is encouraged to consult *Testing Object-Oriented Systems* by Robert V. Binder [19] for details beyond this chapter's overview.

Generally, a test is said to be automated if there exists a program that can: start the IUT or a subset thereof, initialize some suitable inputs for the scenario to

be verified, provide these inputs to the IUT and then verify the resulting output, without any intervening assistance.

2.3.1 Testing frameworks

Automated testing frameworks provide developers with the ability to write and organize test scenarios, execute them sequentially and independently from one another and finally report on their successes.

This thesis focuses on testing frameworks [18] that allow the test development environment to be identical to the implementation development environment, collectively called xUnit [61]. A single test scenario can be implemented as a *test method* in a *test class*, exercising functionality in the method under test (MUT) in another class, the class under test (CUT). A collection of test classes is known as a *test suite.*

Test methods will generally have at least three phases: arrange, act and assert (Meszaros calls them *setup*, *exercise* and *verify*, plus adds *teardown* [61]). The functionality in a test method is not always separated as such and might instead be intermixed or even be located elsewhere in the test class. The *arrange* phase concerns itself with preparing the CUT and/or the environment, effectively setting the indirect inputs for the MUT and optionally initializing complex direct inputs. The *act* phase contains code that calls the MUT with its direct inputs and collects the direct outputs, if any. The *assert* phase derives any indirect outputs, if any, and verifies the *actual* outputs against the *expected* outputs. The optional fourth phase consists of cleaning up the environment, as necessary.

The terms *test inputs* and *test outputs* will henceforth be used to refer to both their direct and indirect sets. It is worth noting that because exhaustive testing is not feasible, test suites will usually contain scenarios that sample only a fraction of the possible input space. In those cases, the test inputs will have been selected to represent typical use, boundary conditions and other potential areas for defects.

Such classes of values for the test inputs earns them the "interesting and relevant" designation.

2.3.2 Test scope

As alluded to in the previous sub-section, this thesis concerns itself with automated testing, at the lowest level, by focusing the test scenarios on methods, which is also known as *unit testing*. This is in contrast to *integration testing* which concerns itself with the interaction between a class and another class and/or some external environment such as a file system or a network, and *subsystem testing* which involves observing the interaction between a class and several other classes and/or external components such as a database or a web service.

2.3.3 Code coverage

An IUT can be automatically *instrumented* to record which statements and expressions (collectively known as *sequence points*) are executed—or *visited*—during an automated test run. The resulting report is known as *statement coverage* and will usually correlate its results with the original source code by highlighting the statements and expressions that were executed [20]. This report will also include enough data to determine the coverage percentage, which is computed as *visited sequence points* (VSP) divided by *total sequence points* (TSP). The report usually aggregates this information at the method, class, namespace and program levels [74].

Code coverage can be used to determine the completeness of the test suite or, more precisely, to identify areas of the IUT that are untouched by the tests. The presence of uncovered code can also reveal surprises in the implementation, although its best use remains the identification of missing test scenarios.

2.4 Computer-Generated Tests

2.4.1 Motivation for Computer-Generated Tests

It may not always be possible or feasible to have a programmer write automated tests for a component or an implementation. Reasons provided by Bühler and Wegener [24] include the simultaneous satisfaction of functional, real-time and safety requirements as well as the very large test space resulting from the interaction of 50-70 independently-developed embedded systems. The nature of the assembled system—in this case a premium vehicle—also makes manual testing, such as driving the vehicle in an environment suitable for each scenario, prohibitively expensive. Bühler and Wegener go on to suggest generating tests automatically to work around these obstacles and thus reduce costs. Evolutionary testing (ET), "the application of evolutionary computation to test automation" [24], is therefore suggested to automatically and efficiently sample the test input space. ET will usually be implemented using genetic algorithms (GA), which are well-suited to a mix of exploration and exploitation in the search space [57].

Binder writes:

> "Automatic test generation can be used without having to pregenerate expected results. Although generating input values automatically is easy, generating the corresponding expected results automatically is usually difficult." [21]

Even if tests are automatically generated without the use of an oracle (human or otherwise), the generated test suite can still serve as a baseline and could be used to detect changes in the outputs as a result of adding features or fixing defects. These changes in the outputs could indicate a regression, just as changes in the outputs of human-generated tests would.

In the present case, there's an additional reason computer-generated tests are interesting: we can use an automatic structural test generator to objectively compare the testability of source code under various transformations. Sub-section 4.2.2 will introduce how the coverage of CGT can be interpreted as a measure of testability and sub-section 4.5.3 will show the results of that approach.

More details about the various flavours and uses of computer-generated tests are provided in the following sub-sections.

2.4.2 Approaches to Computer-Generated Tests

There are three major approaches in generating tests: Random Testing, Search Based Software Testing and Concolic Testing [52] are described in the following sub-sections.

Random Testing

Random testing (RT), as the name implies, consists of generating inputs based on random numbers and is thus undirected. RT does not perform as well as ET [86], although it has been used as a baseline for comparing other test generators [52].

Search Based Software Testing

Search-based software testing (SBST) is the application of a metaheuristic search technique (MST) to the problem of generating test inputs [57]. The technique is based around the optimization of an *objective function*, trying to improve an initial solution based on feedback from the objective function—either by maximizing or minimizing its value, depending on the desired outcome—effectively making it a guided quest for global optima in the search space.

For test input generation, the search space is the domain of the test inputs and the objective function is carefully constructed to reward interesting and relevant test inputs. The nature of "interesting and relevant" depends on the *intent* of the testing

exercise. In general, the biggest difficulty of SBST lies in the definition of the objective function [16]. Consult the following section for the major intents of SBST.

The general flow of MST implementations is to start with one or more "initial solutions", created either with domain knowledge or at random, and apply the objective function on the solution(s) to establish a solution score. If no solution score exceeds a pre-defined threshold, tweak the solution(s) according to some heuristic and repeat the process with the new solution(s) until either no improvement in score is detected or some pre-defined time or iteration threshold is reached.

The simplest heuristic is called "hill climbing", a strategy that focuses on the best objective function value in the neighbourhood of the current solution's search space. This greedy, exploit-only strategy will often cause a hill climber to get stuck in local optima.

An improvement to hill climbing was proposed in the "simulated annealing" heuristic whereby solutions with worse objective function values will be considered with a probability decreasing with the number of iterations, allowing more of the search space to be explored in the beginning and ending with a hill climber-like exploitation.

One of the better MST heuristics is "genetic algorithms (GA)", a class of "evolutionary algorithms". It is designed to simulate the process of natural selection (sometimes known as "survival of the fittest") by modelling a *population* of solutions as chromosomes and providing ways for pairs of solutions to "breed" through recombination (splicing part of one parent with part of the other and also combining the remaining two parts), providing the next generation of solutions. The simulation is made more authentic through the use of mutations (random, low-probability alterations to the chromosome) and various reproduction selection mechanisms, such as fitness-proportional selection and tournament selection. Through this simulation, solutions will appear to evolve across the generations and, given appropriately-tuned

parameters, the population of solutions will be diverse and "fit", thus providing a good balance of exploration and exploitation of the search space.

Concolic Testing

Concolic testing (CT) is the combination of concrete and symbolic execution applied to the problem of test data generation. It is also known as dynamic symbolic execution [57].

Symbolic execution (SE) is a simulated execution of a given path through the program using symbols in place of input values, and keeping track of expressions instead of concrete values as assignments and operations take place [50]. For a program operating exclusively with integers, one could say SE is the algebraic equivalent of the arithmetic operations that would be evaluated with concrete execution. The resulting output of SE is generally a system of constraints on the inputs for executing the given path. .

Concrete execution—running an instrumented version of the program—can be combined with SE such that the two executions are correlated to verify the generated constraints against concrete values. Often times, this correlation allows the simplification of these constraints, for example by removing non-linear sub-expressions [57]. A constraint solver can then be used to find test input values that cause the execution of specific paths through the IUT [78].

2.4.3 Intents of Computer-Generated Tests

There are four major uses for Computer-Generated Tests (CGT) described in the following sub-sections. They differ mostly in the choice of executable statements to target and are not usually specific to any CGT approach in particular. The exception is non-functional testing, which does not target any executable statement in particular and is more suited to an optimization approach (such as SBST).

Structural Testing

> testing that takes into account the internal mechanism of a system or component. Syn: glass-box testing, white-box testing [12]

Coupling an automated test generator with a code coverage tool makes it possible to discover which inputs can be used to reach otherwise hard-to-cover areas as well as areas of the code where no inputs can be found to reach them (within a pre-defined search time). This could indicate that the code is unreachable because it contains, for example, conflicting pre-conditions. CGT intended for structural testing will generally attempt to execute all statements of an IUT at least once.

Functional Testing

> 1. testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions 2. testing conducted to evaluate the compliance of a system or component with specified functional requirements. Syn: black-box testing [12]

Functional testing can even be performed by encoding the specification's requirements as a component of a simulator. For example, Bühler and Wegener evaluated the functionality of a brake assist system using a simulator [24]. The simulator would report if the safety requirements were violated by the brake assist system, such as instances where the system either did not enhance the driver's brake torque in a critical situation or provided too much braking in a non-critical situation.

Grey-Box Testing

> Grey-box testing combines both structural and functional information for the purposes of testing. [57]

The objectives of grey-box testing are to attempt to get assertions, already embedded in a program, to be violated and to trigger exception conditions. Some of this work even temporarily turned specially-formatted comments into executable assertions for the purposes of targeting their fault branches for execution [57].

Non-Functional Testing

Non-functional testing concerns itself with verifying non-functional requirements[1], such as constraints on execution time, working memory, long-term storage and processor load, usually for embedded and real-time systems. Test inputs are sought which cause the program to exceed pre-defined thresholds, such as taking more time than usual or consuming more memory than expected. This can be measured by instrumentation external to the IUT, such as the operating system's timers and counters, or by using a simulator [77].

2.5 Related Work

The following sub-sections are introduced by quotes relevant to their topic.

2.5.1 The State Problem

> "Side-effects, by which I mean changes to the values stored in fields of objects or elements of arrays, are clearly intended to be used frequently in Java. However, the presence of side-effects can make it harder to reason about a program, because there is invisble [sic] state to the side of computations which changes. That means a reader needs to keep track of both the visible aspects of a program and the hidden values off to the side that may change." [39]

[1] a software requirement that describes not what the software will do but how the software will do it. Syn: design constraints, nonfunctional requirement [12]

> "Although encapsulation does not directly contribute to the occurrence of bugs, it can present an obstacle to testing. Testing requires accurate reporting of the concrete and abstract states of an object and the ability to set state easily. Object-oriented languages make it difficult—if not impossible—to directly set or get the concrete state." [22]

McMinn identified the "State Problem" in the context of evolutionary testing and described a few possible solutions that enhanced the ET system to compensate for problems brought on by state [60]. Difficulties with internal state are also mentioned by Bühler and Wegener [24], Harman et al. [43], Tillman and de Halleux [78] and Baresel et al. [16].

Difficulties associated with internal state can be reduced or worked around with better analysis, such as data dependency analysis [51] or with data flow testing [63]. Because data dependency analysis can often be time-consuming, techniques such as program slicing [79] were developed to help reduce the search space. McMinn went on to develop a technique that enhances the search to look for method call sequences and combines the chaining approach with evolutionary search [58].

Another approach to generate tests for programs with state—specifically those written in OO programming languages—was proposed by Wappler and Wegener [85]. Their two-phase technique, called "strongly-typed genetic programming", consists of first using genetic programming to generate a sequence of methods to call (creating instances of classes as necessary), then using a genetic algorithm to find suitable values for the parameters of the generated method calls.

Binder suggests the following four approaches to work around a programming language's visibility constraints:

1. Private Access Driver

> "A Private Access Driver may be developed only in a language that

> offers the ability to bypass normal scoping/visibility rules, providing visibility of all elements of a class implementation to a client class or subclass." [19]

While C++ has the `friend` feature which can grant the test class full access to the CUT's members, Java and C# do not have such a feature. The closest equivalent is a reflection-based solution to access hidden members[2].

2. Test Control Interface

> "Use special-purpose language mechanisms that make all elements of a class's implementation visible to a driver to achieve necessary control of pretest state and observation of post-test state." [19]

This approach is applicable to Objective-C and Smalltalk/ENVY which make it possible to extend the CUT with methods that have access to the hidden members of the CUT. The closest equivalent in C# would be the use of the `partial` keyword that allows classes to be defined across multiple files, however a parial-based CUT extension would have to be defined in the same project as the CUT, given that test classes are, by convention, defined in a separate project.

3. Drone

> "Use multiple inheritance to compose a test package class that combines the driver and the class under test." [19]

This approach is very limited because it relies on a programming language having both multiple inheritance and private member access from subclasses. It appears only Eiffel supports this approach.

[2]For example, Microsoft provides the `PrivateObject` class with the Visual Studio Testing Tools to simplify this approach: `http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.privateobject.aspx`

4. Built-in Test Driver

> "Implement a test suite and a test driver wholly containd within the class under test. Provide an interface to activate and exercise the tests." [19]

Adding test methods directly to the CUT or to an inner class thereof is the main idea of this approach, which side-steps the hidden member problem (inner classes have access to all of their outer class's members) but also provides the potential for interference from the test code.

Meszaros' "Test Utility Methods" pattern [61] is similar in intent to SME as it tries to overcome the difficulties in setting the inputs and reading the outputs of an MUT. Although Meszaros' technique also adds methods that accept the logic under test's direct inputs and return the direct outputs, the difference is that the complexity is simply hidden in the test utility methods, whereas SME will not have such a complex intermediary method between the test method and the method under test.

2.5.2 The Case for Functional Programming

> "However, the functional style is also perfectly adapted to incremental testing: programs written in this style can also be *tested* one function at a time. When a function neither examines nor alters external state, any bugs will appear immediately. Such a function can affect the outside world only through its return values. Insofar as these are what you expected, you can trust the code which produced them." [38]

"Imperative functional programming" is promoted by Reddy [69] and appears to push the idea of a *public* functional programming interface built on top of imperative features. This is in contrast to the present thesis, which would see an *internal* functional

programming interface (built on top of imperative features) as a building block for a possibly pre-existing object-oriented interface.

Van-Roy and Haridi support the spirit of our approach, declaring "We find that a good rule of thumb for complex systems is to keep as many components as possible declarative. State should not be 'smeared out' over many components. It should be concentrated in just a few carefully selected components." [66] Support is also provided by Wampler: "Side-effect-free functions make excellent building blocks for reuse, since they don't depend on the context in which they run. Compared to functions with side effects, they are also easier to design, comprehend, optimize, and test." [84]

Hevery, on the other hand, appears to disagree with our approach [46] and argues for a pure object-oriented approach [48]. His technique of choice for increasing testability is the addition of "test seams" [46]. Haahr nevertheless argues for "referential transparency" [39] to not only make computations easier to understand but also easier to parallelise.

2.5.3 The Case for Testability Transformations

> "It could be said that the algorithm for side-effect removal creates an 'almost functional' language, in which all state changes are expressed using the assignment statement. This would return the programming paradigm to that considered by the initial work on the Axiomatic Method, where axiomatic semantics is comparatively elegant and easy to define and use. This, perhaps, provides further additional anecdotal evidence that side-effects are hard to reason about and that their removal is worthwhile for comprehension." [44]

Because programs are not always easily testable as written, it has been suggested by Harman et al. to automatically create an alternate (and temporary) version of the program that is easier to analyze or test by applying some transformations [41].

These transformations can be as simple as "flag removal"; the inline expansion of a boolean variable with its underlying expression [42]. Another example would be the removal of expressions with side-effects [44].

The general idea of testability transformation was further developed by Korel et al. [51] as well as McMinn et al. [59], among others. Program slicing could also be considered a testability transformation [43]. Harman eventually goes one step further and suggests "testability refactoring": a permanent program transformation that has the simultaneous goals of making the program easier to test and better (or at least no worse) for the programmer [40].

The technique proposed in this document consists of a sequence of transformations that are all refactorings; "altering [an existing body of code's] internal structure without changing its external behavior" [31]. SME will be applied to the IUT's source code in a transparent way to consumers of the code, except for the new methods that will be extracted and made available to write small, focused unit tests. Meszaros, on the other hand, suggests refactorings that are mostly applied to the source code of the tests themselves [61].

SME requires no new language features or extensions to be created and the source code to the IUT can be compiled as it was before, with no additional steps. This is in contrast to the TXL source transformation language [27], whose purpose is to augment a programming language with syntactic extensions. TXL works by transforming a program written in an augmented language into a program for the original programming language before being compiled.

2.5.4 The Case for Simplicity

> "A low cyclomatic complexity generally indicates a method that is easy to understand, test, and maintain." [6]

> "(...) long tests are often Obscure Tests (page 186) and tend not to provide very good Defect Localization (see page 22) when they fail partway through the test." [61]

McCabe introduced the concept of "cyclomatic complexity" with the intention of evaluating the difficulty involved when testing programs based on their structure [56]. Indeed, McCabe even discussed the possibility of reducing a program's structure to reduce the testing effort, with the intent of writing software that is both testable and maintainable. Basili et al. [17] showed that the *Response For a Class (RFC)* metric[3] is correlated to the probability of finding faults for a class, suggesting smaller methods will be less likely to have defects.

Hevery created the Testability Explorer tool to compute the "Non-Mockable Total Recursive Cyclomatic Complexity" metric, compute the "Law of Demeter" metric [54] as well as count the number of global mutable fields in Java programs [47]. The first metric is an evolution of McCabe's metric for OO programs while the other two metrics also help to reduce the testing effort through smaller call chains (Demeter) and less "spooky action at a distance" (surprise side-effects) by reducing the use of mutable global fields [45].

2.6 Scope

The thesis focuses on addressing the glass-box unit testing difficulties introduced by hidden mutable state, in the context of object-oriented software, through the use of stateless method extraction. In other words, the implementation under test is available for inspection as tests are written on a per-method basis for classes that contain private instance fields whose values are altered by calls to instance methods.

[3]RFC is defined as "The number of methods that can potentially be executed in response to a message received."

Considerations outside the scope of this thesis therefore include:

- Opaque-box testing
- Integration (or other high-level) testing
- Non-object-oriented software

Additional considerations outside the scope of this thesis include:

- Testing of hidden state mutations. These must still be tested, but those tests are not covered by this thesis.
- Run-time performance of IUT or unit tests
- Performance of SBST (such as ET) on IUT

Stateless method extraction is to be compared against 3 other strategies for unit testing OO software containing hidden mutable state; table 2.1 describes precisely the strategies covered.

Strategies	Known effects		Covered
	Affects IUT or tests	Affects public interface	
Do nothing	neither	N	Y
Stateless method extraction	IUT	N	Y
Making mutable state visible	IUT	N	Y
Making state and methods visible	IUT	N	Y
Making hidden state immutable	IUT	Y	N
"Test Utility Methods"	tests	N	N
"Private Access Driver"	IUT	N	N
"Test Control Interface"	both	N	N
"Drone"	tests	N	N
"Built-in Test Driver"	both	N	N
Adding "test seams"	IUT	Y	N

Table 2.1: A matrix of known strategies designed to improve testability, their known effects and whether they are covered by this thesis.

2.7 Summary

This chapter introduced the reader to representative terminology and domain concepts before describing more advanced concepts, related work and finally how the thesis fits into the existing research.

Chapter 3

Approach

3.1 Introduction

This chapter describes our approach to improving the testability of classes that suffer from the state problem. Our approach takes the form of the manual application of our SME testability refactoring and we describe several examples of its use, followed by discussions on when SME can be used as well as how. We conclude with a discussion of important decisions made.

3.2 Design

Stateless method extraction's intent is to increase the testability of an IUT by separating the interaction between the IUT's hidden mutable state and the inputs and outputs of that IUT's instance methods. This decoupling allows programmers to first focus their unit tests on the pure computation aspects—in other words, the business logic—then on the object-oriented layer above the stateless methods. This document covers the former while the latter is out of scope, the domain of the likes of Spec Explorer [81].

It is important to note that SME relies on the fact that all its steps will only modify the IUT's source code in ways that will not alter any of the behaviours and functionality present before SME was performed. That is to say, all the operations performed should be pure refactorings [31] and no functionality changes are to be performed during the process. These rules help ensure the use of SME is invisible to

end-users and has the least chance of introducing defects or regressions in the affected methods.

3.2.1 Overview

Stateless method extraction is performed in six high-level steps, applied repeatedly until no more candidate blocks can be identified:

1. Identification of a candidate block.

   A "candidate block" is a functionality subset of an instance method that looks promising to turn into a stateless method.

2. Re-ordering of incidental operations.

   On some occasions, it may be necessary to move some statements out of the candidate block that are not relevant to the functionality subset and/or whose ordering is not critical (such as logging).

3. Assignment of inputs and outputs to local variables.

   Identify the new method's input and output parameters, then create local variables for them.

4. Simplification of inputs.

   Optimize the input parameters to be native or built-in types.

5. Extraction of the candidate block into a method.

   The candidate block should now be easily converted into a stateless method.

6. Re-inline any variables created to facilitate a refactor.

   This reverses the actions of step #3 so the refactor has as little impact on the original code as possible and only contains the necessary changes.

Listing 3.1: Pseudo-code showing the general procedure for extracting stateless methods

```
Procedure extract_stateless_method(instance_method)
  While (block = find_candidate_block(instance_method)) != null
    If contains_incidental_operations(block)
      If incidental_operations_can_be_re-ordered(block)
        re-order_incidental_operations(block)
      Else
        # the second half will be picked up by the next iteration
        split_block(block)
      End If
    End If
    introduce_local_variables(block)
    Until parameters_and_return_values_are_simple_types(block)
      shrink_candidate_block(block)
      introduce_local_variables(block)
    Repeat
    extract_candidate_block(block)
    re-inline_local_variables(instance_method)
  End While
End Procedure
```

Listing 3.1 shows a pseudo-code version of the stateless method extraction procedure. The following sub-sections will cover the individual steps in greater detail while section 3.6 will explain the decisions behind them.

3.2.2 Step 1: Identification of a Candidate Block

The first step of SME consists of identifying, in an instance method, one or more functionality subsets that would be good candidates for extraction and subsequent unit testing. The idea is to look for units of computation (i.e. business logic) that read from and/or write to hidden mutable state. It is not necessary to extract the entire body of an instance method if there are parts of the method that would be more suitable to extract and unit test separately. It is also possible a method will contain no candidate blocks to extract; the technique need not be forcibly applied if no unit testing advantages can be realized.

The criteria for establishing candidacy of a block consist of:

1. A sequence of related stateless statements.

   Stateless statements are defined as depending directly on their inputs and without side-effects. Relatedness of the statements can be determined by data dependency analysis, which can be performed informally using the "highlight references" features of modern development tools. Statements that interpret input parameters to interact with the environment (such as reading/writing files and sending/receiving network traffic) are not considered stateless. Statements may produce effects (such as assignment and return values) but may not cause other values to change as a result of execution (such as modifying HMS), as this would defeat the purpose. Section 3.2.3 describes how to deal with non-stateless statements.

2. Inputs (local variables, fields and/or method parameters): at least 1, but no more than 6.

3. Outputs (local variables, fields and/or return values): at least 1, but no more than 3. At least one non-trivial result must be produced by processing the input(s).

Categories of interesting candidate blocks include:

- Implementations of mathematical formulas.

  For example, converting from degrees to radians or computing the great circle distance (the shortest path between two points on a sphere), both of which are shown being performed inline in listing 3.2, while they have been respectively extracted into the `ToRadians()` and `CalculateGreatCircleDistance()` methods in listing 3.3.

- Processing of a single element where such processing currently takes place while looping through a sequence of elements.

  For a before-after example, refer to listings 3.4 and 3.5.

- String parsing, especially when using a regular expression. This allows the parsing to be separated from the use of its results, so that the parsing and the subsequent computations can be tested separately.

- Filling in of templates.

  This can be considered the reverse of parsing and usually involves combining a few inputs to yield a meaningful construction, such as a string representation of an object. An example of this category can be found in the `GenerateItemNode()` method extracted in listing 3.5. Here an `XElement` instance is created (on line 36) and its attributes initialized from two of the supplied parameters (lines 37 and 38) as well as from a value derived from the last two parameters (line 45).

Listing 3.2: C# showing the `ComputeTourLength()` method in the `FlyingSalespersonProblem` class with some inlined mathematical formulas

```
 1  public class FlyingSalespersonProblem {
 2    private const double AverageEarthRadius = 6372.8;
 3    private readonly int _numberOfCities;
 4    private readonly IDictionary<int, LatLon> _cities;
 5    private readonly IList<int> _visitingOrder;
 6
 7    public FlyingSalespersonProblem
 8      (ICollection<double[]> cities) {
 9      _numberOfCities = cities.Count;
10      _cities =
11        new Dictionary<int, LatLon>(_numberOfCities);
12      _visitingOrder = new List<int>(_numberOfCities);
13      foreach (double[] triplet in cities)
14      {
15        var point = new LatLon(triplet[1], triplet[2]);
16        _cities.Add ((int) triplet[0], point);
17        _visitingOrder.Add(0);
18      }
19    }
20
21    public IList<int> VisitingOrder
22    { get { return _visitingOrder; } }
23
24    public double ComputeTourLength() {
25      double tourLength = 0;
26      for (int i = 0; i < _numberOfCities; i++)
27      {
28        int j = (i + 1) % _numberOfCities;
29        var from = _cities[_visitingOrder[i]];
30        var to = _cities[_visitingOrder[j]];
31        var delta1 = Math.PI * from.Lat / 180.0;
32        var lambda1 = Math.PI * from.Lon / 180.0;
33        var delta2 = Math.PI * to.Lat / 180.0;
34        var lambda2 = Math.PI * to.Lon / 180.0;
35
36        tourLength += AverageEarthRadius
37          * Math.Acos(
38              Math.Cos(delta1)
39            * Math.Cos(delta2)
40            * Math.Cos(lambda1 - lambda2)
41            + Math.Sin(delta1) * Math.Sin(delta2)
42          );
43      }
44      return tourLength;
45    }
46  }
```

Listing 3.3: C# showing the `ComputeTourLength()` method with the `ToRadians()` and `CalculateGreatCircleDistance()` methods that were extracted from it

```
1   public double ComputeTourLength()
2   {
3     double tourLength = 0;
4     for (int i = 0; i < _numberOfCities; i++)
5     {
6       int j = (i + 1) % _numberOfCities;
7       var from = _cities[_visitingOrder[i]];
8       var to = _cities[_visitingOrder[j]];
9       tourLength +=
10        CalculateGreatCircleDistance(
11          from.Lat, from.Lon, to.Lat, to.Lon);
12    }
13    return tourLength;
14  }
15
16  internal static double CalculateGreatCircleDistance
17    (double fromLat, double fromLon, double toLat, double toLon)
18  {
19    var delta1 = ToRadians(fromLat);
20    var lambda1 = ToRadians(fromLon);
21    var delta2 = ToRadians(toLat);
22    var lambda2 = ToRadians(toLon);
23
24    return AverageEarthRadius
25      * Math.Acos(
26        Math.Cos(delta1)
27        * Math.Cos(delta2)
28        * Math.Cos(lambda1 - lambda2)
29        + Math.Sin(delta1) * Math.Sin(delta2)
30      );
31  }
32
33  internal static double ToRadians(double degrees)
34  {
35    return Math.PI * degrees / 180.0;
36  }
```

Listing 3.4: C# showing the `GenerateXml()` method with a candidate block between lines 19-30

```
 1  public XElement GenerateXml
 2    (string relativePathToRoot)
 3  {
 4    var itemsNode =
 5      new XElement(DeepZoom2008 + "Items");
 6    var collectionNode =
 7      new XElement(DeepZoom2008 + "Collection",
 8      new XAttribute("MaxLevel", 7),
 9      new XAttribute("TileSize", 256),
10      new XAttribute("Format", _imageFormatName),
11      itemsNode
12    );
13
14    var mortonNumber = 0;
15    var maxPostId = 0;
16    foreach (var postId in _postIds)
17    {
18      maxPostId = Math.Max(maxPostId, postId);
19      // <I N="0" Id="351" Source="../0351.dzi"/>
20      var itemNode = new XElement(ItemNodeName);
21      itemNode.SetAttributeValue("N", mortonNumber);
22      itemNode.SetAttributeValue("Id", postId);
23      var fileName =
24        postId.ToString(_fileNameIdFormat);
25      var relativeDziSubPath =
26        Path.ChangeExtension(fileName, "dzi");
27      var relativeDziPath = Path.Combine(
28        relativePathToRoot, relativeDziSubPath);
29      itemNode.SetAttributeValue("Source",
30        relativeDziPath);
31      itemNode.Add(_sizeNode);
32      itemsNode.Add(itemNode);
33      mortonNumber++;
34    }
35
36    collectionNode.SetAttributeValue("NextItemId",
37      maxPostId + 1);
38    return collectionNode;
39  }
```

Listing 3.5: C# showing the `GenerateXml()` method with the candidate block extracted as the `GenerateItemNode()` method

```
 1  public XElement GenerateXml
 2    (string relativePathToRoot)
 3  {
 4    var itemsNode =
 5      new XElement(DeepZoom2008 + "Items");
 6    var collectionNode =
 7      new XElement(DeepZoom2008 + "Collection",
 8      new XAttribute("MaxLevel", 7),
 9      new XAttribute("TileSize", 256),
10      new XAttribute("Format", _imageFormatName),
11      itemsNode
12    );
13
14    var mortonNumber = 0;
15    var maxPostId = 0;
16    foreach (var postId in _postIds)
17    {
18      maxPostId = Math.Max(maxPostId, postId);
19      var itemNode = GenerateItemNode(mortonNumber,
20        postId, _fileNameIdFormat, relativePathToRoot);
21      itemNode.Add(_sizeNode);
22      itemsNode.Add(itemNode);
23      mortonNumber++;
24    }
25
26    collectionNode.SetAttributeValue("NextItemId",
27      maxPostId + 1);
28    return collectionNode;
29  }
30
31  internal static XElement GenerateItemNode
32    (int mortonNumber, int postId,
33    string fileNameIdFormat, string relativePathToRoot)
34  {
35    // <I N="0" Id="351" Source="../0351.dzi"/>
36    var itemNode = new XElement(ItemNodeName);
37    itemNode.SetAttributeValue("N", mortonNumber);
38    itemNode.SetAttributeValue("Id", postId);
39    var fileName =
40      postId.ToString(fileNameIdFormat);
41    var relativeDziSubPath =
42      Path.ChangeExtension(fileName, "dzi");
43    var relativeDziPath = Path.Combine(
44      relativePathToRoot, relativeDziSubPath);
45    itemNode.SetAttributeValue("Source",
46      relativeDziPath);
47    return itemNode;
48  }
```

It follows that there are uninteresting categories of blocks, such as:

- Trivial state changes.

  A good example is a `Reset()` method that sets a number of instance fields to a specific set of values, such as constants. The whole point of such a method is to mutate hidden state and thus there would likely be very little functionality that could be tested in isolation.

- Too many inputs and/or outputs.

  This category could be considered a more general case of the previous category, but there are instances where it may not make sense to break up some processing for readability, maintenance or performance reasons.

- Not enough outputs.

  An example of a block in this category would be an instance method calling instance methods on its fields in such a way that there are no observable changes to the fields' own hidden mutable state. Extracting that block would likely provide no unit testing advantage and is thus uninteresting.

- Too many interactions with the environment.

  If an instance method consists mostly of interactions with external entities such as a file system or a database, there may be very little functionality left that is stateless that would benefit from extraction and isolated unit testing.

3.2.3 Step 2: Re-ordering of Incidental Operations

The second SME step is about finding and, if necessary, moving statements that are either unrelated to a computation or statements that are not stateless (such as those that cause side-effects) out of the candidate block. This should only take place if the original instance method's functionality will not be adversely affected

by the move. If the functionality would indeed be adversely affected by the re-ordering of unrelated or non-stateless statements, the candidate block should be split such that its new endpoint is just before the unrelated or non-stateless statement. A subsequent candidate block could be attempted just after the unrelated or non-stateless statement.

An example of unrelated statements can be seen in listing 3.6, specifically in the candidate block identified between lines 18 and 33. The majority of the operations in the block concern themselves with the creation and configuration of the `itemNode` instance, while at least 3 of them perform tasks incidental to the `itemNode` configuration: lines 21, 23 and 25. Line 20 does augment `itemNode`, but does so with a trivial operation (the `Add()` method call) and so its inclusion in the candidate block could be argued either way. For the purposes of this example, line 20 will be considered incidental. The candidate block could be shrunk by re-ordering all the operations identified as incidental without affecting the functionality. In particular, the `mortonNumber++;` operation on line 23 is only sensitive to ordering insofar as `itemNode.SetAttributeValue("N", mortonNumber);` on line 22 is concerned and thus could be moved down and out of the candidate block. Lines 20 and 21 depend on one another as well as line 19 (the creation of the `itemNode` instance), but the behaviour would be the same if they appeared at the end of the block (the XML API used in the example guarantees this) and therefore, because they have no advantage being located immediately after the creation of `itemNode`, they can be safely moved out of the candidate block. The last incidental operation is `maxPostId = Math.Max(maxPostId, postId);` on line 25. Its only dependency is the `postId` loop variable and thus could be moved up to be the very first operation in the loop. Listing 3.4 illustrates the result of re-ordering the lines containing incidental operations and the smaller, simpler candidate block obtained between lines 19 and 30.

Listing 3.6: C# showing the `GenerateXmlIntermixed()` method with a candidate block between lines 18-33

```
 1  public XElement GenerateXmlIntermixed
 2    (string relativePathToRoot)
 3  {
 4    var itemsNode =
 5      new XElement(DeepZoom2008 + "Items");
 6    var collectionNode =
 7      new XElement(DeepZoom2008 + "Collection",
 8      new XAttribute("MaxLevel", 7),
 9      new XAttribute("TileSize", 256),
10      new XAttribute("Format", _imageFormatName),
11      itemsNode
12    );
13
14    var mortonNumber = 0;
15    var maxPostId = 0;
16    foreach (var postId in _postIds)
17    {
18      // <I N="0" Id="351" Source="../0351.dzi"/>
19      var itemNode = new XElement(ItemNodeName);
20      itemNode.Add(_sizeNode);
21      itemsNode.Add(itemNode);
22      itemNode.SetAttributeValue("N", mortonNumber);
23      mortonNumber++;
24      itemNode.SetAttributeValue("Id", postId);
25      maxPostId = Math.Max(maxPostId, postId);
26      var fileName =
27        postId.ToString(_fileNameIdFormat);
28      var relativeDziSubPath =
29        Path.ChangeExtension(fileName, "dzi");
30      var relativeDziPath = Path.Combine(
31        relativePathToRoot, relativeDziSubPath);
32      itemNode.SetAttributeValue("Source",
33        relativeDziPath);
34    }
35
36    collectionNode.SetAttributeValue("NextItemId",
37      maxPostId + 1);
38    return collectionNode;
39  }
```

Listing 3.7: C# showing common code used by listings 3.8 and 3.9

```
1  public class Info {
2    public int Revision { get; set; }
3    public Uri Url { get; set; }
4    public Uri Root { get; set; }
5    public Guid Uuid { get; set; }
6  }
7
8  public partial class SubversionClient
9  {
10   private string ExecuteSvn(string arguments)
11   {
12     string output;
13     using (var p = new Process())
14     {
15       p.StartInfo.UseShellExecute = false;
16       p.StartInfo.RedirectStandardOutput = true;
17       p.StartInfo.FileName = _pathToSvnProgram;
18       p.StartInfo.Arguments = arguments;
19       p.Start();
20       output = p.StandardOutput.ReadToEnd();
21       p.WaitForExit();
22     }
23     return output;
24   }
25 }
```

A statement is said to be non-stateless if values or data not explicitly specified by parameters or the return value are modified as a result of execution. This can be as simple as an instance method modifying hidden mutable state, a method modifying global state or a method interpreting its parameters for interaction with the environment, such as using a string parameter as the name of a file to create or delete, or the name of a database table to read from or write to. Non-stateless operations will generally be more difficult to re-order because, by definition, they cause or depend on side-effects and standard data dependency analysis will be made difficult because it would have to be aware of private instance field changes.

Listing 3.8: C# showing the `LoadInfo()` method with a candidate block between lines 7-32

```
1  public partial class SubversionClient {
2    private readonly string _pathToSvnProgram;
3    public SubversionClient(string pathToSvnProgram)
4    { _pathToSvnProgram = pathToSvnProgram; }
5
6    public Info LoadInfo(string pathToWorkingCopy) {
7      var sb = new StringBuilder();
8      sb.Append("info").Append(' ');
9      sb.Append('"');
10     sb.Append(pathToWorkingCopy);
11     sb.Append('"').Append(' ');
12     sb.Append("--xml");
13     var arguments = sb.ToString();
14
15     var output = ExecuteSvn(arguments);
16
17     var doc = new XmlDocument();
18     doc.LoadXml(output);
19
20     var urlNode = doc.SelectSingleNode("/info/entry/url");
21     var rootNode = doc.SelectSingleNode(
22       "/info/entry/repository/root");
23     var uuidNode = doc.SelectSingleNode(
24       "/info/entry/repository/uuid");
25     var revisionNode = doc.SelectSingleNode("/info/entry/@revision");
26
27     var result = new Info {
28       Revision = Convert.ToInt32(revisionNode.Value, 10),
29       Url = new Uri(urlNode.InnerText),
30       Root = new Uri(rootNode.InnerText),
31       Uuid = new Guid(uuidNode.InnerText),
32     };
33     return result;
34   }
35 }
```

An example of a non-stateless method in the middle of a candidate block can be seen at line 15 of listing 3.8. The `ExecuteSvn()` method, defined at line 10 of listing 3.7, starts a sub-process and returns the contents of `StandardOutput` and thus is said to interact with the environment, making it definitely non-stateless. In this particular case, because the call to the `ExecuteSvn()` method depends on the

computations immediately above it and provides input needed by the computations immediately below it, it can not be moved before or after the candidate block (lines 7-32). This causes the original candidate block to be split into parts excluding the immovable, non-stateless method with the first half being lines 7-13 and the second half lines 17-32. Listing 3.9 shows the two stateless methods that were extracted, respectively, from the first and second candidate blocks: `CreateInfoArguments()` and `ParseInfoFromXml()`.

Listing 3.9: C# showing the `CreateInfoArguments()` and `ParseInfoFromXml()` stateless methods that were extracted from the `LoadInfo()` method

```
1  public partial class SubversionClient {
2    private readonly string _pathToSvnProgram;
3    public SubversionClient(string pathToSvnProgram)
4    { _pathToSvnProgram = pathToSvnProgram; }
5
6    internal static string CreateInfoArguments
7      (string pathToWorkingCopy) {
8      var sb = new StringBuilder();
9      sb.Append("info").Append(' ');
10     sb.Append('"');
11     sb.Append(pathToWorkingCopy);
12     sb.Append('"').Append(' ');
13     sb.Append("--xml");
14     return sb.ToString();
15   }
16
17   public Info LoadInfo(string pathToWorkingCopy) {
18     var arguments = CreateInfoArguments(pathToWorkingCopy);
19
20     var output = ExecuteSvn(arguments);
21
22     var result = ParseInfoFromXml(output);
23     return result;
24   }
25
26   internal static Info ParseInfoFromXml(string output) {
27     var doc = new XmlDocument();
28     doc.LoadXml(output);
29
30     var urlNode = doc.SelectSingleNode("/info/entry/url");
31     var rootNode = doc.SelectSingleNode(
32       "/info/entry/repository/root");
33     var uuidNode = doc.SelectSingleNode(
34       "/info/entry/repository/uuid");
35     var revisionNode = doc.SelectSingleNode("/info/entry/@revision");
36
37     var result = new Info {
38       Revision = Convert.ToInt32(revisionNode.Value, 10),
39       Url = new Uri(urlNode.InnerText),
40       Root = new Uri(rootNode.InnerText),
41       Uuid = new Guid(uuidNode.InnerText),
42     };
43     return result;
44   }
45 }
```

Listing 3.10: C# showing the `ToPolar()` method with interleaved computations for `r` and `theta`

```
 1  public class PolarCoordinate {
 2    private readonly double _r, _theta;
 3    public PolarCoordinate(double r, double theta)
 4    { _r = r; _theta = theta; }
 5    public double R { get { return _r; } }
 6    public double Theta { get { return _theta; } }
 7  }
 8
 9  public class CartesianCoordinate {
10    private readonly double _x, _y;
11    public CartesianCoordinate(double x, double y)
12    { _x = x; _y = y; }
13
14    public PolarCoordinate ToPolar() {
15      var xSquared = _x * _x;
16      var radians = Math.Atan2(_y, _x);
17      var ySquared = _y * _y;
18      var theta = radians * (180 / Math.PI);
19      var r = Math.Sqrt(xSquared + ySquared);
20      return new PolarCoordinate(r, theta);
21    }
22  }
```

It could also be the case that the original instance method is computing several outputs simultaneously and such statements are intermixed. If the outputs are independently-computed (or their computations share some intermediate results, then diverge), it may be worth investigating the separation of their computations by re-ordering statements such that they can then be extracted, and subsequently unit tested, independently. An example of intermixed computations can be seen in listing 3.10 between lines 15 and 19, while another version of the method with the computations separated through re-ordering can be seen in listing 3.11.

Incidental operation re-ordering can be considered a form of program slicing [79] in that data dependency analysis [51] can be used to identify a subset of computations of a method responsible for one of the outputs and isolate them. Assuming all the discarded operations were stateless, it would be safe to say that the remaining

Listing 3.11: C# showing the `ToPolar()` method with the computations for `r` and `theta` separated into their own consecutive sequences

```
 1  public class PolarCoordinate {
 2    private readonly double _r, _theta;
 3    public PolarCoordinate(double r, double theta)
 4    { _r = r; _theta = theta; }
 5    public double R { get { return _r; } }
 6    public double Theta { get { return _theta; } }
 7  }
 8
 9  public class CartesianCoordinate {
10    private readonly double _x, _y;
11    public CartesianCoordinate(double x, double y)
12    { _x = x; _y = y; }
13
14    public PolarCoordinate ToPolar() {
15      var xSquared = _x * _x;
16      var ySquared = _y * _y;
17      var r = Math.Sqrt(xSquared + ySquared);
18      var radians = Math.Atan2(_y, _x);
19      var theta = radians * (180 / Math.PI);
20      return new PolarCoordinate(r, theta);
21    }
22  }
```

operations can execute one immediately after the other and still produce the same result, the same way lines 15, 17 and 19 in listing 3.10 are necessary for the computation of `r` and can be re-arranged to be consecutive, as seen in lines 15, 16 and 17 in listing 3.11.

3.2.4 Step 3: Assignment of Inputs and Outputs to Local Variables

The third step concerns the temporary introduction of local variables for each of the input and output parameters, as per the "introduce explaining variable" refactoring [34]. This makes the upcoming stateless method parameters and return values explicit so that it is easier to provide them with suitable names, easier to evaluate their simplicity (consult section 3.2.5 for details) and to more clearly delineate the candidate block so that it will be easier to extract it using existing "extract method"

Listing 3.12: C# showing the original `Plane` class computing a cross-product and a dot-product inline

```
 1  public class Plane
 2  {
 3    private readonly Vector _p1, _p2, _p3;
 4    public Plane(Vector p1, Vector p2, Vector p3)
 5    {
 6      _p1 = p1; _p2 = p2; _p3 = p3;
 7    }
 8    public double DistanceFromPlane(Vector p)
 9    {
10      var a = new Vector(_p2.X - _p1.X, _p2.Y - _p1.Y, _p2.Z - _p1.Z);
11      var b = new Vector(_p3.X - _p1.X, _p3.Y - _p1.Y, _p3.Z - _p1.Z);
12      var n = new Vector(
13        a.Y * b.Z - a.Z * b.Y,
14        a.Z * b.X - a.X * b.Z,
15        a.X * b.Y - a.Y * b.X);
16      var length = Math.Sqrt(n.X * n.X + n.Y * n.Y + n.Z * n.Z);
17      var normal = new Vector(n.X/length, n.Y/length, n.Z/length);
18      var v = new Vector(p.X - _p1.X, p.Y - _p1.Y, p.Z - _p1.Z);
19      var distance = v.X * normal.X + v.Y * normal.Y + v.Z * normal.Z;
20      return distance;
21    }
22  }
```

refactoring tools.

The local variables introduced during this step should be remembered to allow their re-inlining to be performed again at the end of the current iteration, as described in section 3.2.7.

3.2.5 Step 4: Simplification of Inputs

Stateless method extraction's fourth step entails finding the simplest types that can be used for the parameters and return values without causing an undue increase in their numbers. Any type with an easy construction (a single call to a constructor or a factory method) and without side-effects while reading its component values (typical of an immutable instance) is considered "simple" for the purpose of SME.

As a first example of possible input simplification, refer to listing 3.12 where line 19

Listing 3.13: C# showing the `Plane` class with the `CrossProduct()` and `DotProduct()` stateless methods extracted from it

```
1  public class Plane
2  {
3    private readonly Vector _p1, _p2, _p3;
4    public Plane(Vector p1, Vector p2, Vector p3)
5    {
6      _p1 = p1; _p2 = p2; _p3 = p3;
7    }
8    public double DistanceFromPlane(Vector p)
9    {
10     var a = new Vector(_p2.X - _p1.X, _p2.Y - _p1.Y, _p2.Z - _p1.Z);
11     var b = new Vector(_p3.X - _p1.X, _p3.Y - _p1.Y, _p3.Z - _p1.Z);
12     var n = CrossProduct(a, b);
13     var length = Math.Sqrt(n.X * n.X + n.Y * n.Y + n.Z * n.Z);
14     var normal = new Vector(n.X/length, n.Y/length, n.Z/length);
15     var v = new Vector(p.X - _p1.X, p.Y - _p1.Y, p.Z - _p1.Z);
16     var distance = DotProduct(v, normal);
17     return distance;
18   }
19   internal static Vector CrossProduct(Vector a, Vector b)
20   {
21     return new Vector(
22       a.Y * b.Z - a.Z * b.Y,
23       a.Z * b.X - a.X * b.Z,
24       a.X * b.Y - a.Y * b.X);
25   }
26   internal static double DotProduct(Vector a, Vector b)
27   {
28     return a.X * b.X + a.Y * b.Y + a.Z * b.Z;
29   }
30 }
```

computes the dot product of two 3D vectors. This candidate block has, as parameters, two instances of a `Vector` type. The stateless method that was extracted as lines 26-29 of listing 3.13 could also have been written to accept 6 parameters representing both triplets of the vectors' values. Assuming a `Vector` instance can be created in one statement (a call to its constructor, in this case) and reading its component values can cause no side-effects, it may be worth keeping the input parameters as two `Vector` instances for readability reasons. It is worth noting that an argument could also be made to make the new `DotProduct()` method accept the 6 component

Listing 3.14: C# showing the original `CodeBlockModifier` class

```
1  public class CodeBlockModifier : BlockModifier
2  {
3    public override string ModifyLine(string line)
4    {
5      // Replace "@...@" zones with "<code>" tags.
6      line = Regex.Replace(line,
7        @"(?<before>^|([\s\([{]))" + // before
8         "@" +
9        @"(\|(?<lang>\w+)\|)?" +    // lang
10        "(?<code>[^@]+)" +         // code
11        "@" +
12       @"(?<after>$|([\]}])|(?=" +
13         Globals.PunctuationPattern +
14         @"{1,2}|\s|$))",  // after
15       CodeFormatMatchEvaluator);
16     return line;
17   }
18
19   static public string CodeFormatMatchEvaluator
20     (Match m)
21   {
22     string res = m.Groups["before"].Value + "<code";
23     if (m.Groups["lang"].Length > 0)
24       res += " language=\"" +
25         m.Groups["lang"].Value + "\"";
26     res += ">" + m.Groups["code"].Value +
27       "</code>" + m.Groups["after"].Value;
28     return res;
29   }
30 }
```

values so that the method could be subsequently re-used to calculate dot products of vectors not expressed using instances of `Vector`.

Listing 3.15: C# showing the `CodeBlockModifier` class after SME

```
1  public class CodeBlockModifier : BlockModifier
2  {
3    private const string Pattern =
4        @"(?<before>^|([\s\([{]))" + // before
5         "@" +
6        @"(\|(?<lang>\w+)\|)?" +    // lang
7         "(?<code>[^@]+)" +        // code
8         "@" +
9        @"(?<after>$|([\]}])|(?=" +
10         Globals.PunctuationPattern +
11         @"{1,2}|\s|$))";  // after
12   internal static readonly Regex CodeBlockRegex =
13     new Regex(Pattern);
14
15   public override string ModifyLine(string line)
16   {
17     // Replace "@...@" zones with "<code>" tags.
18     line = CodeBlockRegex.Replace(line,
19       CodeFormatMatchEvaluator);
20     return line;
21   }
22
23   static public string CodeFormatMatchEvaluator
24     (Match m)
25   {
26     return BuildCodeElementString(
27       m.Groups["before"].Value,
28       m.Groups["lang"].Value,
29       m.Groups["code"].Value,
30       m.Groups["after"].Value
31     );
32   }
33
34   internal static string BuildCodeElementString
35     (string before, string lang,
36     string code, string after)
37   {
38     string res = before + "<code";
39     if (lang.Length > 0)
40       res += " language=\"" + lang + "\"";
41     res += ">" + code + "</code>" + after;
42     return res;
43   }
44 }
```

On the other hand, suppose the candidate block's purpose is to reformat text based on the results of a regular expression match, as shown between lines 22-27 in listing 3.14. In this case, because the input parameter is of a type that is difficult to create[1], it is easy to argue that the method's parameters should be four strings (consult lines 34-43 of listing 3.15 to see the extracted method) instead of one `Match` instance, given how trivial strings are to create versus the complexity involved in creating instances of the `Match` class. Indeed, as listing 3.16 shows, the original method must either be tested indirectly (through calling the `ModifyLine()` method) or by creating a `Match` instance (in a rather pathological way, lest the original regular expression pattern is refactored for re-use as in lines 12-13 of listing 3.15), while listing 3.17 shows the extracted stateless `BuildCodeElementString()` method can be tested directly.

3.2.6 Step 5: Extraction of the Candidate Block Into a Method

The fifth step consists of replacing the candidate block with a call to a new method into which the contents of the candidate block was moved. The names and types of the parameters will come from the local variables introduced in step 3, as will the names and types of the new method's local variables representing the return values. The name of the extracted method should describe how the parameters are processed to produce the return values, although the new method could become an overload of the original method.

The newly-extracted method, by virtue of the candidate block processing in the previous steps, should only read from its input parameters and write to its return values, otherwise it would not qualify as [externally] "stateless". To help prevent the accidental interaction with any instance fields, the method should be marked as being

[1]The `Match` class has no visible constructors. The only way to create an instance is through the return value of the `Match()` method overloads on the `Regex` class.

Listing 3.16: C# showing two ways to test the `CodeFormatMatchEvaluator()` method

```
1    [Test]
2    public void ModifyLine() {
3      var cbm = new CodeBlockModifier();
4      const string input = "Call the @|ruby|r_tohtml();@ method";
5      var actual = cbm.ModifyLine(input);
6      const string expected =
7        "Call the <code language=\"ruby\">r_tohtml();</code> method";
8      Assert.AreEqual(expected, actual);
9    }
10
11   [Test]
12   public void CodeFormatMatchEvaluator() {
13     var m = Regex.Match(
14       "Call the ruby r_tohtml(); method",
15       @"(?<before>Call\sthe\s)" +
16       @"(?<lang>ruby)\s" +
17       @"(?<code>r_tohtml\(\);)" +
18       @"(?<after>\smethod)"
19     );
20     var actual = CodeBlockModifier.CodeFormatMatchEvaluator(m);
21     const string expected =
22       "Call the <code language=\"ruby\">r_tohtml();</code> method";
23     Assert.AreEqual(expected, actual);
24   }
```

scoped to the class (as opposed to being scoped to instances), which is done in many programming languages by decorating the method with the `static` keyword.

The "extract method" refactoring [32] is implemented as a built-in interactive tool in development environments [1, 5], which makes this step almost trivial.

3.2.7 Step 6: Re-inline any Variables Created to Facilitate a Refactor

The SME technique's sixth and final step cleans up the temporary refactoring performed as part of step 3. This is done by performing the "inline temp" refactoring [33], an inline expansion of the local variables introduced to represent the new method's input parameters and return values.

Listing 3.17: C# showing the parsing and formatting functionality tested directly and separately

```
1    [Test]
2    public void ParseZone()
3    {
4      const string input = "Call the [@|ruby|r_tohtml();@] method";
5      var actual = CodeBlockModifier.CodeBlockRegex.Match(input);
6      Assert.AreEqual("[", actual.Groups["before"].Value);
7      Assert.AreEqual("ruby", actual.Groups["lang"].Value);
8      Assert.AreEqual("r_tohtml();", actual.Groups["code"].Value);
9      Assert.AreEqual("]", actual.Groups["after"].Value);
10   }
11
12   [Test]
13   public void BuildCodeElementString()
14   {
15     var actual =
16       CodeBlockModifier.BuildCodeElementString(
17         "Call the ",
18         "ruby",
19         "r_tohtml();",
20         " method"
21       );
22     const string expected =
23       "Call the <code language=\"ruby\">r_tohtml();</code> method";
24     Assert.AreEqual(expected, actual);
25   }
```

3.2.8 End-to-end Detailed Walk-through

The following example demonstrates the application of SME on a simplified version of the `FootNoteFormatterState` class from the *Textile.NET* open-source project [25]. First, the original implementation is presented in listing 3.18, along with the source code to its parent class in listing 3.19. Second, a unit test will be attempted against the original implementation. SME will then be applied and finally new unit tests will be written against the transformed implementation.

Analysis and corresponding unit test

As we can see in listing 3.18, the `FootNoteFormatterState` class is part of a framework and must fulfill a contract by implementing some methods from its base class, `SimpleBlockFormatterState` (see listing 3.19). One of these methods is `OnContextAcquired()` (see lines 31-35 of listing 3.18), which reads from the `Tag` property, parses it with a regular expression and writes the result to the `m_noteID` field, a classic example of interaction with HMS. We will focus on this method for the purposes of testing it, starting with an analysis of its inputs and outputs, so that we can feed it test data and verify its results.

Looking at listing 3.19, we see that the `Tag` property's underlying value comes from the `m_tag` field (line 32), which is only written to at line 16, as part of the `Consume()` method. We see that the `Consume()` method also calls the `OnContextAcquired()` method, so our test can call the `Consume()` method to invoke the `OnContextAcquired()` method indirectly. Even if the test class subclassed `FootNoteFormatterState` to be able to call `OnContextAcquired()` directly (due to its visibility of `protected`), it would not be sufficient because `m_tag` is only written to by a call to `Consume()`. The `Consume()` method's parameter is an instance of the `Match` class, which has no visible constructors and can only be obtained from a call to one of the `Match()` method overloads on the `Regex` class. As for the

Listing 3.18: C# showing the original FootNoteFormatterState class

```
1  public class FootNoteFormatterState
2    : SimpleBlockFormatterState
3  {
4    private int m_noteID = 0;
5
6    public FootNoteFormatterState(TextWriter w)
7      : base(w)
8    {
9    }
10
11   public override void Enter()
12   {
13     Writer.Write(
14       string.Format(
15         "<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
16         m_noteID,
17         FormattedStylesAndAlignment(),
18         m_noteID));
19   }
20
21   public override void Exit()
22   {
23     Writer.WriteLine("</p>");
24   }
25
26   public override void FormatLine(string input)
27   {
28     Writer.Write(input);
29   }
30
31   protected override void OnContextAcquired()
32   {
33     Match m = Regex.Match(Tag, @"^fn(?<id>[0-9]+)");
34     m_noteID = Int32.Parse(m.Groups["id"].Value);
35   }
36 }
```

Listing 3.19: C# showing the `SimpleBlockFormatterState` base class

```
1  public abstract class SimpleBlockFormatterState
2  {
3    protected readonly TextWriter Writer;
4    private string m_tag = null;
5    private string m_atts = null;
6    protected SimpleBlockFormatterState(TextWriter w)
7    {
8      Writer = w;
9    }
10
11   public abstract void Enter();
12   public abstract void Exit();
13   public abstract void FormatLine(string input);
14   public string Consume(Match m)
15   {
16     m_tag = m.Groups["tag"].Value;
17     m_atts = m.Groups["atts"].Value;
18     var input = m.Groups["content"].Value;
19     OnContextAcquired(); Enter();
20     return input;
21   }
22
23   protected abstract void OnContextAcquired();
24   protected string FormattedStylesAndAlignment()
25   {
26     return String.IsNullOrEmpty(m_atts)
27       ? "" : " style=\"" + m_atts + "\"";
28   }
29
30   public string Tag
31   {
32     get { return m_tag; }
33   }
34 }
```

outputs, we see that the m_noteID field is unavailable for direct inspection (due to its visibility of private), however the Enter() method reads the m_noteID field to insert its value into a template, the result of which is written to the TextWriter instance provided at construction. As luck would have it, the Consume() method also calls the Enter() method.

Listing 3.20: C# showing the unit test for the original OnContextAcquired() method

```
 1  [Test]
 2  public void EnterAndOnContextAcquired()
 3  {
 4    // arrange
 5    var output = new StringWriter();
 6    var fnfs = new FootNoteFormatterState(output);
 7    var expression = @"^\s*(?<tag>fn[0-9]+)"
 8      + @"(?:\{(?<atts>[^}]+)\})?"
 9      + @"\.(?:\s+)?(?<content>.*)$";
10    var input = "fn1{color:red}. This is the footnote";
11    Match m = Regex.Match(input, expression);
12
13    // act
14    // Consume() causes OnContextAcquired()
15    // and Enter() to be called
16    fnfs.Consume(m);
17
18    // assert
19    Assert.AreEqual(
20      "<p id=\"fn1\" style=\"color:red\"><sup>1</sup> ",
21      output.ToString());
22  }
```

Armed with the information gathered by this analysis, we can now write a unit test, which can be seen in listing 3.20. As we can see, the arrange phase is rather complex, given it must initialize five variables to satisfy the requirements for all the methods involved. Notice also how the assert phase can verify that the substring "1" we provided inside the input string on line 10 was indeed parsed out by the OnContextAcquired() method and then formatted again as the substring "¹" into the string that was written—through the Enter()

method—to the `TextWriter`-compatible instance we provided on line 6. This level of indirection is at best frustrating and at worse a sufficient obstacle to prevent such unit tests to be written in the first place, encouraging more involved and/or fragile integration tests, if any.

Step 1: Identification of a candidate block

Keeping with our focus on the `OnContextAcquired()` method, the first step of SME would identify almost all the processing that takes place, since matching a regular expression, extracting one of the captured groups and converting a string into an integer are all "stateless statements" and together they produce a non-trivial result. We can also identify one input—the value of the `Tag` property—and one output—assignment to the `m_noteID` field.

Step 2: Re-ordering of incidental operations

All three statements are necessary and related to the computation; in this particular case there are no incidental operations and thus the second SME step results in no actions taken.

Step 3: Assignment of inputs and outputs to local variables

Listing 3.21 shows the `OnContextAcquired()` method after the small transformation called for by the third SME step: the introduction of local variables for inputs and outputs, namely the `tag` variable on line 3 and the `result` variable on line 5.

Step 4: Simplification of inputs

The only input parameter is a string and a string is considered one of the "simple" types, thus the fourth SME step results in no action being taken.

Listing 3.21: C# showing the `OnContextAcquired()` method with additional local variables

```
1    protected override void OnContextAcquired()
2    {
3      string tag = Tag;
4      Match m = Regex.Match(tag, @"^fn(?<id>[0-9]+)");
5      int result = Int32.Parse(m.Groups["id"].Value);
6      m_noteID = result;
7    }
```

Step 5: Extraction of the candidate block into a method

The application of SME's fifth step can be seen in listing 3.22, where the `ParseFootNoteId()` stateless method was introduced based on the candidate block inside the `OnContextAcquired()` method. Notice how the `ParseFootNoteId()` method has been decorated with the visibility of `internal` so that it is not visible to outside callers, while still visible to the tests. Notice also how the method was decorated with the scope of `static` because it no longer needs to read from or write to any instance fields or properties, nor call any instance methods[2].

Listing 3.22: C# showing the `OnContextAcquired()` method with the `ParseFootNoteId()` method extracted

```
1    protected override void OnContextAcquired()
2    {
3      string tag = Tag;
4      int result = ParseFootNoteId(tag);
5      m_noteID = result;
6    }
7
8    internal static int ParseFootNoteId(string input)
9    {
10     Match m = Regex.Match(input, @"^fn(?<id>[0-9]+)");
11     return Int32.Parse(m.Groups["id"].Value);
12   }
```

[2]Microsoft recommends that methods which do not access instance members be marked as static for performance reasons. This is documented as Code Analysis rule *CA1822: Mark members as static* at `http://msdn.microsoft.com/en-us/library/ms245046.aspx`

Step 6: Re-inline any variables created to facilitate a refactor

The sixth step of SME reverses the actions of the third step and inlines the `tag` and `result` temporary variables, as can be seen in listing 3.23.

Listing 3.23: C# showing the `OnContextAcquired()` and `ParseFootNoteId()` methods after SME

```
1    protected override void OnContextAcquired()
2    {
3      m_noteID = ParseFootNoteId(Tag);
4    }
5
6    internal static int ParseFootNoteId(string input)
7    {
8      Match m = Regex.Match(input, @"^fn(?<id>[0-9]+)");
9      return Int32.Parse(m.Groups["id"].Value);
10   }
```

Writing a unit test after SME

Now that SME has been applied on the `OnContextAcquired()` method, our mission to unit test its functionality has been greatly simplified, as the computations have been *outsourced* to the `ParseFootNoteId()` method and `OnContextAcquired()` simply serves as a proxy for said computations, while coordinating access to the HMS. The `ParseFootNoteId()` method's inputs consist of a single string and its return values consist of a single integer, making unit tests almost trivial, as listing 3.24 shows. We can still keep the test shown in listing 3.20 as it tests at a higher level the functionality of `ParseFootNoteId()` and its integration with the HMS, but if a defect is later found in the footnote parsing code, we can write a very short test to expose it instead of a more involved test.

Listing 3.24: C# showing the unit test for the `ParseFootNoteId()` method

```
1  [Test]
2  public void ParseFootNoteId()
3  {
4    // arrange
5    var input = "fn1{color:red}. This is the footnote";
6
7    // act
8    var actual = FootNoteFormatterState.ParseFootNoteId(input);
9
10   // assert
11   Assert.AreEqual(1, actual);
12 }
```

3.2.9 Accelerated Walk-through

The following example demonstrates the application of SME for the purposes of testing the functionality of the `Enter()` method of the `FootNoteFormatterState` class in listing 3.18. We are able to re-use the analysis we used for the `OnContextAcquired()` method and find that a test written against the original version of the class would look a lot like the one we came up with in listing 3.20. The rest of this sub-section will feature listings 3.25, 3.26, 3.27, 3.28 and 3.29 that illustrate—without narrative—the SME steps applied to increase the testability of the `Enter()` method and the unit test that can be written against the newly-extracted `FormatFootNote()` method.

Listing 3.25: C# showing the original `Enter()` method

```
1    public override void Enter()
2    {
3      Writer.Write(
4        string.Format(
5          "<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
6          m_noteID,
7          FormattedStylesAndAlignment(),
8          m_noteID));
9    }
```

Listing 3.26: C# showing the `Enter()` method with temporary variables introduced

```
1    public override void Enter()
2    {
3      int noteId = m_noteID;
4      string atts = FormattedStylesAndAlignment();
5      string result = string.Format(
6        "<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
7        noteId,
8        atts,
9        noteId);
10     Writer.Write(result);
11   }
```

Listing 3.27: C# showing the `FormatFootNote()` method that was extracted from the `Enter()` method

```
1    public override void Enter()
2    {
3      int noteId = m_noteID;
4      string atts = FormattedStylesAndAlignment();
5      string result = FormatFootNote(noteId, atts);
6      Writer.Write(result);
7    }
8
9    internal static string FormatFootNote
10     (int noteId, string atts)
11   {
12     return string.Format(
13       "<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
14       noteId,
15       atts,
16       noteId);
17   }
```

Listing 3.28: C# showing the temporary variables re-inlined in the `Enter()` method

```
1    public override void Enter()
2    {
3      Writer.Write(
4        FormatFootNote(m_noteID,
5          FormattedStylesAndAlignment()));
6    }
7
8    internal static string FormatFootNote
9      (int noteId, string atts)
10   {
11     return string.Format(
12       "<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
13       noteId,
14       atts,
15       noteId);
16   }
```

Listing 3.29: C# showing the unit test for the `FormatFootNote()` method

```
1  [Test]
2  public void FormatFootNote()
3  {
4    // act
5    var actual =
6      FootNoteFormatterState.FormatFootNote
7        (1, " style=\"color:red\"");
8
9    // assert
10   Assert.AreEqual(
11     "<p id=\"fn1\" style=\"color:red\"><sup>1</sup> ",
12     actual);
13 }
```

3.3 Suitability

Although designed to work around the difficulties introduced by HMS, SME is suitable for methods that are difficult to test—even in the absence of HMS—due to unwieldy arrange phases, computation coupling, the presence of side-effects and/or restrictions of visibility. These additional suitabilities are addressed in the following sub-sections.

3.3.1 Unwieldy Arrange Phase

If a unit test method has an *arrange* phase that is longer than 50% of the test method, it is probably an indication that there are too many parameters, the parameters are too difficult to create and/or the functionality under test is buried too deep. SME would help isolate the functionality that is intended to be tested and thus reduce the number of parameters, since candidate blocks are selected to use between 1 and 6 parameters. SME would also help reduce the complexity of the parameters—the input simplification step takes care of this—and expose the computation as a single, direct method call. The arrange phase would then be shrunk considerably, if not entirely removed, leading to simpler unit tests. An example of an unwieldy arrange phase was seen in listing 3.20 and can also be seen in listing 3.30 whereby a test for the *great circle distance* formula [87] in the `FlyingSalespersonProblem` class (refer back to listing 3.2) must initialize a list of cities, an instance of the `FlyingSalespersonProblem` class and a visiting order. Contrast this with the test for the `CalculateGreatCircleDistance()` stateless method in listing 3.31 which only needs to supply the coordinates of the two points.

Listing 3.30: C# showing an indirect test of the *great circle distance* formula used by the `ComputeTourLength()` method

```
1    [Test]
2    public void ComputeTourLength_BnaToLax ()
3    {
4      // arrange
5      var cities = new double[][]
6      {
7        new double[] {1, 36.12, -86.67},
8        new double[] {2, 33.94, -118.40},
9      };
10     var fsp = new FlyingSalespersonProblem(cities);
11     fsp.VisitingOrder[0] = 1;
12     fsp.VisitingOrder[1] = 2;
13
14     // act
15     // since we computed to AND from, we divide by 2
16     var actual = fsp.ComputeTourLength() / 2;
17
18     // assert
19     Assert.AreEqual(2887.26, actual, 0.01);
20   }
```

Listing 3.31: C# showing a direct test of the *great circle distance* formula by calling the `CalculateGreatCircleDistance()` stateless method

```
1    [Test]
2    public void CalculateGreatCircleDistance_BnaToLax()
3    {
4      var actual = FlyingSalespersonProblem.
5        CalculateGreatCircleDistance
6          (36.12, -86.67, 33.94, -118.40);
7      Assert.AreEqual(2887.26, actual, 0.01);
8    }
```

3.3.2 Computation Coupling

A method may be performing several computations in such a way to make it difficult to unit test any such computation on an individual basis, usually due to operations that make it difficult to observe intermediate results. We call this *computation coupling.* For example, the `ModifyLine()` method in listing 3.14 combines parsing and formatting in a single operation[3] while listing 3.15 shows how exposing the `CodeBlockRegex` object and the `BuildCodeElementString()` method makes the functionality more directly reachable. Indeed, consult listing 3.17 to see how this decoupling enables the unit testing of the parsing and the formatting, seen in the `ParseZone()` and `BuildCodeElementString()` test methods, respectively, without needing to always perform both at once. Contrast this to the `ModifyLine()` test method in listing 3.16, which must exercise the parsing and the formatting simultaneously.

In this particular case, the SME technique was extended to include two other extractions: the compile-time `Pattern` string constant representing the regular expression pattern and the `CodeBlockRegex` run-time constant representing the compiled regular expression. This provided the unit tests with access to the same functionality used for performing the work, thereby enhancing the unit tests' realism and relevance. Finally, SME's candidate block identification enabled the separation of parsing and formatting, which allowed both to be tested separately as units, as well as together since the public-facing functionality did not change as a result of the refactorings.

Computation de-coupling might not always be addressable through SME as there could be computations that are tightly-coupled for readability or performance reasons and separating them would lead to a readability or performance loss.

[3]The `Replace()` method will call the `CodeFormatMatchEvaluator()` method to re-format every match.

Listing 3.32: C# showing the original `Decode()` method computing the values of `x` and `y` simultaneously

```
1  public class MortonLayout
2  {
3    private const int NumberOfBits = 32;
4    public static Point Decode(int index)
5    {
6      var bits = new BitArray(new[] {index});
7      int x = 0, y = 0;
8      for (int o = 1, power = 1; o < NumberOfBits; o += 2, power <<= 1)
9      {
10       if (bits[o - 1])
11       {
12         x += power;
13       }
14       if (bits[o])
15       {
16         y += power;
17       }
18     }
19     var result = new Point(x, y);
20     return result;
21   }
22 }
```

An example of two computations that would be worse if separated can be seen in listing 3.32, where we can see that `x` and `y` are computed simultaneously. Compare this to listing 3.33, where `x` and `y` are computed separately, meaning two runs through the loop on `bits` instead of one. It is also less obvious, in the second case, that the values of `x` and `y` were obtained by interpreting interleaved bits of the Morton Layout's `index` [4].

3.3.3 The Presence of Side-effects

A method may not suffer from the state problem (it neither reads from nor writes to instance fields), but nevertheless has side-effects that increase the difficulty when writing unit tests, such as the modification of input parameters or interactions with

Listing 3.33: C# showing the `Decode()` method with the `DecodeAxis()` method extracted out of it

```
 1  public class MortonLayout
 2  {
 3    private const int NumberOfBits = 32;
 4    public static Point Decode(int index)
 5    {
 6      var bits = new BitArray(new[] {index});
 7      var x = DecodeAxis(bits, 1);
 8      var y = DecodeAxis(bits, 0);
 9      var result = new Point(x, y);
10      return result;
11    }
12    internal static int DecodeAxis(BitArray bits, int offset)
13    {
14      var result = 0;
15      for (int o = 1, power = 1; o < NumberOfBits; o += 2, power <<= 1)
16      {
17        if (bits[o - offset])
18        {
19          result += power;
20        }
21      }
22      return result;
23    }
24  }
```

the environment. Stateless method extraction could potentially still take place, provided some candidate blocks can be identified around the code causing the side-effects. This would allow some simple unit tests to be written against the simple stateless methods extracted, while still allowing more involved integration tests against the original method.

An example of a method producing a result and causing side-effects can be seen in listing 3.34. The `CalculateAngle()` method calls the `Normalize()` method on `MutableVector`, which irreversibly scales the parameters to be unit vectors. SME's candidate block identification would exclude both calls to `Normalize()` and would allow a version of the `CalculateAngle()` method to be created that had no side-effects but assumed both parameters were of unit length.

Listing 3.34: C# showing the `CalculateAngle()` method causing side-effects on its parameters

```
1  public class MutableVector
2  {
3    private double _x, _y, _z;
4    public MutableVector(double x, double y, double z)
5    {
6      _x = x; _y = y; _z = z;
7    }
8    public double X { get { return _x; } }
9    public double Y { get { return _y; } }
10   public double Z { get { return _z; } }
11   public void Normalize()
12   {
13     var length = Math.Sqrt(_x * _x + _y * _y + _z * _z);
14     _x = _x / length;
15     _y = _y / length;
16     _z = _z / length;
17   }
18 }
19
20 public class EuclideanVectorSpace
21 {
22   public static double CalculateAngle
23     (MutableVector a, MutableVector b)
24   {
25     a.Normalize();
26     b.Normalize();
27     var theta = Math.Acos(a.X * b.X + a.Y * b.Y + a.Z * b.Z);
28     return theta * 180 / Math.PI;
29   }
30 }
```

3.3.4 Visibility or Access Restrictions

An instance method may already be stateless, but, because of its nature, may be difficult to reach with unit tests. This situation can manifest itself when the class under test subclasses another class to override some of its methods, but those methods are only visible to the superclass and subclasses thereof. This is usually achieved with a visibility or accessibility of `protected`. This can also happen when the class under test explicitly implements an interface (when using C#): the explicitly-implemented interface members will not be visible from the class' default interface [7].

In both cases, SME can help extract one or more stateless methods that will not require an instance of the class to be created and can be directly unit tested, while keeping the original methods with the visibility and accessibility they require.

3.4 Applicability

In general, SME is most applicable to existing or legacy projects that were built with few or no unit tests. As a result, their source code was not written with testability in mind, nor is it possible (for various reasons) to make large, sweeping testability-related changes. The prospect of fixing defects or adding features to such a project, without sufficient tests to catch regressions, would seem risky. SME could be used to introduce small, easily-testable methods—along with some unit tests—to the affected source code before making each fix or adding each new feature, thereby establishing a functionality baseline, if only a partial one.

As noted in section 3.3, SME is suitable for more than HMS and, by the same token, can also be applied to more than legacy projects. Sometimes there are technological reasons that impede the testability of a class or method. For example, some frameworks force the introduction of a parameterless constructor to be able to create an instance of a class for the purposes of serialization. Such frameworks may introduce mutable state where none would be present otherwise and SME can then be used to help keep the unit tests simple.

It would seem that any software project with insufficient testing coverage is fair game, although as the following sub-sections discuss, there are conditions under which SME could deliver greater value than other techniques.

3.4.1 Identifying Stateless Methods

SME will work best if the programming language or toolkit used has a concept of a *stateless* method. This would ensure that each method marked as a stateless method

Listing 3.35: C# showing a proposed `Stateless` attribute with a sample of its use

```
1  [AttributeUsage(AttributeTargets.Method,
2    AllowMultiple = false, Inherited = true)]
3  public class StatelessAttribute : Attribute
4  {
5  }
6
7  public static class DateFormatter
8  {
9    const string ExtendedIso8601Utc =
10     "{0:0000}-{1:00}-{2:00}T{3:00}:{4:00}:{5:00}Z";
11
12   [Stateless]
13   public static string FormatIso8601Utc(
14     int year, int month, int day,
15     int hour, int minute, int second)
16   {
17     var result = String.Format(
18       ExtendedIso8601Utc,
19       year, month, day, hour, minute, second);
20     return result;
21   }
22 }
```

only reads from its input parameters, only writes to its return values, does not read from or write to any instance or static fields (reading from constants is acceptable, as long as they represent immutable instances), does not interact with the environment (such as reading a file, sending data across a network or updating a database) and only calls methods similarly identified as stateless.

Such a concept need not be implemented at the language level and could instead be implemented by a static code analysis (SCA) tool with the help of a designated decoration. For example, this decoration could be implemented as an attribute in .NET or an annotation in Java. Listing 3.35 shows a C# implementation of the proposed `Stateless` attribute (lines 1-5) and a sample of its use (line 12). By placing the line `[Stateless]` above the `FormatIso8601Utc()` method definition, the C# compiler will include the reference to `StatelessAttribute` in the byte code such that its presence on the method can be queried by SCA tools.

Listing 3.36: C# showing a proposed `Immutable` attribute with a sample of its use

```
1  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
2    AllowMultiple = false, Inherited = true)]
3  public class ImmutableAttribute : Attribute
4  {
5  }
6
7  [Immutable]
8  public class EarthLocation
9  {
10   private readonly double _lat, _lon, _alt;
11   public EarthLocation(double latitude,
12     double longitude, double altitude)
13   {
14     _lat = latitude;
15     _lon = longitude;
16     _alt = altitude;
17   }
18   public double Latitude { get { return _lat; } }
19   public double Longitude { get { return _lon; } }
20   public double Altitude { get { return _alt; } }
21 }
```

3.4.2 Identifying Immutable Instances

A class which contains no mutable state is said to be *immutable.* Any state an immutable class has can only be set during construction—through the parameters of constructors or factory methods—and all its instance methods either return that state directly or perform computations with it (possibly including the method's parameters), in a manner very much like stateless methods. Immutable classes can make it easier to implement SME because there exists, by definition, no sequence of method calls on their instances that can modify their state. Instances of immutable classes therefore make excellent inputs to stateless methods. As discussed in section 3.2.5, immutable classes are considered "simple" and can therefore provide stateless methods with convenient groupings of related input parameters, instead of specifying them separately. The use of immutable classes can thus provide additional readability in unit tests at the expense of a constructor call.

Similar to the `Stateless` decoration suggested in sub-section 3.4.1, a decoration could be defined to identify classes that are immutable and thus safe to use as constants or input parameters. Listing 3.36 shows a C# implementation of the proposed `Immutable` attribute (lines 1-5) and a sample of its use (line 7). By placing the line `[Immutable]` above the `EarthLocation` class definition, the C# compiler will include the reference to the `ImmutableAttribute` in the byte code such that its presence on the class can be queried by SCA tools. This idea was further developed by Bazuzi and Pilch-Bisson [67].

These proposed `Stateless` and `Immutable` decorations have the additional benefit that they could also be exploited by an optimizing compiler to inline the stateless methods (and apply other such performance improvements) in release mode, due to the new guarantees of immutability and invariance they provide by being enforced by the compiler and/or SCA.

3.4.3 Controlling Visibility

The SME technique will work best if the programming language has features to control the visibility of methods to ensure that the newly-introduced stateless methods do not pollute the public interface of the affected classes. C# has the `internal` access modifier [8], which, when combined with the `InternalsVisibleTo` attribute [9], can expose the stateless methods to a unit test project, but not to consumers of the API, which, in effect, hides the results of SME from end-users. Java has a similar concept with its packages [37]. Indeed, unless the `public` or `private` access modifiers are used, a method will be visible from within the package it was defined [37], which means unit tests in the same package will have access to the newly-extracted stateless methods, but consumers of the API will not.

An alternative to visibility controls or access modifiers—especially for programming languages without such features—is to place the extracted methods in a separate

class (possibly named after the original class, but suffixed with `Stateless`) or even not attached to any class, provided the language makes this possible. It may even be the case that, upon further consideration, a newly-extracted method should really belong in the class of one of its parameters. For example, the `CrossProduct()` and `DotProduct()` methods extracted from the `Plane` class in listing 3.13 would probably be best located in the `Vector` class.

3.5 Public Interface Comparer Tool

The Public Interface Comparer[4] is a tool written specifically to measure the *Public Interface Changes* metric. This section explains how it works and gives a few examples of its use.

3.5.1 How it Works

The tool is a command-line program that accepts three arguments: the path to the `baselineFile`, the path to the `challengerFile` and the path to the `reportFile`. For each of the assemblies (byte code binary files) pointed to by `baselineFile` and `challengerFile`, the `PublicInterfaceScanner` class uses the reflection features of .NET to find all the publicly-visible types (classes and structs) and, for each of those, finds all their publicly-visible members (fields, properties, events and methods). Each such member is converted into a canonical string representation of its definition and added to a list. When all members of an assembly have been added to the list, the list is sorted alphabetically and returned to the main program.

Public visibility is defined as the application programming interface (API) intentionally exposed by the authors of the library for consumption by third-party programs. In .NET and Java, the mechanism for doing so is to decorate a member

[4]The source code for the Public Interface Comparer Tool can be found in the public Subversion repository at `http://testoriented.googlecode.com/svn/pic/trunk/`

Listing 3.37: C# showing the `BlockModifier` class from the `Textile` project after the `visibility` strategy was applied

```
 1  namespace Textile
 2  {
 3    public class BlockModifier
 4    {
 5      protected internal BlockModifier()
 6      {
 7      }
 8
 9      public virtual string ModifyLine(string line)
10      {
11        return line;
12      }
13
14      public virtual string Conclude(string line)
15      {
16        return line;
17      }
18    }
19  }
```

with the visibility keywords of `public` or `protected`.

When the program has received both lists of publicly-visible members, the creation of the difference report—which will be written to the file pointed to by `reportFile`—is a matter of finding differences between the two lists. This is done in a single pass in a manner similar to merging.

3.5.2 Examples

An example use of the Public Interface Comparer tool is presented here, where listing 3.37 shows one version of the `BlockModifier` class while listing 3.38 shows another with a few differences in visibilities.

Listing 3.38: C# showing the `BlockModifier` class from the `Textile` project after some illustrative visibility changes were applied

```
 1  namespace Textile
 2  {
 3    public class BlockModifier
 4    {
 5      internal BlockModifier()
 6      {
 7      }
 8
 9      public virtual string ModifyLine(string line)
10      {
11        return line;
12      }
13
14      internal virtual string Conclude(string line)
15      {
16        return line;
17      }
18    }
19  }
```

Listing 3.39: Sample report of the public interface differences between the code in listing 3.37 and that of listing 3.38

```
1  Textile.BlockModifier protected Void .ctor()
2  Textile.BlockModifier public System.String Conclude(System.String)
```

As we can see in listing 3.39, the constructor (line 1) and the `Conclude()` method (line 2) were identified as public interface changes between the two versions of the `BlockModifier` class.

3.6 Decisions Made

The candidate block identification step in section 3.2.2 requires that blocks contain only stateless statements, because statelessness is a transitive property and the presence of any operation that causes side-effects in a method will disqualify all methods

that call such a method from claiming statelessness. A stateless method must therefore only call stateless methods, otherwise the effort is defeated.

The candidate block identification also requires selecting computations that use 1 to 6 inputs, because a method that accepts no parameters would have to return a constant (otherwise it could not be considered stateless) while 6 was selected as a suggested maximum number of inputs, as a matter of experience, since too many inputs might make it difficult to write tests, defeating the goal of this thesis. This is in line with Binder's thinking [23] and is supported by Meszaros' "Minimal Fixture" strategy:

> "We use the smallest and simplest fixture possible for each test. (...) Tests are much easier to understand if the fixture is small and simple." [61]

The last criterion in block candidacy is selecting computations that produce 1 to 3 outputs, because a method that does not return any results likely causes side-effects—thus can not be considered stateless—and side-effects are not trivial to observe during unit testing. The suggested maximum number of outputs was also selected from experience and is again intended to keep unit tests simple, mostly because very few programming languages support returning more than one value at once. Returning multiple values at once is often emulated by using a container for the values, either through a specialized type (such as `Vector` used in listings 3.12 and 3.13 or `EarthLocation` in listing 3.36) or an anonymous one, such as a tuple [13, 3].

The intent of incidental operation re-ordering is to maximize the number of operations in a candidate block while minimizing the number of inputs and outputs. This is done to avoid extracting stateless methods that perform too few operations, which would cause unit testing to become a high-work and low-return activity. Although this incidental operation re-ordering step is automatable through data dependency

analysis and program slicing, a programmer will likely be the best judge of which operations are considered related and if these operations collectively have a recognized or standard meaning.

For example, the expression `new Vector(a.Y * b.Z-a.Z * b.Y, a.Z * b.X-a.X * b.Z, a.X * b.Y-a.Y * b.X)` is better identified as a `CrossProduct()` method that accepts two `Vector` instances `a` and `b`.

Input simplification's intent is to minimize the amount of knowledge the stateless method will need to perform its processing or computation. It does this by optimizing the inputs and outputs such that the upcoming unit tests (exercising the to-be-extracted stateless method) contain a minimal number of statements, while balancing the readability and maintainability of both the stateless method and the unit tests. Another reason for favouring "simple" types—usually the language's built-in base types, such as strings, integers and floating-point numbers—is that they are immutable and/or passed by value, meaning their use from within a stateless method will not result in side-effects. Although any type with instance fields that are set at construction time, and are read-only thereafter, is acceptable as an input parameter, readability and maintainability should be considered if less than the majority of its instance fields are used by the method under test. For example, the use of a `DateTime` struct as a parameter when only its `Year` and `Month` properties are needed might suggest that two integer parameters for `year` and `month` would make for a better choice.

In the absence of programming language features to definitely identify and enforce a method as stateless, the `static` keyword is used to at least make it more difficult to access or manipulate HMS. It should be noted that it is not always possible to restrict the use of HMS from a static method. Making a stateless method static also makes it callable without first having to create an instance of a class. Unfortunately, the `static` keyword does not prevent manipulation of static fields, nor does it enforce

that all method calls therein are also stateless.

The stateless methods' visibility of `package` or `internal` was selected to make the use of SME invisible to end-users. Indeed, having the test fixture in the same package as the class under test is the convention for JUnit [82]. Using MSTest to test methods marked as `internal` in another project is not only documented but also assisted by Visual Studio [11].

Sub-section 3.3.1 argues for a threshold of 50% on the size of a unit test's *arrange* phase, which should trigger an investigation into the content of said arrange phase when exceeded, to see if simplification through SME or other means is possible. The intent here is to minimize the size of unit tests to encourage their use. Tests with long arrange phases are still valid but beyond a certain size might be considered *integration* tests more than *unit* tests.

The set of transformations performed during SME were designed to be low-impact in order to encourage their use and reduce the risk of introducing regressions in the very code to which unit tests added. Indeed, "altering internal structure without changing external behaviour" is the very definition of refactoring [31] and, as such, care should be taken to *only* perform the SME steps during the process and to resist the temptation to fix defects or add features at the same time. Once SME has been performed, unit tests are written against the new method to establish a functionality baseline and both sets of changes are committed to source control. Only then will it be safe to add unit tests to expose any potential defects or missing features encountered during SME and then fix the defects or add the features in the presence of the safety net added in the previous step. Binder calls this approach *Incremental Testing* [20].

3.7 Summary

This chapter introduced our approach, describing the steps of the SME technique, along with samples of the associated source code transformations. The suitability of SME for more than HMS was discussed, as well as the conditions under which SME would work best, concluding with a description of the PIC tool and the motivations behind the decisions made in the SME.

Chapter 4

Results and Validation

4.1 Introduction

This chapter describes experiments with which SME's efficiency at improving testability can be measured and compared against other strategies. The results of the experiments are summarized and also presented in great detail, concluding with the relation of the results with the metrics of the goal.

4.2 Evaluation Design

4.2.1 Overview

The goal of stateless method extraction is to improve the testability of business logic in classes that contain hidden mutable state.

Strategies

SME will be compared to three other strategies, for a total of four:

1. Do nothing
2. Extract stateless methods
3. Increase state visibility
4. Increase state and method visibility

Projects

The evaluation makes use of the source code from the following C# open-source projects, selected by the author based on his familiarity with them:

1. Textile.NET, changeset 26030 [25], henceforth referred to as `Textile`
2. KeePass, version 2.10 [71], henceforth referred to as `KeePassLib`
3. Atomic CMS, version 2.0 [72], henceforth referred to as `AtomicCms`

Metrics

The goal's validation metrics (see section 1.4) are repeated here and are evaluated for each of the competing strategies on each of the open-source projects:

1. Complexity of the unit tests.
2. Number of changes to the class under test's public interface.
3. Percentage of branches covered using concolic testing.

The metrics are now explained in further detail, followed by the strategies, the selected methods from the open-source projects and finally how the evaluation was conducted.

4.2.2 Metrics

Complexity of the Unit Tests

> "If the effort to produce and run tests is high, however, less testing will be done." [19]
>
> "The fewer tests you write, the less productive you are and the less stable your code becomes." [36]

There appears to be consensus regarding the minimization of unit tests' sizes [30, 64, 61, 76]. Keeping unit tests small is generally with the intention of reducing the friction related to writing said unit tests (as noted above by Binder) as well as increasing the readability and maintainability of the source code [38].

The tool used for evaluating the complexity of the manually-written unit tests is the *Visual Studio Code Metrics PowerTool 10.0* [14] (CMPT). It can compute, among others, a *Maintainability Index* (MI) that is derived and adapted from the Carnegie Mellon University's Software Engineering Institute metric of the same name [80]. In short, the tool combines a few low-level metrics, obtained by scanning the byte code, and scales the result between 0 and 100. Higher values are better. The modifications Microsoft made are as follows:

> "The metric originally was calculated as follows: $MaintainabilityIndex = 171 - 5.2 \times ln(HalsteadVolume) - 0.23 \times (CyclomaticComplexity) - 16.2 \times ln(LinesOfCode)$
>
> This meant that it ranged from 171 to an unbounded negative number. We noticed that as code tended toward 0 it was clearly hard to maintain code and the difference between code at 0 and some negative value was not useful. (...) As a result of the decreasing usefulness of the negative numbers and a desire to keep the metric as clear as possible we decided to treat all 0 or less indexes as 0 and then re-base the 171 or less range to be from 0 to 100. Thus, the formula we use is:
>
> $MaintainabilityIndex = MAX(0, (171 - 5.2 \times ln(HalsteadVolume) - 0.23 \times (CyclomaticComplexity) - 16.2 \times ln(LinesOfCode)) \times 100/171)$" [62]

The CMPT is also used to evaluate the IUT of each strategy, to see if the transformations performed as part the respective strategies adversely affected the maintainability of the code being tested. Because this metric is not part of the goal, its

results will not be included in the unit test complexity metric and are recorded more as a "canary in a coal mine" to detect undesirable transformations.

Number of Changes to the Class Under Test's Public Interface

It could be undesirable to make changes to the CUT's public interface simply to increase its testability. Reasons for not wanting any such changes include:

1. The class is part of an API with pre-existing releases. Changes to the API could break third-party applications consuming it.

2. The class must implement a specific interface; a technological variation of the previous reason.

3. Exposing instance state as directly mutable might allow callers to violate some constraints or invariants.

This second metric counts how often each strategy causes a change to the public interface of the class under test, compared to the original version. This metric will be referred to as the *Public Interface Changes* (PIC) metric. Lower values are better, with zero being ideal. The tool used is the *Public Interface Comparer* that was described in section 3.5.

Percentage of Branches Covered Using Concolic Testing

The Pex tool [78] is a concolic, constraint solver-based test generator that can be used to analyze a specified project and generate unit tests to exercise as much of that project's source code as it can. Pex's ability to exercise the CUT's methods in an automated and unattended fashion provides a very good objective measure of testability. This ability is measured by the code coverage reported when executing the generated unit tests with a code coverage tool. Higher values are better. Version

0.91.50418.0 of Pex is used, along with NUnit [68] version 2.4.8 to run the unit tests and NCover [83] version 1.5.8 to measure the code coverage.

4.2.3 Strategies

Do nothing

No transformations are to be performed to the IUT, the source code will be used as-is to establish a baseline. Henceforth referred to as `base`.

Increase State Visibility

All the fields (both instance and static fields) in all the classes had their visibility increased through a manual transformation that converted fields with a visibility (explicit or otherwise) of `private` to `internal` and `protected` to `protected internal`. Henceforth referred to as `visibility-state`.

Increase State and Method Visibility

Similar to the previous strategy, all the fields and all the methods (both instance and static) in all the classes had their visibility increased through a manual transformation that converted fields and methods with a visibility (explicit or otherwise) of `private` to `internal` and `protected` to `protected internal`. This strategy helps to understand the effectiveness of the Pex-related code coverage metric. Henceforth referred to as `visibility`.

Extract Stateless Methods

This is the very approach proposed by this thesis; stateless methods are extracted—through a manual process—where good candidate blocks are identified. Henceforth referred to as `manual`.

Project Namespace Class Method
Textile
Textile.Blocks
CodeBlockModifier
string ModifyLine(string)
string Conclude(string)
string CodeFormatMatchEvaluator(Match)
Textile.States
FootNoteFormatterState
void Enter()
void OnContextAcquired()
KeePassLib
KeePassLib.Cryptography.PasswordGenerator
PatternBasedGenerator
string ExpandPattern(string)
KeePassLib.Security
XorredBuffer
byte[] ChangeKey(byte[])
AtomicCms
AtomicCms.Web.Controllers
AdminMenuItemController
void FormatResultMessage(IMenuItem, int?)

Table 4.1: Methods targeted for manually-written unit tests

4.2.4 Selected Methods for Maintainability Metric

Table 4.1 lists the methods that have been selected because they were identified as particularly suitable for SME as per the criteria in section 3.3, such as interacting with HMS or computing several results simultaneously. This selection forms a representative sample of the testing difficulties associated with HMS and care was taken to exclude methods that were deemed to already be trivially testable or too similar to those already selected. Unit tests were written to target these methods for each strategy, exploiting the respective transformations to the IUT to simplify the tests where possible. It is for the manually-written tests of these methods that the "complexity

of the unit tests" metric will be computed.

4.2.5 Preparing the Projects for the Evaluation

The source code to the projects was imported into a public source control repository[1]. Each project gets its own folder and the imported version (which serves as a baseline) is placed in a sub-folder called `base`. Some slight modifications were made to the source code to simplify the evaluation, detailed as follows:

Consolidate Projects

If a project consisted of many sub-projects, the source files for those sub-projects were combined into a single project, to simplify the configuration of Pex.

Remove Provided Tests

Any existing automated tests were removed, otherwise Pex would be able to trivially reach areas of the CUTs through their tests, defeating the purpose of that part of the evaluation.

Add Two Empty Test Projects

To help Pex generate unit tests, a corresponding empty test project was created for each project (suffixed with `.Tests`), as was another empty test project created for the complexity metric (suffixed with `.ManualTests`). Each empty test project was configured to reference its associated project. The test projects are ready to have test classes added to them.

[1]The files used for the suitability evaluation of this thesis are available inside the Subversion repository at `https://testoriented.googlecode.com/svn/suitability/trunk`

Expose Internal Members to Pex

Each project was configured to make any methods marked as `internal` visible to both the empty test projects and to Pex itself, using the `InternalsVisibleTo` attribute.

Disable Functionality That Interferes With the Evaluation

The `KeePassLib` project contains code to show a modal message box dialog that must be dismissed by an end-user. This is undesirable for an unattended evaluation and therefore the body of the `SafeShowMessageBox()` method in the `MessageService` class was removed, as was that of the `SafeShowMessageBoxInternal()` method in the `MessageService` class. Also, to prevent assertions in the projects from interrupting the automated evaluation in the same way, the compilation options were modified to use the *Release* configuration with optimizations disabled and full debug symbols enabled[2].

Create a Copy for Each Strategy

Now that we have a baseline for each project, we can create a copy of the `base` folder for each strategy we wish to compare against. As such, the following folders were created: `manual` (manual application of SME), `visibility-state` (increasing the visibility of state members, such as fields) and `visibility` (increasing the visibility of both fields and methods). The corresponding transformations were then performed in their respective folders, isolated from one another and from the baseline.

[2]Assertions will only display a dialog if the DEBUG constant is defined and it is not defined when using the Release configuration. Disabling optimizations and enabling debug symbols is done to help keep the static code analysis as representative to the original source code as possible.

4.2.6 How it Works

The evaluation is conducted automatically using a small program written with NAnt [73]. The major steps are as follows:

Create a Working Copy

Pex modifies the test project file as part of its test generation process. To make the evaluation repeatable, a copy is made, at the start of the run, of each strategy folder of each project, erasing the previous run's folders as necessary. Pex can then operate in each working copy without chance of interference from the artifacts of any previous runs.

Compile

All the analysis tools (CMPT, Public Interface Comparer and Pex) work by analyzing the byte code of the projects (as opposed to their source code) and thus the next step is to compile the projects.

Run Manual Tests

As a sanity check, the tests written to be evaluated for their complexity are run to make sure they all pass.

Determine Public Interface Differences

The PIC metric is computed on each project's IUT and the report is stored in the project's working folder as the file `PublicInterfaceDifferences.txt`. Each line of the report represents a public interface difference between the base code and the code of a strategy and therefore the number of public interface differences is the number of lines in the report. By definition, the report for the `base` strategy will

always contain 0 lines.

Evaluate Maintainability of IUT and Manual Tests

The CMPT is launched to compute the MI metric for both the IUT and the manually-written unit tests (henceforth known as MT). This allows an objective measure of the maintainability of the IUT in the face of testability transformations as well as a measure of maintainability of the corresponding MT. The CMPT emits the reports respectively as `CodeMetrics.Iut.xml` and `CodeMetrics.Mt.xml` in each working folder. A quick post-processing step converts the absolute paths in the reports into relative paths to simplify comparisons, producing the files `CodeMetrics.Iut.xml.filtered` and `CodeMetrics.Mt.xml.fitered`.

Invoke Pex Wizard

Pex first needs to run in "Wizard" mode, which inspects the IUT and generates test helper methods called "parameterized unit tests". Unfortunately, there are some defects in Pex that garble the test project files and must then be repaired. This is done using the Extensible Stylesheet Language (XSL) transformation file `FixPexWizard.xsl` located in the `Tools` folder. Lastly, the projects are recompiled so that Pex may perform the next step by inspecting the new byte code.

Invoke Pex Generator

Pex is now run in "Generate" mode, which produces the actual unit tests by using the helpers it generated in the previous step. Again, some Pex defects call for the repair of the affected test project files, this time using the `FixPexGenerator.xsl` XSL transformation file. The projects are recompiled one last time in preparation for running the generated tests.

Run Generated Tests With Coverage

The console NUnit runner executes all the unit tests that Pex generated while NCover monitors the execution to collect code coverage data for the IUT. It is not important whether the tests pass, fail or end in error—Pex by its very nature will generate tests that throw or expect to throw `ArgumentException` instances—it is only important to record code coverage. The NCover tool emits a coverage report in each project's working folder as the file `coverage.xml`.

Record Results

Results are collected from the various reports and submitted to an online spreadsheet[3], an explanation and a copy of which can be found in Appendix D.

4.3 Representative SME Application

The following listings show the first of the 8 methods targeted for manually-written unit tests—listed in table 4.1—before the application of SME (listing 4.1) and after the application of SME (listing 4.2), as well as the manually-written unit test targeting the original method (listing 4.3) and the manually-written unit test targeting the extracted stateless method (listing 4.4).

Similar quartets of listings for the remaining 7 methods targeted for manually-written unit tests can be found in Appendix B while a list of all the methods transformed in all 3 open-source projects can be found in Appendix C.

[3]The spreadsheet can be seen at `https://docs.google.com/spreadsheet/ccc?key=0Ag6eugnyWA09dG9zQnFtTlQ0Z1dGUjI5TUlFMnpFWWc`

Listing 4.1: C# showing the `base` version of the `ModifyLine()` method of the `CodeBlockModifier` class of the `Textile` project.

```
 1  public override string ModifyLine(string line)
 2  {
 3      // Replace "@...@" zones with "<code>" tags.
 4      MatchEvaluator me = new MatchEvaluator(CodeFormatMatchEvaluator);
 5      line = Regex.Replace(line,
 6          @"(?<before>^|([\s\([{]))" +    // before
 7           "@" +
 8          @"(\|(?<lang>\w+)\|)?" +        // lang
 9           "(?<code>[^@]+)" +              // code
10           "@" +
11          @"(?<after>$|([\]}])|(?=" + Globals.PunctuationPattern + @"{1,2}|\s|$))",  // after
12        me);
13      // Encode the contents of the "<code>" tags so that we don't
14      // generate formatting out of it.
15      line = NoTextileEncoder.EncodeNoTextileZones(line,
16                          @"(?<=(^|\s)<code(" + Globals.HtmlAttributesPattern + @")>)",
17                          @"(?=</code>)");
18      return line;
19  }
```

Listing 4.2: C# showing the `manual` version of the `ModifyLine()` method, after SME plus a few more "introduce variable" refactorings.

```
 1  private const string Pattern =
 2      @"(?<before>^|([\s\([{]))" + // before
 3       "@" +
 4      @"(\|(?<lang>\w+)\|)?" +    // lang
 5       "(?<code>[^@]+)" +        // code
 6       "@" +
 7      @"(?<after>$|([\]}])|(?=" + Globals.PunctuationPattern + @"{1,2}|\s|$))";  // after
 8  internal static readonly Regex CodeBlockRegex = new Regex(Pattern);
 9
10  public override string ModifyLine(string line)
11  {
12      return InnerModifyLine(line);
13  }
14
15  internal static string InnerModifyLine(string line)
16  {
17      // Replace "@...@" zones with "<code>" tags.
18      MatchEvaluator me = new MatchEvaluator(CodeFormatMatchEvaluator);
19      line = CodeBlockRegex.Replace(line, me);
20
21      // Encode the contents of the "<code>" tags so that we don't
22      // generate formatting out of it.
23      line = NoTextileEncoder.EncodeNoTextileZones(line,
24                              @"(?<=(^|\s)<code(" + Globals.HtmlAttributesPattern + @")>)",
25                              @"(?=</code>)");
26      return line;
27  }
```

Listing 4.3: C# showing the test for the `base` version of the `ModifyLine()` method.

```
1  [Test]
2  public void ModifyLine ()
3  {
4      var cbm = new CodeBlockModifier();
5      var actual = cbm.ModifyLine ("(@|ruby|r.to_html@)");
6      Assert.AreEqual (@"(<code language=""ruby"">r.to_html</code>)", actual);
7  }
```

Listing 4.4: C# showing the test for the `manual` version of the `ModifyLine()` method.

```
1  [Test]
2  public void InnerModifyLine ()
3  {
4      var actual = CodeBlockModifier.InnerModifyLine ("(@|ruby|r.to_html@)");
5      Assert.AreEqual (@"(<code language=""ruby"">r.to_html</code>)", actual);
6  }
```

4.4 Results

Table 4.2 summarizes the results of the metrics, which have been averaged across the participating open-source projects. The table is assembled from the results of running the suitability evaluation against the source code as of revision 424 in the public Subversion repository. Detailed results are presented in the following section, although it is clear that, in general, SME performed no worse and often better than the other strategies.

	Metrics			
Strategies	MI IUT[a]	MI MT[b]	PIC[c]	VSP/TSP[d]
`base`	83.67	77.33	0[e]	52.92%
`visibility-state`	83.67	76.67	0	53.04%
`visibility`	83.67	77.00	0	56.35%
`manual`	84.00	85.33	0	57.92%
Difference between `manual` and `base`	+0.33	+8.00	0	+5.00%

Table 4.2: Summary results of the metrics

[a]Maintainability Index of the Implementation Under Test.

[b]Maintainability Index of the Manually-written unit Tests. See Complexity of the Unit Tests in section 4.2.2.

[c]Public Interface Changes. See Number of Changes to the Class Under Test's Public Interface in section 4.2.2.

[d]Visited Sequence Points divided by Total Sequence Points. See Percentage of Branches Covered Using Concolic Testing in section 4.2.2.

[e]This is the reference point other strategies use to measure public interface changes.

4.5 Detailed Results

The following sub-sections illustrate the results of the individual metrics described in sub-section 4.2.2 and discuss some observations made while running the suitability evaluation.

4.5.1 Complexity of the Unit Tests

Table 4.3 lists the per-project overall results of running the CMPT tool on the manually-written unit tests to obtain their maintainability index.

	Strategies				
Projects	base	visibility -state	visibility	manual	Difference[a]
Textile	78.00	76.00	76.00	85.00	+7.00
KeePassLib	70.00	70.00	80.00	81.00	+11.00
AtomicCms	84.00	84.00	75.00	90.00	+6.00

Table 4.3: Overall Maintainability Index of the unit tests in the ManualTests project per evaluation project, per strategy

[a]Difference between manual and base

In the case of the Textile project, the slightly lower MI scores for the visibility and visibility-state strategies are owed to the tests written for FootNoteFormatterState and the fact that the increase in visibility allowed for *two* slightly more precise tests to be written, instead of the *one* general and indirect test possible in base. Two tests mean more lines of code and therefore a lower overall MI score, but this limitation of the CMPT can be avoided by comparing the MI scores on a per-method basis, as shown in table 4.4. There we see that every test method written with a strategy was at least equal to and most of the time better than base. We also see that the manual strategy fared much better because it was able to test the most precise parts of the processing involved in the Enter() and OnContextAcquired() methods in just two lines per test and with less class coupling.

In the case of the KeePassLib project, the transformations afforded to the visibility-state strategy did not help simplify the unit tests by any significant amount, meaning the unit testing difficulties in these MUTs were not surmountable

Projects Test classes Test methods	Strategies base	visibility -state	visib- ility	manual	Difference[a]
Textile					
CodeBlocksModifierTest					
ModifyLine	78.00	78.00	78.00	84.00	+6.00
Conclude	78.00	78.00	78.00	84.00	+6.00
CodeFormatMatchEvaluator	78.00	78.00	78.00	82.00	+4.00
FootNoteFormatterStateTest					
Enter	65.00[b]	65.00	65.00	83.00	+18.00
OnContextAcquired		65.00	69.00	84.00	+19.00
KeePassLib					
PatternBasedGeneratorTest					
ExpandPattern	63.00	63.00	84.00	84.00	+21.00
XorredBufferTest					
ChangeKey	59.00	59.00	59.00	61.00	+2.00
AtomicCms					
AdminMenuItemControllerTest					
FormatResultMessage_Created	64.00	64.00	70.00	84.00	+20.00
FormatResultMessage_Updated	64.00	64.00	70.00	82.00	+18.00

Table 4.4: Detailed breakdown of Maintainability Index scores per method, per strategy

[a]Difference between manual and base

[b]It is impossible to directly test the Enter() and OnContextAcquired() methods so a single, indirect test was written.

by simply increasing the visibility of the instance state. Indeed, the `ExpandPattern()` method is simply not visible to the unit tests, whereas the `ChangeKey()` method operates on state that is initialized from the constructor and from the method parameter, so the results are already easy enough to verify. As a result, the transformations made in the `visibility` strategy were very similar to those made in the `manual` strategy and thus helped simplify the unit tests similarly. SME, in this case, was still useful for the input parameter simplification and for making the test inputs explicit, which explains the slight advantage for the `ChangeKey()` method.

In the case of the `AtomicCms` project, the unit testing of the `FormatResultMessage()` method provided a great example of an unwieldy arrange phase. This is because the constructor of the `AdminMenuItemController` class—in the `base` version—requires two interfaces to be implemented (or mocked), on top of having to indirectly invoke the functionality to be tested (because `FormatResultMessage()` is a private instance method). The difference these factors make can be seen by comparing listing 4.5 and listing 4.6: notice that the `arrange` phase has been completely eliminated in the `manual` version. The need for the `arrange` phase disappeared in the `manual` version because creating an instance of the method's class is no longer required. The method under test is directly callable, all its inputs are provided as simple parameters and the output is the method's return value. Contrast this to listing 4.5, where an instance is created (lines 5-8 and 26-29), the method under test is indirectly invoked (lines 13 and 34), its parameters need to be constructed (lines 9-10 and 30-31) and the output is derived from instance state (lines 16 and 37).

The `visibility-state` strategy was again ineffective in the `AtomicCms` project because HMS was not the biggest unit testing obstacle. The `visibility` strategy fared better because its methods under test could be directly called, although an instance of the CUT still needed to be created and the MUT had not benefited from input simplification like they did with the `manual` strategy.

Listing 4.5: C# showing two AtomicCms unit tests in the base version

```
1  [Test]
2  public void FormatResultMessage_Created()
3  {
4    // arrange
5    var factory = new TestServiceFactory {
6      MenuService = new MenuServiceMole { SaveMenuItem = i => { } }
7    };
8    var iut = new AdminMenuItemController(factory);
9    var item = new MenuItem { Id = 42 };
10   var formCollection = new FormCollection();
11
12   // act - this eventually calls FormatResultMessage
13   iut.EditMenuItem(null, item, formCollection);
14
15   // assert
16   var actual = iut.TempData["SaveResult"];
17   Assert.AreEqual(
18     "Items was successfully created with Id = 42",
19     actual);
20 }
21
22 [Test]
23 public void FormatResultMessage_Updated()
24 {
25   // arrange
26   var factory = new TestServiceFactory {
27     MenuService = new MenuServiceMole { SaveMenuItem = i => { } }
28   };
29   var iut = new AdminMenuItemController(factory);
30   var item = new MenuItem { Id = 42 };
31   var formCollection = new FormCollection();
32
33   // act - this eventually calls FormatResultMessage
34   iut.EditMenuItem(42, item, formCollection);
35
36   // assert
37   var actual = iut.TempData["SaveResult"];
38   Assert.AreEqual("Items was successfully updated", actual);
39 }
```

Listing 4.6: C# showing the same two `AtomicCms` unit tests in the `manual` version

```
 1  [Test]
 2  public void InnerFormatResultMessage_Created()
 3  {
 4    // act
 5    var actual = AdminMenuItemController.
 6        InnerFormatResultMessage(null, 42);
 7
 8    // assert
 9    Assert.AreEqual(
10      "Items was successfully created with Id = 42",
11      actual);
12  }
13
14  [Test]
15  public void InnerFormatResultMessage_Updated()
16  {
17    // act
18    var actual = AdminMenuItemController.
19        InnerFormatResultMessage(42, 42);
20
21    // assert
22    Assert.AreEqual("Items was successfully updated", actual);
23  }
24    }
25  }
```

4.5.2 Number of Changes to the Public Interface of the Class Under Test

Table 4.5 lists the per-project results of running the Public Interface Comparer tool on the IUTs to obtain their number of public interface changes—or differences—relative to the base source code.

This evaluation was mostly to ensure that none of the strategies would transform the IUT in such a way that could change the public interface exposed by the IUT to third-party programs. It did come in handy during development to correct errors introduced during transformations and the presence of zeros across the results is a testament that all three strategies can be implemented without any changes that would be noticed by consumers of the end-result binaries.

Projects	Strategies			
	base	visibility -state	visibility	manual
Textile	n/a	0	0	0
KeePassLib	n/a	0	0	0
AtomicCms	n/a	0	0	0

Table 4.5: Number of changes to the public interface (relative to base) per project, per strategy

4.5.3 Percentage of Branches Covered Using Concolic Testing

Table 4.6 lists the results of measuring the code coverage percentage achieved by the tests generated by Pex.

Projects	Strategies				
	base	visibility -state	visibility	manual	Difference[a]
Textile	59.79%	60.61%	64.13%	67.02%	+7.23%
KeePassLib	45.90%	46.00%	48.33%	49.18%	+3.28%
AtomicCms	50.24%	50.24%	57.85%	55.23%	+4.99%

Table 4.6: Percentage of branch code coverage achieved by concolic testing per project, per strategy.

[a]Difference between manual and base

We can confidently say that the visibility-state strategy produced equivalent results to doing nothing, while the visibility strategy produced better results and the manual strategy produced the best results, save for the AtomicCms project. This is most likely due to the fact that the AtomicCms project consists almost entirely of data access code, loading entities from a database into an object model in order to display them in a web page, as well as code to go the other way around. The project is therefore very state-centric and contains very little of the kind of "pure computations" (such as deterministic data transformations) that lend themselves to be extracted into stateless methods.

The fact that the `visibility` strategy fared better than the `visibility-state` strategy suggests that the biggest obstacle to Pex's quest for coverage is not hidden mutable state but rather methods it cannot reach. Even so, that may be a necessary condition, but not a sufficient one as the `manual` strategy still outperformed the `visibility` strategy in 2 out of the 3 projects, which suggests that there is merit in what differentiates the `manual` strategy from the `visibility` strategy: statelessness and input simplification.

4.6 Validation

To help meet the goal, the proposed SME testability refactoring was applied on selected parts of three open-source projects and then compared against three other strategies.

Metrics / Projects	Strategies: Do nothing	SME	Difference
MI MT[a] (average)	77.33	85.33	+8.00
`Textile`	78.00	85.00	+7.00
`KeePassLib`	70.00	81.00	+11.00
`AtomicCms`	84.00	90.00	+6.00
PIC[b] (average)	0	0	0
`Textile`	0	0	0
`KeePassLib`	0	0	0
`AtomicCms`	0	0	0
VSP/TSP[c] (average)	52.92%	57.92%	+5.00%
`Textile`	59.79%	67.02%	+7.23%
`KeePassLib`	45.90%	49.18%	+3.28%
`AtomicCms`	50.24%	55.23%	+4.99%

Table 4.7: Comparison of the results between `base` and `manual`, on average and per-project.

[a]Complexity of unit tests: A higher number indicates better maintainability and, consequently, lower complexity.

[b]Public interface changes: A lower number is better.

[c]The percentage of branches covered using concolic testing. A higher number is better.

As can be seen in sections 4.4 and 4.5 (summarized in table 4.7), the application of SME on the three test projects had the following general effects on the metrics, relative to the `base` version:

1. The complexity of the manually-written unit tests went down.
2. Zero changes to the public interface of the class under test were necessary.
3. The percentage of branches covered using concolic testing went up.

Given the three metrics are better than or equal to the baseline, we have been able to reach the thesis goal to improve the testability of business logic in classes that suffer from the "state problem".

4.7 Summary

This chapter introduced the mechanism used to compare SME against three other strategies, the open-source projects with which the experiments and metrics were carried out as well as what tools were used and how. The results showed that SME improved testing and how the goal was validated.

Chapter 5

Conclusion

5.1 Introduction

This chapter describes the work that was performed, discusses the results and concludes with ideas for further research.

5.2 Synopsis

In chapter 1, the "state problem" was introduced as a specific problem in the general field of software testing. Section 1.3 motivated the importance of the "state problem" as a research topic and section 1.4 introduced the goal of this thesis: to improve the testability of business logic in classes that suffer from the state problem—because they contain hidden, mutable state—while minimally affecting functionality, maintainability and the public interface of the code under test. These three conditions were converted into three metrics from which the effectiveness of any proposed approach could be evaluated: complexity of the unit tests, number of changes to the public interface of the class under test and percentage of branches covered using concolic testing. The chapter concluded with the following objectives in section 1.5: the design of an approach that addresses the goal, the development of an automated evaluation mechanism to validate the proposed approach, the development of the PIC tool, the application of the approach (as well as two others, for relative comparison purposes) on select open-source projects and the application of the evaluation on the modified open-source projects.

Related background and work was presented in chapter 2 to explain the problem space, establish context and cover previous attempts at solving similar problems in the fields of test generation and program transformation.

Chapter 3 describes the stateless method extraction approach in section 3.2, with many source code examples presented. Suitability is then discussed in section 3.3, covering cases other than HMS where SME can be used, followed by SME's best conditions for applicability in section 3.4. The PIC tool was described in section 3.5 and the chapter concluded with a discussion of intents and their associated consequences in section 3.6.

In chapter 4, an evaluation mechanism was described in section 4.2 after which the proposed approach, as well as two others, were applied on three open-source projects so that each approach could be compared against each other as well as against the baseline source code. Empirical summary and detailed results were presented in sections 4.4 and 4.5, respectively. Lastly, in section 4.6 the results were cross-referenced with the goal's metrics and were shown to validate our approach through concolic testing code coverage increases between 3 to 7 percent, unit test complexity decreases between 6 and 11 points and no changes to the public interface.

5.3 Bias

The following examples identify ways to mitigate bias and its corresponding risk to our results.

1. Selection of the open-source projects.

   If SME could be automated, it could be applied on a much larger selection of projects—at random or exhaustively—and thus could reduce the bias inherent in the author's selection.

2. Selection of the representative methods in said open-source projects.

   The methods listed in table 4.1 were selected at the author's discretion to be targeted for manually-written unit tests. Writing tests for a random selection of methods or—time-permitting—all methods in all projects would reduce bias, but this could come at the expense of increasing bias in the selection of open-source projects, because the writing of unit tests is a time-consuming process and less projects would have to be selected as a result.

3. Manually-written tests.

   The tests that were written for the complexity evaluation suffer from the author's bias and ability. One way to mitigate this bias—and at the same time reduce the amount of manual labour involved in the evaluation—is to avoid the need for human-generated tests in the first place and to measure the complexity of the tests generated by a computer, such as those by Pex. This may not be ideal, either, as Pex-generated tests may not necessarily be written to be maintained by humans, but it would have the advantage of objectivity.

4. The goal's metrics and the tools used to evaluate them.

   Although the CMPT implements a slight variation of the industry-standard Maintainability Index, said metric is a bit dated (it was originally released in 1997) and has been declared "legacy" by Carnegie Mellon University's Software Engineering Institute. Nevertheless, the notion of *complexity* may vary from maintainer to maintainer and thus the measure of the *complexity of the unit tests* metric is probably best left to the maintainer(s) of each project, even at the expense of the objectivity of a completely-automated evaluation.

   The *Public Interface Comparer* tool was written by the author (see section 3.5) and could contain defects that affect the accuracy of the results. Some further

testing of the tool could reduce its threats to validity.

The choice of Pex—a *concolic testing* tool—as a test generator may not be optimal given the *state problem* was originally reported for *evolutionary testing* tools. The unavailability of an ET tool for the .NET platform lead to the use of Pex as a stand-in. This threat to validity could be reduced by repeating the evaluation in a language or on a platform where an ET tool is available, although new open-source projects would have to be selected and the other parts of the evaluation would have to be ported or replaced with equivalents, such as using JUnit instead of NUnit and Emma instead of NCover.

5. Stochastic behaviour of concolic testing.

Although not as severe as that of metaheuristic search techniques, the Pex tool did present some slight variations in its results due to the use of timeouts in the constraint solver and the nature of running the evaluation on non-dedicated hardware[1]. The use of dedicated hardware could reduce the variability of the results, while an increase in the number of runs would allow the use of averages to smooth out variability. However, given the standard deviation observed after 7 runs was no more than 0.28% and averaged 0.14%, this variability was deemed inconsequential compared to the difference between strategies, which was on the order of 3% to 7%.

[1]Since Pex is configured to be single-threaded and will only saturate one CPU core per run, three independent runs were often scheduled simultaneously on the author's 4-core workstation computer to reduce the total amount of time to wait for the results. It is likely the runs would have been competing for resources amongst themselves and other processes running on the computer.

5.4 Future Work

As was seen in sub-section 4.2.3, there is more than one possible strategy for addressing HMS and there are very likely more strategies, both competing and complementing to SME. It would be interesting to compare SME to a non-trivial HMS-oriented testability refactoring, both head-to-head and working together. Example strategies include Binder's "Built-in Test Driver" and "Private Access Driver" [19], Meszaros' "Test Utility Methods" [61] and introducing more "testing seams" [46].

As discussed in sub-section 5.3, one way to increase confidence in the results would be to repeat the evaluation with more projects and this would be substantially easier to achieve if the SME technique could be applied automatically, by writing a program able to perform SME on arbitrary source code.

Although it may be possible to automate the application of SME on the source code of individual projects, most projects have dependencies for which source code is not always available, including that of the base framework. Some mechanism to manually annotate the API of such opaque dependencies regarding immutability of the classes and statelessness of the methods available would very likely help guide SME, especially for the identification of candidate blocks and simplification of inputs.

5.5 Closing Thoughts

A belief behind this thesis is that a class that is difficult to test because of HMS can be worked around by mechanically transforming its implementation in such a way that simple tests are then trivial to write. While transforming a class in a manner invisible to its consumers for the purposes of increasing its testability probably has the least overall impact, it is possible—even likely—a better design would provide even better testability. Such a design change might be risky—or even impossible—depending on the project's non-functional requirements, life cycle, maturity, business pressures and

other factors. The game then becomes that of trade-offs and risks vs. rewards.

On the other hand and at a higher level, since "testing can only show the presence of defects and not their absence" [28], is an increase in the quantity and/or quality of testing really the best way to address the problem of software quality, given that correctness proofs can be simpler and more powerful [29]? This is probably a question of compatibility: if a software project was already built with a formal approach, a change in requirements (or formal functional specification) is probably best addressed with corresponding changes to both the program and the proof, whereas in a less formal setting, a similar change in requirements is probably best addressed with an equally informal approach.

In a world where "worse is better" [35] and "something is infinitely better than nothing" [26], there are pressures and requirements for incrementally making software better. The next time a software project needs to be made better due to the presence of hidden, mutable state, there is now a strategy that can improve its testability with minimal impact: stateless method extraction.

Appendix A

Acronyms

API Application Programming Interface

CT Concolic Testing

CGT Computer-Generated Tests

CMPT Code Metrics PowerTool

CUT Class Under Test

ET Evolutionary Testing

GA Genetic Algorithms

HGT Human-Generated Tests

HMT Hidden Mutable State

IUT Implementation Under Test

MI Maintainability Index

MST Metaheuristic Search Technique

MT Manually-written unit Tests

MUT Method Under Test

OO Object-Oriented

OOP Object-Oriented Programming

PIC Public Interface Changes

RT Random Testing

SBST Search-Based Software Testing

SCA Static Code Analysis

SE Symbolic Execution

SME Stateless Method Extraction

TSP Total Sequence Points

VSP Visited Sequence Points

XSL Extensible Stylesheet Language

Appendix B

Representative SME Applications

This appendix contains code samples illustrating the application of SME on the methods targeted for manually-written unit tests—listed in table 4.1—as well as the unit tests targeting the original methods and those targeting the extracted stateless methods. Table B.1 provides a guide to the code listings.

Project Namespace Class Method	Listings MUT Before	MUT After	Test Before	Test After
`Textile`				
`Textile.Blocks`				
`CodeBlockModifier`				
`string ModifyLine(string)`	4.1	4.2	4.3	4.4
`string Conclude(string)`	B.1	B.2	B.3	B.4
`string CodeFormatMatchEvaluator(Match)`	B.5	B.6	B.7	B.8
`Textile.States`				
`FootNoteFormatterState`				
`void Enter()`	B.9	B.10	B.11	B.12
`void OnContextAcquired()`	B.13	B.14	B.15	B.16
`KeePassLib`				
`KeePassLib.Cryptography.PasswordGenerator`				
`PatternBasedGenerator`				
`string ExpandPattern(string)`	B.17	B.18	B.19	B.20
`KeePassLib.Security`				
`XorredBuffer`				
`byte[] ChangeKey(byte[])`	B.21	B.22	B.23	B.24
`AtomicCms`				
`AtomicCms.Web.Controllers`				
`AdminMenuItemController`				
`void FormatResultMessage(IMenuItem, int?)`	B.25	B.26	B.27	B.28

Table B.1: Methods targeted for manually-written unit tests and their corresponding listings

Listing B.1: C# showing the `base` version of the `Conclude()` method of the `CodeBlockModifier` class of the `Textile` project.

```
1  public override string Conclude(string line)
2  {
3      // Recode everything except "<" and ">";
4      line = NoTextileEncoder.DecodeNoTextileZones(line,
5                              @"(?<=(^|\s)<code(" + Globals.HtmlAttributesPattern + @")>)",
6                              @"(?=</code>)",
7                              new string[] { "<", ">" });
8      return line;
9  }
```

Listing B.2: C# showing the `manual` version of the `Conclude()` method, after SME.

```
1  public override string Conclude(string line)
2  {
3      return InnerConclude(line);
4  }
5
6  internal static string InnerConclude(string line)
7  {
8      // Recode everything except "<" and ">";
9      line = NoTextileEncoder.DecodeNoTextileZones (line,
10                             @"(?<=(^|\s)<code(" + Globals.HtmlAttributesPattern + @")>)",
11                             @"(?=</code>)",
12                             new string[] { "<", ">" });
13     return line;
14 }
```

Listing B.3: C# showing the test for the `base` version of the `Conclude()` method.

```
1  [Test]
2  public void Conclude ()
3  {
4      const string input = @"<code language=""ruby"">return &#39;3 < 5&#39;</code>";
5      var cbm = new CodeBlockModifier ();
6      var actual = cbm.Conclude (input);
7      Assert.AreEqual (@"<code language=""ruby"">return '3 < 5'</code>", actual);
8  }
```

Listing B.4: C# showing the test for the `manual` version of the `Conclude()` method.

```
1  [Test]
2  public void InnerConclude ()
3  {
4      const string input = @"<code language=""ruby"">return &#39;3 < 5&#39;</code>";
5      var actual = CodeBlockModifier.InnerConclude (input);
6      Assert.AreEqual (@"<code language=""ruby"">return '3 < 5'</code>", actual);
7  }
```

Listing B.5: C# showing the base version of the CodeFormatMatchEvaluator() method of the CodeBlockModifier class of the Textile project.

```
1  static public string CodeFormatMatchEvaluator(Match m)
2  {
3      string res = m.Groups["before"].Value + "<code";
4      if (m.Groups["lang"].Length > 0)
5          res += " language=\"" + m.Groups["lang"].Value + "\"";
6      res += ">" + m.Groups["code"].Value + "</code>" + m.Groups["after"].Value;
7      return res;
8  }
```

Listing B.6: C# showing the manual version of the CodeFormatMatchEvaluator() method, after SME.

```
1  static public string CodeFormatMatchEvaluator(Match m)
2  {
3      return BuildCodeElementString(
4          m.Groups["before"].Value,
5          m.Groups["lang"].Value,
6          m.Groups["code"].Value,
7          m.Groups["after"].Value);
8  }
9
10 internal static string BuildCodeElementString(string before, string lang, string code, string after)
11 {
12     string res = before + "<code";
13     if (lang.Length > 0)
14         res += " language=\"" + lang + "\"";
15     res += ">" + code + "</code>" + after;
16     return res;
17 }
```

Listing B.7: C# showing the test for the `base` version of the `CodeFormatMatchEvaluator()` method.

```
1  [Test]
2  public void CodeFormatMatchEvaluator ()
3  {
4      var m = Regex.Match (
5            "Call the ruby r_tohtml(); method",
6            @"(?<before>Call\sthe\s)" +
7            @"(?<lang>ruby)\s" +
8            @"(?<code>r_tohtml\(\);)" +
9            @"(?<after>\smethod)"
10     );
11     var actual = CodeBlockModifier.CodeFormatMatchEvaluator(m);
12     Assert.AreEqual ("Call the <code language=\"ruby\">r_tohtml();</code> method", actual);
13 }
```

Listing B.8: C# showing the test for the `manual` version of the `CodeFormatMatchEvaluator()` method.

```
1  [Test]
2  public void BuildCodeElementString ()
3  {
4      var actual = CodeBlockModifier.BuildCodeElementString ("(", "ruby", "r.to_html", ")");
5      Assert.AreEqual (@"(<code language=""ruby"">r.to_html</code>)", actual);
6  }
```

Listing B.9: C# showing the `base` version of the `Enter()` method of the `FootNoteFormatterState` class of the `Textile` project.

```
1  public override void Enter()
2  {
3      Formatter.Output.Write(
4          string.Format("<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
5              m_noteID,
6              FormattedStylesAndAlignment(),
7              m_noteID));
8  }
```

Listing B.10: C# showing the `manual` version of the `Enter()` method, after SME.

```
1  public override void Enter()
2  {
3      Formatter.Output.Write(
4          FormatFootNote(m_noteID, FormattedStylesAndAlignment()));
5  }
6
7  internal static string FormatFootNote(int noteId, string formattedStylesAndAlignment)
8  {
9      return string.Format("<p id=\"fn{0}\"{1}><sup>{2}</sup> ",
10         noteId,
11         formattedStylesAndAlignment,
12         noteId);
13 }
```

Listing B.11: C# showing the test for the `base` version of the `Enter()` method.

```
1  [Test]
2  public void EnterAndOnContextAcquired()
3  {
4      // arrange
5      var output = new StringBuilderTextileFormatter ();
6      output.Begin();
7      var fnfs = new FootNoteFormatterState(new TextileFormatter(output));
8      var expression = SimpleBlockFormatterState.PatternBegin + @"fn[0-9]+" + SimpleBlockFormatterState.PatternEnd;
9      var input = "fn1{color:red}. This is the footnote";
10     Match m = Regex.Match(input, expression);
11     fnfs.Consume (input, m);
12
13     // act
14     // do nothing, since Consume() already caused OnContextAcquired() and Enter() to be called
15
16     // assert
17     Assert.AreEqual("<p id=\"fn1\" style=\"color:red;\"><sup>1</sup> ", output.GetFormattedText());
18 }
```

Listing B.12: C# showing the test for the `manual` version of the `Enter()` method.

```
1  [Test]
2  public void FormatFootNote()
3  {
4      // act
5      var actual = FootNoteFormatterState.FormatFootNote(1, " style=\"color:red;\"");
6
7      // assert
8      Assert.AreEqual("<p id=\"fn1\" style=\"color:red;\"><sup>1</sup> ", actual);
9  }
```

Listing B.13: C# showing the `base` version of the `OnContextAcquired()` method of the `FootNoteFormatterState` class of the `Textile` project.

```
1  protected override void OnContextAcquired()
2  {
3      Match m = Regex.Match(Tag, @"^fn(?<id>[0-9]+)");
4      m_noteID = Int32.Parse(m.Groups["id"].Value);
5  }
```

Listing B.14: C# showing the `manual` version of the `OnContextAcquired()` method, after SME.

```
1  protected override void OnContextAcquired()
2  {
3      m_noteID = ParseFootNoteId(Tag);
4  }
5
6  internal static int ParseFootNoteId(string input)
7  {
8      Match m = Regex.Match(input, @"^fn(?<id>[0-9]+)");
9      return Int32.Parse(m.Groups["id"].Value);
10 }
```

Listing B.15: C# showing the test for the `base` version of the `OnContextAcquired()` method.

```
1  [Test]
2  public void EnterAndOnContextAcquired()
3  {
4      // arrange
5      var output = new StringBuilderTextileFormatter ();
6      output.Begin();
7      var fnfs = new FootNoteFormatterState(new TextileFormatter(output));
8      var expression = SimpleBlockFormatterState.PatternBegin + @"fn[0-9]+" + SimpleBlockFormatterState.PatternEnd;
9      var input = "fn1{color:red}. This is the footnote";
10     Match m = Regex.Match(input, expression);
11     fnfs.Consume (input, m);
12
13     // act
14     // do nothing, since Consume() already caused OnContextAcquired() and Enter() to be called
15
16     // assert
17     Assert.AreEqual("<p id=\"fn1\" style=\"color:red;\"><sup>1</sup> ", output.GetFormattedText());
18 }
```

Listing B.16: C# showing the test for the `manual` version of the `OnContextAcquired()` method.

```
1  [Test]
2  public void ParseFootNoteId()
3  {
4      // act
5      var actual = FootNoteFormatterState.ParseFootNoteId("fn42");
6
7      // assert
8      Assert.AreEqual(42, actual);
9  }
```

Listing B.17: C# showing the base version of the ExpandPattern() method of the PatternBasedGenerator class of the KeePassLib project.

```
 1  private static string ExpandPattern(string strPattern)
 2  {
 3    Debug.Assert(strPattern != null); if(strPattern == null) return string.Empty;
 4    string str = strPattern;
 5
 6    while(true)
 7    {
 8      int nOpen = str.IndexOf('{');
 9      int nClose = str.IndexOf('}');
10
11      if((nOpen >= 0) && (nOpen < nClose))
12      {
13        string strCount = str.Substring(nOpen + 1, nClose - nOpen - 1);
14        str = str.Remove(nOpen, nClose - nOpen + 1);
15
16        uint uRepeat;
17        if(StrUtil.TryParseUInt(strCount, out uRepeat) && (nOpen >= 1))
18        {
19          if(uRepeat == 0)
20            str = str.Remove(nOpen - 1, 1);
21          else
22            str = str.Insert(nOpen, new string(str[nOpen - 1], (int)uRepeat - 1));
23        }
24      }
25      else break;
26    }
27
28    return str;
29  }
```

Listing B.18: C# showing the manual version of the ExpandPattern() method, after SME.

```
1  internal static string ExpandPattern(string strPattern)
2  {
3    Debug.Assert(strPattern != null); if(strPattern == null) return string.Empty;
4    string str = strPattern;
5
6    while(true)
7    {
8      int nOpen = str.IndexOf('{');
9      int nClose = str.IndexOf('}');
10
11     if((nOpen >= 0) && (nOpen < nClose))
12     {
13       string strCount = str.Substring(nOpen + 1, nClose - nOpen - 1);
14       str = str.Remove(nOpen, nClose - nOpen + 1);
15
16       uint uRepeat;
17       if(StrUtil.TryParseUInt(strCount, out uRepeat) && (nOpen >= 1))
18       {
19         if(uRepeat == 0)
20           str = str.Remove(nOpen - 1, 1);
21         else
22           str = str.Insert(nOpen, new string(str[nOpen - 1], (int)uRepeat - 1));
23       }
24     }
25     else break;
26   }
27
28   return str;
29 }
```

Listing B.19: C# showing the test for the base version of the ExpandPattern() method.

```
1  [Test]
2  public void ExpandPattern()
3  {
4      // arrange
5      var psOutBuffer = new ProtectedString();
6      var pwProfile = new PwProfile();
7      pwProfile.Pattern = "g{5}";
8      var pbKey = new byte[] { 0x00 };
9      var crsRandomSource = new CryptoRandomStream(CrsAlgorithm.Salsa20, pbKey);
10     var error = PatternBasedGenerator.Generate(psOutBuffer, pwProfile, crsRandomSource);
11
12     // act
13     // nothing to do as ExpandPattern() would have been called by calling Generate()
14
15     // assert
16     Assert.AreEqual(PwgError.Success, error);
17     var actual = psOutBuffer.ReadString();
18     Assert.AreEqual("ggggg", actual);
19 }
```

Listing B.20: C# showing the test for the manual version of the ExpandPattern() method.

```
1  [Test]
2  public void ExpandPattern()
3  {
4      var actual = PatternBasedGenerator.ExpandPattern("g{5}");
5      Assert.AreEqual("ggggg", actual);
6  }
```

Listing B.21: C# showing the base version of the ChangeKey() method of the XorredBuffer class of the KeePassLib project.

```
 1  public byte[] ChangeKey(byte[] pbNewXorPad)
 2  {
 3    Debug.Assert(pbNewXorPad != null); if(pbNewXorPad == null) throw new ArgumentNullException("pbNewXorPad");
 4
 5    Debug.Assert(pbNewXorPad.Length == m_pbData.Length);
 6    if(pbNewXorPad.Length != m_pbData.Length) throw new ArgumentException();
 7
 8    if(m_pbXorPad.Length == m_pbData.Length) // Data is protected
 9    {
10      for(int i = 0; i < m_pbData.Length; ++i)
11        m_pbData[i] ^= (byte)(m_pbXorPad[i] ^ pbNewXorPad[i]);
12    }
13    else // Data is unprotected
14    {
15      for(int i = 0; i < m_pbData.Length; ++i)
16        m_pbData[i] ^= pbNewXorPad[i];
17    }
18
19    m_pbXorPad = pbNewXorPad;
20    return m_pbData;
21  }
```

Listing B.22: C# showing the `manual` version of the `ChangeKey()` method, after SME.

```
1  public byte[] ChangeKey(byte[] pbNewXorPad)
2  {
3    Debug.Assert(pbNewXorPad != null); if(pbNewXorPad == null) throw new ArgumentNullException("pbNewXorPad");
4
5          m_pbData = InternalChangeKey(m_pbData, m_pbXorPad, pbNewXorPad);
6
7    m_pbXorPad = pbNewXorPad;
8    return m_pbData;
9  }
10
11 internal static byte[] InternalChangeKey(byte[] pbData, byte[] pbXorPad, byte[] pbNewXorPad)
12 {
13     var result = new byte[pbData.Length];
14     Debug.Assert(pbNewXorPad.Length == pbData.Length);
15     if(pbNewXorPad.Length != pbData.Length) throw new ArgumentException();
16
17     if(pbXorPad.Length == pbData.Length) // Data is protected
18     {
19         for(int i = 0; i < pbData.Length; ++i)
20             result[i] = (byte)(pbData[i] ^ (pbXorPad[i] ^ pbNewXorPad[i]));
21     }
22     else // Data is unprotected
23     {
24         for(int i = 0; i < pbData.Length; ++i)
25             result[i] = (byte)(pbData[i] ^ pbNewXorPad[i]);
26     }
27     return result;
28 }
```

Listing B.23: C# showing the test for the `base` version of the `ChangeKey()` method.

```
1  [Test]
2  public void ChangeKey()
3  {
4      // arrange
5      var data = new byte[] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
6      var firstPad = new byte[] { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };
7      var protectedData = new byte[data.Length];
8      for (var i = 0; i < data.Length; i++)
9      {
10         protectedData[i] = (byte) (data[i] ^ firstPad[i]);
11     }
12     var xb = new XorredBuffer(protectedData, firstPad);
13     var secondPad = new byte[] { 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27 };
14
15     // act
16     var actual = xb.ChangeKey(secondPad);
17
18     // assert
19     Assert.AreEqual(data.Length, actual.Length);
20     Assert.AreSame(protectedData, actual);
21     for (var i = 0; i < data.Length; i++ )
22     {
23         Assert.AreEqual(0x20, actual[i]);
24     }
25 }
```

Listing B.24: C# showing the test for the `manual` version of the `ChangeKey()` method.

```
1  [Test]
2  public void InternalChangeKey()
3  {
4      // arrange
5      var data = new byte[] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
6      var firstPad = new byte[] { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };
7      var protectedData = new byte[data.Length];
8      for (var i = 0; i < data.Length; i++)
9      {
10         protectedData[i] = (byte) (data[i] ^ firstPad[i]);
11     }
12     var secondPad = new byte[] { 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27 };
13
14     // act
15     var actual = XorredBuffer.InternalChangeKey(protectedData, firstPad, secondPad);
16
17     // assert
18     Assert.AreEqual(data.Length, actual.Length);
19     for (var i = 0; i < data.Length; i++)
20     {
21         Assert.AreEqual(0x20, actual[i]);
22     }
23 }
```

Listing B.25: C# showing the base version of the FormatResultMessage() method of the AdminMenuItemController class of the AtomicCms project.

```
 1  private void FormatResultMessage(IMenuItem menuItem, int? id)
 2  {
 3      if (null == id || 0 == id)
 4      {
 5          TempData["SaveResult"] = string.Format("Items was successfully created with Id = {0}",
 6                                                 menuItem.Id);
 7      }
 8      else
 9      {
10          TempData["SaveResult"] = "Items was successfully updated";
11      }
12  }
```

Listing B.26: C# showing the manual version of the FormatResultMessage() method, after SME.

```
1  private void FormatResultMessage(IMenuItem menuItem, int? id)
2  {
3      TempData["SaveResult"] = InnerFormatResultMessage(id, menuItem.Id);
4  }
5
6  internal static string InnerFormatResultMessage(int? id, int menuItemId)
7  {
8      string result;
9      if (null == id || 0 == id)
10     {
11         result = string.Format("Items was successfully created with Id = {0}",
12                                menuItemId);
13     }
14     else
15     {
16         result = "Items was successfully updated";
17     }
18     return result;
19 }
```

Listing B.27: C# showing the test for the base version of the FormatResultMessage() method.

```
1  [Test]
2  public void FormatResultMessage_Created()
3  {
4      // arrange
5      var serviceFactory = new TestServiceFactory
6      {
7          MenuService = new MenuServiceMole
8          {
9              SaveMenuItem = i => { },
10         },
11     };
12     var iut = new AdminMenuItemController(serviceFactory);
13     var item = new MenuItem
14     {
15         Id = 42,
16     };
17     var formCollection = new FormCollection();
18
19     // act - EditMenuItem eventually calls FormatResultMessage
20     iut.EditMenuItem(null, item, formCollection);
21
22     // assert
23     var actual = iut.TempData["SaveResult"];
24     Assert.AreEqual("Items was successfully created with Id = 42", actual);
25 }
```

Listing B.28: C# showing the test for the `manual` version of the `FormatResultMessage()` method.

```
1  [Test]
2  public void InnerFormatResultMessage_Created()
3  {
4      // act
5      var actual = AdminMenuItemController.InnerFormatResultMessage(null, 42);
6
7      // assert
8      Assert.AreEqual("Items was successfully created with Id = 42", actual);
9  }
```

Appendix C

SME Applications

The following pages list all the methods that were transformed with SME, notes regarding the transformation and the revision in the Subversion repository where the changes took place.

Project		
Namespace		
Class		
Method	Notes	Revision
Textile		
Textile.Blocks		
BoldPhraseBlockModifier		
string ModifyLine(string)	Extract method	126
CapitalBlocksModifier		
string CapitalsFormatMatchEvaluator(Match)	Increase visibility	99
CitePhraseBlockModifier		
string ModifyLine(string)	Extract method	129
CodeBlockModifier		
string ModifyLine(string)	Extract method. The regular expression string and the Regex instance were later extracted into constants	130
string Conclude(string)	Extract method	130
string CodeFormatMatchEvaluator(Match)	Extract method	100
EmphasisPhraseBlockModifier		
string ModifyLine(string)	Extract method	132
FootNoteReferenceBlockModifier		
string ModifyLine(string)	Extract method	133
GlyphBlockModifier		
string ModifyLine(string)	Extract method	134
HyperLinkBlockModifier		
string ModifyLine(string)	Extract method	135
string HyperLinksFormatMatchEvaluator(Match)	Extract method	135
ImageBlockModifier		
string ModifyLine(string)	Extract method	136
string ImageFormatMatchEvaluator(Match)	Extract method. Also fixed a bug in the handling of href	136
InsertedPhraseBlockModifier		
string ModifyLine(string)	Extract method	137
ItalicPhraseBlockModifier		
string ModifyLine(string)	Extract method	138
NoTextileBlockModifier		
string ModifyLine(string)	Extract method	139
string Conclude(string)	Extract method	139
NoTextileEncoder		
string EncodeNoTextileZonesMatchEvaluator(Match)	Extract method	140
string DecodeNoTextileZonesMatchEvaluator(Match)	Extract method	140
PhraseBlockModifier		
string PhraseModifierFormat(string, string, string)	Extract method	265
PhraseBlockModifier.PhraseModifierMatchEvaluator	Increase visibility	99
string MatchEvaluator(Match)	Extract method	141

Project		
Namespace		
Class		
Method	Notes	Revision
PreBlockModifier		
string ModifyLine(string)	Extract method	142
string Conclude(string)	Extract method	142
SpanPhraseBlockModifier		
string ModifyLine(string)	Extract method	143
StrongPhraseBlockModifier		
string ModifyLine(string)	Extract method	144
Textile.States		
CodeFormatterState		
string FixEntities(string)	Increase visibility and scope	99
FootNoteFormatterState		
void Enter()	Extract method	202
void OnContextAcquired()	Extract method	99
OrderedListFormatterState		
bool IsMatchForMe(string, int, int)	Extract method	99
bool IsMatchForOthers(string, int, int)	Extract method	99
PreCodeFormatterState		
string FixEntities(string)	Increase visibility and scope	99
TableCellParser		
string GetLineFragmentFormatting()	Extract method	99
TableFormatterState		
bool ShouldExit(string)	Extract method	99
TableRowFormatterState		
void FormatLine(string)	Extract method	99
UnorderedListFormatterState		
bool IsMatchForMe(string, int, int)	Extract method	99
bool IsMatchForOthers(string, int, int)	Extract method	99
Textile		
TextileFormatter		
void Format(string)	Extract method	99
string PrepareInputForFormatting(string)	Increase visibility and scope	99
string CleanWhiteSpace(string)	Increase visibility and scope	99
string HandleFormattingState(string)	Extract method	100

Project Namespace Class Method	Notes	Revision
KeePassLib		
KeePassLib.Cryptography		
CryptoRandom		
void AddEntropy(byte[])	Extract methods	270, 271
byte[] GenerateRandom256()	Extract method	272
CryptoRandomStream		
CryptoRandomStream .ctor(CrsAlgorithm, byte[])	Extract method	273
ulong GetRandomUInt64()	Extract method	104
Salsa20Cipher		
void NextOutput()	Extract method	104
uint Rotl32(uint, int)	Increase visibility	104
uint U8To32Little(byte[], int)	Increase visibility	104
void KeySetup(byte[])	Extract method	104
void IvSetup(byte[])	Extract method	104
StandardAesEngine		
void ValidateArguments(Stream, bool, byte[], byte[])	Increase visibility	104
Stream CreateStream(Stream, bool, byte[], byte[])	Increase visibility	104
HmacOtp		
uint CalculateChecksum(uint, uint)	Increase visibility	104
KeePassLib.Cryptography.PasswordGenerator		
CustomPwGeneratorPool		
int FindIndex(PwUuid)	Extract method	274
PatternBasedGenerator		
string ExpandPattern(string)	Increase visibility	104
PwCharSet		
bool Contains(char)	Extract method	274
PwGenerator		
CryptoRandomStream CreateCryptoStream(byte[])	Increase visibility	104
PwgError GenerateCustom(ProtectedString, PwProfile, CryptoRandomStream, CustomPwGeneratorPool)	Increase visibility	104
PwProfile		
PwProfile DeriveFromPassword(ProtectedString)	Extract method	104
KeePassLib.Keys		
CompositeKey		
byte[] CreateRawCompositeKey32()	Extract method	105
void ValidateUserKeys()	Extract method	105
byte[] TransformKey(byte[], byte[], ulong)	Increase visibility	105
KcpKeyFile		
byte[] LoadKeyFile(string)	Extract method	275

Project		
Namespace		
Class		
Method	**Notes**	**Revision**
byte[] LoadBinaryKey32(byte[])	Increase visibility	105
byte[] LoadHexKey32(byte[])	Increase visibility	105
byte[] LoadXmlKeyFile(string)	Extract method	277
KcpPassword		
void SetKey(byte[])	Extract method	278
KeyValidatorPool		
string Validate(string, KeyValidationType)	Extract method	280
KeePassLib.Security		
XorredBuffer		
byte[] ChangeKey(byte[])	Extract method which creates a new byte array instead of mutating one	370
KeePassLib.Serialization		
HashedBlockStream		
bool ReadHashedBlock()	Extract methods	283, 284
IOConnection		
bool ValidateServerCertificate(object, X509Certificate, X509Chain, SslPolicyErrors)	Increase visibility	106
IOConnectionInfo		
string GetDisplayName()	Extract method	106
bool IsLocalFile()	Extract method	106
Kdb4File		
XmlReader CreateXmlReader(Stream)	Increase visibility	106
void ReadHeader(BinaryReaderEx)	Extract methods	106, 286
void SetCipher(byte[])	Extract method	287
void SetCompressionFlags(byte[])	Extract method	287
void SetInnerRandomStreamID(byte[])	Extract method	287
KeePassLib.Translation		
KpccLayout		
int? GetModControlParameter(Control, LayoutParameterEx, string)	Increase visibility	107
void WriteControlDependentParams(StringBuilder, Control)	Increase visibility	107
void WriteCpiParam(StringBuilder, string)	Increase visibility	107
KPTranslation		
void RtlApplyToControls(Control.ControlCollection)	Increase visibility	107
void RtlMoveChildControls(Control)	Increase visibility	107
void RtlApplyToToolStripItems(ToolStripItemCollection)	Increase visibility	107
KeePassLib.Utility		
MemUtil		
byte[] HexStringToByteArray(string)	Extract method	107
MessageService		

Project		
Namespace		
Class		
Method	Notes	Revision
string ObjectsToMessage(object[])	Increase visibility	107
string ObjectsToMessage(object[], bool)	Increase visibility	107
CharStream		
char ReadChar(bool)	Extract method	107
KeePassLib		
PwDatabase		
uint FindLocationChangedPivot<T>(PwObjectList<T>, KeyValuePair<uint, uint>, PwObjectPool, PwObjectPool, Queue<PwUuid>, Queue<PwUuid>, bool)	Increase visibility	108
void GetNeighborItems(List<IStructureItem>, PwUuid, Queue<PwUuid>, Queue<PwUuid>)	Increase visibility	108
void RemoveCustomIconUuid(PwEntry, List<PwUuid>)	Increase visibility	108
PwGroup		
void LinearizeGroupRecursive(LinkedList<PwGroup>, PwGroup, ushort)	Increase visibility and scope	108
void SearchEvalAdd(string, string, StringComparison, Regex, PwEntry, PwObjectList<PwEntry>)	Increase visibility	108

Project		
Namespace		
Class		
Method	Notes	Revision
AtomicCms		
AtomicCms.Common.Utils		
SimpleHash		
bool VerifyHash(string, Algorith, string)	Extract method	417
string ComputeHash(string, Algorith, byte[])	Extract method	417
Strong		
string PropertyName(Expression)	Increase visibility	417
string MethodName(Expression)	Increase visibility	417
AtomicCms.Core.Models		
EntryService		
int GetParsedDefaultPageId()	Extract method	417
IEnumerable<IEntry> TruncateEntryTitle(IEnumerable<IEntry>)	Increase visibility and scope	417
IEntry Trunc(IEntry)	Increase visibility and scope	417
InfrastructureService		
SyndicationFeed BuildFeed(string, Func<IEntry, string>)	Extract methods	417
ChangeFrequency CalculateFrequency(DateTime)	Extract method	417
SeoService		
string CreateAlias(string)	Extract method	417
string Sanitarize(string)	Increase visibility and scope	417
AtomicCms.Web.Controllers		
AdminMenuItemController		
void FormatResultMessage(IMenuItem, int?)	Extract method	417, 420
AtomicCms.Web.Core.Extensions		
LocalizationExtensions		
string GetResourceString(HttpContextBase, string, string, object[])	Increase visibility	417
string GetVirtualPath(HtmlHelper)	Increase visibility	417
AtomicCms.Web.Core.Mvc		
SkinSupportViewEngine		
string[] AddNewLocationFormats(IEnumerable<string>, *IEnumerable<string>*)	*Increase visibility and scope*	417
string OverrideMasterPage(string, ControllerContext)	Increase visibility and scope	417
bool NeedChangeMasterPage(ControllerContext)	Increase visibility and scope	417
AtomicCms.Web		
MvcApplication		
bool RemoveDoubleSlashes(string)	Extract method	417
bool RemoveWWWPrefix(string, string, string, string, out string)	Extract method	417
bool AddTrailingSlash(string)	Extract method	417

Appendix D

Raw Data Spreadsheet

The following pages reproduce the "Raw data" sheet of the online spreadsheet[1], which was used by the "Record results" step described in section 4.2.6. The spreadsheet contains every row submitted to it by this step, with many of these rows created by investigating various tests and tweaks. Its columns are explained in table D.1. Notes were occasionally added in the "Notes" column to explain the test or tweak being investigated.

Column name	Description
Timestamp	The date and time the result row was inserted
Batch #	The revision of the Subversion repository or `private` if run for testing purposes instead of in a batch
Project	The name of the project
Strategy	The name of the strategy employed
TSP	Total Sequence Points, the denominator for code coverage percentage calculation
VSP	Visited Sequence Points, the numerator for code coverage percentage calculation
PIC	Public Interface Changes
MI IUT	Maintainability Index of the Implementation Under Test
MI MT	Maintainability Index of the Manually-written unit Tests

Table D.1: Names and descriptions of the columns in the "Raw data" sheet of the online results spreadsheet.

[1]The online spreadsheet can be seen at `https://docs.google.com/spreadsheet/ccc?key=0Ag6eugnyWA09dG9zQnFtTlQ0Z1dGUjI5TUlFMnpFWWc`

Timestamp	Batch #	Project	Strategy	TSP	VSP	PIC	MI IUT	MI MT	Percent coverage	Notes
4/18/2010 12:01:01	10	StringExtensions	base	15	15				100.00%	
4/18/2010 12:01:03	10	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:04:11	11	StringExtensions	base	15	15				100.00%	
4/18/2010 12:04:12	11	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:10:21	12	StringExtensions	base	15	15				100.00%	
4/18/2010 12:10:21	12	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:25:41	13	StringExtensions	base	15	15				100.00%	
4/18/2010 12:25:41	13	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:27:55	15	StringExtensions	base	15	15				100.00%	
4/18/2010 12:27:55	15	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:31:26	16	StringExtensions	base	15	15				100.00%	
4/18/2010 12:31:28	16	StringExtensions	testoriented	15	15				100.00%	
4/18/2010 12:35:29	17	StringExtensions	base	15	15				100.00%	
4/18/2010 12:35:29	17	StringExtensions	testoriented	15	15				100.00%	
4/21/2010 7:16:21	19	StringExtensions	base	15	15				100.00%	
4/21/2010 7:16:22	19	StringExtensions	testoriented	15	15				100.00%	
4/21/2010 20:41:29	private	KeePassLib	base	8113	2632				32.44%	
4/21/2010 20:41:32	private	StringExtensions	base	15	15				100.00%	
4/21/2010 21:22:32	21	KeePassLib	base	8113	2575				31.74%	
4/21/2010 21:22:32	21	StringExtensions	base	15	15				100.00%	
4/24/2010 14:58:02	private	Textile	base	853	517				60.61%	
4/24/2010 15:19:51	private	KeePassLib	base	8113	2632				32.44%	
4/24/2010 15:19:52	private	StringExtensions	base	15	15				100.00%	
4/24/2010 15:19:52	private	Textile	base	853	517				60.61%	
4/24/2010 15:48:08	22	KeePassLib	base	8113	2611				32.18%	
4/24/2010 15:48:09	22	StringExtensions	base	15	15				100.00%	
4/24/2010 15:48:12	22	Textile	base	853	517				60.61%	
4/24/2010 19:49:29	private	AtomicCms	base	1025	502				48.98%	
4/24/2010 21:26:44	private	AtomicCms	base	1025	502				48.98%	
4/24/2010 22:26:25	24	AtomicCms	base	1025	503				49.07%	
4/24/2010 22:26:26	24	KeePassLib	base	8113	2609				32.16%	
4/24/2010 22:26:29	24	StringExtensions	base	15	15				100.00%	
4/24/2010 22:26:29	24	Textile	base	853	517				60.61%	
5/1/2010 15:20:56	27	AtomicCms	base	1025	502				48.98%	
5/1/2010 16:53:39	private	StringExtensions	base	15	15				100.00%	
5/1/2010 16:56:58	28	StringExtensions	base	15	15				100.00%	
5/1/2010 17:04:08	30	StringExtensions	base	15	15				100.00%	
5/1/2010 17:12:51	private	StringExtensions	base	15	15				100.00%	
5/1/2010 17:16:32	31	StringExtensions	base	15	15				100.00%	
5/3/2010 22:07:27	1	KeePassLib	base	8113	2635				32.48%	
5/3/2010 22:29:44	3	StringExtensions	base	15	15				100.00%	
5/3/2010 22:30:38	3	Textile	base	853	517				60.61%	
5/3/2010 22:49:13	4	AtomicCms	base	1025	497				48.49%	
5/3/2010 23:31:23	4	KeePassLib	base	7624	3475				45.58%	
5/8/2010 17:18:29	4	StringExtensions	base	15	15				100.00%	
5/8/2010 17:22:56	4	Textile	base	853	517				60.61%	
5/8/2010 17:24:39	5	StringExtensions	base	15	15				100.00%	
5/8/2010 17:24:46	5	Textile	base	853	517				60.61%	
5/8/2010 17:34:50	5	AtomicCms	base	1025	497				48.49%	
5/8/2010 17:56:03	6	AtomicCms	base	1025	497				48.49%	
5/9/2010 12:14:03	private	KeePassLib	base	7624	3475				45.58%	
5/9/2010 12:50:58	private	KeePassLib	base	7624	3473				45.55%	
5/9/2010 13:33:55	private	KeePassLib	base	7624	3472				45.54%	
5/29/2010 10:49:29	6	StringExtensions	base	15	15				100.00%	
5/29/2010 10:51:04	6	Textile	base	853	517				60.61%	336
5/29/2010 10:55:35	1	Textile	manual	853	513				60.14%	340
5/29/2010 11:10:17	7	AtomicCms	base	1025	497				48.49%	528
5/29/2010 14:58:16	private	Textile	manual	873	539				61.74%	334
5/29/2010 15:52:54	private	Textile	manual	877	543				61.92%	334

Timestamp	Batch #	Project	Strategy	TSP	VSP	PIC	MI IUT	MI MT	Percent coverage	Notes
5/29/2010 15:57:26	7	StringExtensions	base	15	15				100.00%	0
5/29/2010 15:59:11	7	Textile	base	853	517				60.61%	336
5/29/2010 16:04:42	2	Textile	manual	877	625				71.27%	252
5/29/2010 16:19:36	8	AtomicCms	base	1025	497				48.49%	
5/29/2010 17:35:05	private	Textile	manual	877	543				61.92%	
5/29/2010 18:33:43	private	Textile	manual	886	556				62.75%	
5/30/2010 10:00:34	private	Textile	manual	884	555				62.78%	
5/30/2010 10:05:07	8	StringExtensions	base	15	15				100.00%	
5/30/2010 10:09:11	3	Textile	manual	884	554				62.67%	
5/30/2010 10:10:47	8	Textile	base	853	511				59.91%	
6/1/2010 20:03:13	private	StringExtensions	base	15	15				100.00%	
6/1/2010 20:06:21	private	StringExtensions	base	15	15				100.00%	
6/1/2010 20:24:04	private	StringExtensions	base	15	15				100.00%	
6/1/2010 20:24:24	private	StringExtensions	base	15	15				100.00%	
6/1/2010 20:41:38	4	Textile	manual	884	555				62.78%	
6/1/2010 20:42:02	9	Textile	base	853	517				60.61%	
6/1/2010 20:54:48	11	StringExtensions	base	15	15				100.00%	
6/1/2010 21:32:52	11	KeePassLib	base	7624	3476				45.59%	
6/1/2010 21:40:41	private	AtomicCms	base	1025	503				49.07%	
6/1/2010 21:44:37	12	StringExtensions	base	15	15				100.00%	
6/1/2010 21:45:47	10	Textile	base	853	517				60.61%	
6/1/2010 21:48:52	5	Textile	manual	884	555				62.78%	
6/1/2010 22:12:34	12	AtomicCms	base	1025	503				49.07%	
6/1/2010 22:16:14	12	KeePassLib	base	7624	3483				45.68%	
6/5/2010 14:38:03	11	Textile	base	853	517				60.61%	
6/5/2010 14:39:23	6	Textile	manual	884	554				62.67%	
6/5/2010 14:45:18	14	StringExtensions	base	15	15				100.00%	
6/5/2010 14:55:44	13	AtomicCms	base	1025	497				48.49%	
6/5/2010 15:05:27	13	KeePassLib	base	7624	3487				45.74%	
6/5/2010 15:05:32	1	KeePassLib	manual	7624	3484				45.70%	
6/27/2010 12:50:44	private	KeePassLib	manual	7634	3495				45.78%	
7/3/2010 11:16:54	15	StringExtensions	base	15	15				100.00%	
7/3/2010 11:23:22	7	Textile	manual	884	551				62.33%	
7/3/2010 11:41:49	15	AtomicCms	base	1025	502				48.98%	
7/3/2010 11:48:27	13	Textile	base	853	517				60.61%	
7/3/2010 11:49:18	2	KeePassLib	manual	7634	3489				45.70%	
7/3/2010 11:52:20	15	KeePassLib	base	7624	3480				45.65%	
7/3/2010 12:42:57	private	Textile	manual	884	789				89.25%	Trying Pex 0.92, but would not run reliably enough
7/10/2010 11:01:17	16	StringExtensions	base	15	15				100.00%	
7/10/2010 11:03:06	14	Textile	base	853	513				60.14%	
7/10/2010 11:06:25	8	Textile	manual	884	554				62.67%	
7/10/2010 11:24:54	16	AtomicCms	base	1025	503				49.07%	
7/10/2010 11:33:50	16	KeePassLib	base	7624	3488				45.75%	
7/10/2010 12:08:08	private	KeePassLib	manual	7639	3496				45.77%	
7/10/2010 14:22:49	5	KeePassLib	manual	7639	3496				45.77%	
7/10/2010 17:34:36	15	Textile	base	853	513				60.14%	
7/10/2010 17:38:22	9	Textile	manual	884	549				62.10%	
7/10/2010 17:55:16	17	AtomicCms	base	1025	503				49.07%	
7/10/2010 18:04:10	6	KeePassLib	manual	7659	3502				45.72%	
7/10/2010 18:04:45	17	KeePassLib	base	7624	3484				45.70%	
7/10/2010 19:25:04	18	StringExtensions	base	15	15				100.00%	
7/10/2010 20:05:13	19	StringExtensions	base	15	15				100.00%	
7/10/2010 20:07:02	16	Textile	base	853	514				60.26%	
7/10/2010 20:11:17	10	Textile	manual	884	551				62.33%	
7/10/2010 20:27:42	18	AtomicCms	base	1025	503				49.07%	
7/10/2010 20:37:10	7	KeePassLib	manual	7659	3501				45.71%	
7/10/2010 20:37:40	18	KeePassLib	base	7624	3483				45.68%	
7/14/2010 22:22:56	private	KeePassLib	manual	7662	3497				45.64%	
7/14/2010 22:59:41	private	KeePassLib	manual	7662	3506				45.76%	

Timestamp	Batch #	Project	Strategy	TSP	VSP	PIC	Mi IUT	Mi MT	Percent coverage	Notes
7/17/2010 11:31:07	private	KeePassLib	manual	7662	3506				45.76%	
7/17/2010 12:05:33	private	KeePassLib	manual	7662	3504				45.73%	
7/17/2010 13:29:23	private	Textile	manual	884	555				62.78%	
7/24/2010 13:36:17	20	StringExtensions	base	15	15				100.00%	
7/24/2010 13:41:22	11	Textile	manual	884	555				62.78%	
7/24/2010 13:42:37	17	Textile	base	853	515				60.38%	
7/24/2010 13:57:31	19	AtomicCms	base	1025	503				49.07%	
7/24/2010 14:07:18	8	KeePassLib	manual	7659	3502				45.72%	
7/24/2010 14:07:44	19	KeePassLib	base	7624	3484				45.70%	
7/24/2010 14:26:16	private	AtomicCms	manual	1025	503				49.07%	
7/24/2010 14:31:22	21	StringExtensions	base	15	15				100.00%	
7/24/2010 14:32:25	18	Textile	base	853	517				60.61%	
7/24/2010 14:37:40	12	Textile	manual	884	544				61.54%	
7/24/2010 14:52:05	20	AtomicCms	base	1025	497				48.49%	
7/24/2010 14:52:54	1	AtomicCms	manual	1025	503				49.07%	
7/24/2010 16:57:50	private	KeePassLib	manual	7659	3505				45.76%	tweaked Pex, but took 45 minutes
7/25/2010 15:18:35	110	StringExtensions	base	15	15				100.00%	
7/25/2010 15:20:30	110	Textile	base	853	517				60.61%	
7/25/2010 15:23:56	110	Textile	manual	884	554				62.67%	
7/25/2010 15:41:29	110	AtomicCms	base	1025	497				48.49%	
7/25/2010 15:41:39	110	AtomicCms	manual	1025	503				49.07%	
7/25/2010 16:06:31	110	KeePassLib	base	7624	3484				45.70%	
7/25/2010 16:13:42	110	KeePassLib	manual	7659	3458				45.15%	
9/11/2010 11:33:03	private	KeePassLib	base	7624	3530				46.30%	
9/11/2010 11:53:56	private	KeePassLib	manual	7659	3777				49.31%	
9/11/2010 13:42:17	114	StringExtensions	base	15	15				100.00%	
9/11/2010 13:51:58	114	Textile	base	853	515				60.38%	
9/11/2010 14:01:33	114	Textile	manual	884	555				62.78%	
9/11/2010 14:03:03	114	AtomicCms	manual	1025	502				48.98%	
9/11/2010 14:03:08	114	AtomicCms	base	1025	503				49.07%	
9/11/2010 14:16:01	114	KeePassLib	base	7624	3525				46.24%	
9/11/2010 14:56:52	114	KeePassLib	manual	7659	3735				48.77%	
11/14/2010 21:35:45	private	PivotStack	original	299	195				65.22%	Early test with unfinished version
1/21/2011 11:18:41	private	StringExtensions	base	15	15				100.00%	
1/21/2011 11:29:26	private	PivotStack	base	834	395				47.36%	
1/21/2011 11:45:34	private	PivotStack	original	835	401				48.02%	
1/28/2011 11:04:26	private	PivotStack	project-original	835	468				56.05%	Temporarily back-ported the code coverage to original branch
1/28/2011 15:57:21	private	PivotStack	project-base	834	476				57.07%	
2/12/2011 12:42:46	private	PivotStack	project-base	872	514				58.94%	
2/12/2011 12:50:27	private	PivotStack	project-base	862	505				58.58%	
2/12/2011 14:59:48	private	PivotStack	project-base	861	504				58.54%	
2/12/2011 15:13:08	private	PivotStack	project-base	859	502				58.44%	
2/12/2011 15:14:35	private	PivotStack	project-base	857	500				58.34%	
2/12/2011 15:16:19	private	PivotStack	project-base	854	497				58.20%	
2/12/2011 15:17:43	private	PivotStack	project-base	854	497				58.20%	
2/12/2011 15:45:22	private	PivotStack	project-base	854	497				58.20%	
2/12/2011 15:47:43	private	PivotStack	project-base	850	497				58.47%	
2/12/2011 15:57:28	private	StringExtensions	base	33	15				45.45%	Testing with Eric's Math.zip
2/12/2011 15:58:50	private	StringExtensions	base	33	33				100.00%	Testing with Eric's Math.zip
2/15/2011 21:17:13	private	PivotStack	project-base	850	497				58.47%	
7/23/2011 21:03:27	private	KeePassLib	base	7624	3527				46.26%	
7/25/2011 19:37:12	private	KeePassLib	base	7622	3526				46.26%	
7/25/2011 19:37:13	private	KeePassLib	manual	7657	3775				49.30%	
7/25/2011 20:03:40	261	StringExtensions	base	15	15				100.00%	
7/26/2011 11:36:55	private	Textile	visibility	853	548	0			64.24%	
7/26/2011 11:40:04	private	Textile	base	853	517	0			60.61%	
7/26/2011 11:42:24	private	Textile	manual	928	624	0			67.24%	
7/26/2011 12:43:15	private	Textile	visibility	853	546	0			64.01%	
7/26/2011 21:48:37	private	Textile	visibility	853	548	0			64.24%	

Timestamp	Batch #	Project	Strategy	TSP	VSP	PIC	MI IUT	MI MT	Percent coverage	Notes
7/26/2011 21:50:42	private	Textile	base	853	517	0			60.61%	
7/26/2011 21:53:04	private	Textile	manual	928	624	0			67.24%	
7/26/2011 21:58:37	private	Textile	manual	928	624	0			67.24%	
7/27/2011 17:48:19	private	Textile	base	853	517	0			60.61%	
7/27/2011 17:48:57	private	Textile	visibility	853	548	90			64.24%	
7/27/2011 17:49:04	private	Textile	manual	930	624	0			67.10%	
7/27/2011 20:35:44	private	Textile	visibility-safe	853	548	90			64.24%	
7/27/2011 20:45:24	private	Textile	visibility	853	549	90			64.36%	
7/27/2011 20:46:39	private	Textile	visibility-safe	853	549	90			64.36%	
7/27/2011 21:09:55	private	Textile	visibility	853	549	90			64.36%	
7/27/2011 21:09:59	private	Textile	visibility-safe	853	549	180			64.36%	bug in DPI code
7/27/2011 21:39:11	private	Textile	visibility-safe	853	549	0			64.36%	
7/27/2011 21:39:11	private	Textile	visibility	853	549	90			64.36%	
7/27/2011 21:57:53	private	Textile	visibility	853	549	90			64.36%	
7/27/2011 21:57:54	private	Textile	visibility-safe	853	549	0			64.36%	
7/27/2011 22:37:24	private	Textile	visibility-state	853	518	0			60.73%	
7/27/2011 23:15:28	private	KeePassLib	base	7622	3519	0			46.17%	
7/28/2011 11:40:01	private	KeePassLib	manual	7657	3776	30			49.31%	bugs in PIC code
7/28/2011 13:51:52	private	KeePassLib	manual	7657	3776	0			49.31%	
7/28/2011 14:50:17	private	KeePassLib	base	7567	3482	0			46.02%	deleted SelfTest
7/28/2011 15:09:54	private	KeePassLib	manual	7602	3728	0			49.04%	deleted SelfTest
7/28/2011 19:49:44	private	KeePassLib	manual	7625	3736	0			49.00%	trying harder: hiding non-stateless and extracting more
7/28/2011 21:13:42	private	KeePassLib	visibility-state	7567	3464	0			45.78%	
7/28/2011 22:02:53	private	PivotStack	base	834	403	0			48.32%	
7/28/2011 22:09:14	private	KeePassLib	manual	7623	3739	0			49.05%	even more improvements for authenticity
8/28/2011 10:03:05	private	PivotStack	manual	846	411	0			48.58%	
9/3/2011 15:07:51	private	StringExtensions	base	15	15	0			100.00%	
9/3/2011 17:09:46	private	StringExtensions	manual	15	15	0			100.00%	
9/25/2011 11:17:26	private	Textile	base	853	517	0	83		60.61%	MI metric is for the manual tests
9/25/2011 11:17:26	private	Textile	manual	930	624	0	90		67.10%	MI metric is for the manual tests
9/25/2011 11:41:22	private	Textile	base	853	517	0	83		60.61%	MI metric is for the IUT
9/25/2011 11:41:23	private	Textile	manual	930	624	0	84		67.10%	MI metric is for the IUT
9/25/2011 15:52:37	private	Textile	base	853	517	0	83	83	60.61%	Both MI metrics recorded separately
9/25/2011 15:52:37	private	Textile	manual	930	624	0	84	90	67.10%	Both MI metrics recorded separately
10/10/2011 14:10:47	private	KeePassLib	base	7567	3484	0	81	100	46.04%	
10/10/2011 14:10:48	private	KeePassLib	visibility	7567	3624	0	81	100	47.89%	
10/10/2011 14:10:49	private	KeePassLib	visibility-state	7567	3469	0	81	100	45.84%	
10/10/2011 14:26:03	private	Textile	visibility-state	853	518	0	83	83	60.73%	
10/10/2011 14:26:13	private	Textile	visibility	853	549	180	83	83	64.36%	non-zero PIC was because of bug in PIC tool
10/10/2011 15:02:22	private	KeePassLib	manual	7623	3744	0	81	100	49.11%	
10/17/2011 10:02:27	356	Textile	visibility	853	549	180	83	83	64.36%	non-zero PIC was because of bug in PIC tool
10/17/2011 10:12:16	356	Textile	visibility-state	853	517	0	83	83	60.61%	
10/17/2011 10:17:37	356	Textile	manual	930	624	0	84	90	67.10%	
10/17/2011 10:27:13	346	KeePassLib	visibility-state	7567	3471	0	81	100	45.87%	
10/17/2011 10:31:33	356	Textile	base	853	514	0	83	83	60.26%	
10/17/2011 10:48:39	356	Textile	visibility-state	853	517	0	83	83	60.61%	
10/17/2011 10:53:17	356	KeePassLib	base	7567	3489	0	81	100	46.11%	
10/17/2011 11:04:03	356	KeePassLib	visibility	7567	3664	0	81	100	48.42%	
10/17/2011 11:20:49	356	KeePassLib	manual	7623	3744	0	81	100	49.11%	
10/17/2011 11:27:46	356	KeePassLib	visibility-state	7567	3466	0	81	100	45.80%	
10/17/2011 15:14:03	private	Textile	visibility	853	549	0	83	83	64.36%	confirmed fix for bug in PIC
10/17/2011 15:23:44	361	Textile	base	853	517	0	83	83	60.61%	
10/17/2011 15:34:49	361	Textile	manual	930	624	0	84	90	67.10%	
10/17/2011 15:50:13	361	Textile	visibility	853	546	0	83	83	64.01%	
10/17/2011 15:55:37	361	KeePassLib	base	7567	3442	0	81	100	45.49%	
10/17/2011 15:57:08	361	Textile	visibility-state	853	517	0	83	83	60.61%	
10/17/2011 16:29:54	361	KeePassLib	manual	7623	3739	0	81	100	49.05%	
10/17/2011 16:41:59	361	KeePassLib	visibility	7567	3619	0	81	100	47.83%	
10/17/2011 22:13:20	361	KeePassLib	visibility-state	7567	3478	0	81	100	45.96%	

Timestamp	Batch #	Project	Strategy	TSP	VSP	PIC	MI IUT	MI MT	Percent coverage	Notes
10/18/2011 9:22:08	366	Textile	base	853	514	0	83	83	60.26%	
10/18/2011 9:38:34	366	Textile	manual	930	624	0	84	90	67.10%	
10/18/2011 9:48:38	366	KeePassLib	base	7567	3481	0	81	100	46.00%	
10/18/2011 9:49:51	366	Textile	visibility	853	548	0	83	83	64.24%	
10/18/2011 9:59:06	366	Textile	visibility-state	853	516	0	83	83	60.49%	
10/18/2011 10:58:14	366	KeePassLib	manual	7623	3740	0	81	100	49.06%	
10/18/2011 11:03:46	366	KeePassLib	visibility-state	7567	3470	0	81	100	45.86%	
10/18/2011 11:11:00	366	KeePassLib	visibility	7567	3658	0	81	100	48.34%	
10/18/2011 15:25:42	private	Textile	base	853	517	0	83	78	60.61%	Focusing the manual tests based on section 4.2.4
10/18/2011 16:07:13	private	Textile	manual	931	625	0	84	85	67.13%	Focusing the manual tests based on section 4.2.4
10/18/2011 16:29:17	private	Textile	visibility	853	549	0	83	76	64.36%	Focusing the manual tests based on section 4.2.4
10/18/2011 16:31:31	private	Textile	visibility-state	853	516	0	83	76	60.49%	Focusing the manual tests based on section 4.2.4
10/18/2011 18:20:10	368	Textile	manual	931	625	0	84	85	67.13%	
10/18/2011 18:22:11	368	Textile	visibility	853	549	0	83	76	64.36%	
10/18/2011 18:27:45	368	Textile	base	853	515	0	83	78	60.38%	
10/18/2011 18:27:57	368	Textile	visibility-state	853	516	0	83	76	60.49%	
10/18/2011 18:53:30	368	KeePassLib	visibility-state	7567	3478	0	81	71	45.96%	
10/18/2011 19:28:20	368	KeePassLib	visibility	7567	3651	0	81	92	48.25%	
10/18/2011 20:20:24	368	KeePassLib	base	7567	3477	0	81	71	45.95%	
10/18/2011 20:36:49	368	KeePassLib	manual	7623	3740	0	81	92	49.06%	
10/19/2011 13:31:36	private	PivotStack	base	834	402	0	77	71	48.20%	testing possibility of including PivotStack
10/19/2011 13:31:36	private	PivotStack	manual	835	407	33	78	71	48.74%	PivotStack was converted with breaking changes
10/19/2011 13:55:36	378	Textile	base	853	515	0	83	78	60.38%	
10/19/2011 14:07:02	378	Textile	manual	931	624	0	84	85	67.02%	
10/19/2011 14:17:43	378	Textile	visibility	853	548	0	83	76	64.24%	
10/19/2011 14:24:48	378	KeePassLib	base	7567	3466	0	81	70	45.80%	
10/19/2011 14:26:28	378	Textile	visibility-state	853	518	0	83	76	60.73%	
10/19/2011 14:42:25	378	KeePassLib	manual	7627	3748	0	81	81	49.14%	
10/19/2011 14:55:15	378	KeePassLib	visibility	7567	3657	0	81	80	48.33%	
10/19/2011 14:59:00	378	KeePassLib	visibility-state	7567	3481	0	81	70	46.00%	
10/19/2011 17:09:38	393	Textile	base	853	517	0	83	78	60.61%	
10/19/2011 17:09:58	393	Textile	visibility	853	549	0	83	76	64.36%	
10/19/2011 17:10:12	393	Textile	manual	931	625	0	84	85	67.13%	
10/19/2011 17:13:49	393	Textile	visibility-state	853	516	0	83	76	60.49%	
10/19/2011 17:45:19	393	KeePassLib	base	7567	3489	0	81	70	46.11%	
10/19/2011 18:03:20	393	KeePassLib	manual	7627	3751	0	81	81	49.18%	
10/19/2011 18:23:21	393	KeePassLib	visibility-state	7567	3460	0	81	70	45.72%	
10/19/2011 18:48:00	393	KeePassLib	visibility	7567	3661	0	81	80	48.38%	
10/20/2011 17:01:42	private	AtomicCms	base	1025	517	0	87	100	50.44%	testing possibility of including AtomicCms
10/20/2011 17:01:42	private	AtomicCms	visibility	1025	595	0	87	100	58.05%	
10/20/2011 17:01:43	private	AtomicCms	visibility-state	1025	518	0	87	100	50.54%	
10/21/2011 16:50:17	private	AtomicCms	manual	1052	580	0	87	100	55.13%	not many stateless methods could be extracted
11/2/2011 21:28:32	417	Textile	base	853	513	0	83	78	60.14%	
11/2/2011 21:42:04	417	AtomicCms	base	1025	510	0	87	100	49.76%	MT has 100 because there are no tests
11/2/2011 21:43:41	417	Textile	manual	931	625	0	84	85	67.13%	
11/2/2011 21:46:46	417	AtomicCms	visibility	1025	594	0	87	100	57.95%	*MT has 100 because there are no tests*
11/2/2011 21:47:02	417	AtomicCms	manual	1052	580	0	87	100	55.13%	MT has 100 because there are no tests
11/2/2011 21:58:14	417	Textile	visibility	853	547	0	83	76	64.13%	
11/2/2011 22:05:43	417	AtomicCms	visibility-state	1025	516	0	87	100	50.34%	MT has 100 because there are no tests
11/2/2011 22:09:27	417	Textile	visibility-state	853	517	0	83	76	60.61%	
11/2/2011 22:24:50	417	KeePassLib	base	7567	3485	0	81	70	46.06%	
11/2/2011 22:45:13	417	KeePassLib	manual	7627	3752	0	81	81	49.19%	
11/5/2011 9:31:55	417	KeePassLib	visibility	7567	3652	0	81	80	48.26%	
11/5/2011 14:41:48	417	KeePassLib	visibility-state	7567	3467	0	81	70	45.82%	
11/18/2011 16:05:31	422	Textile	base	853	515	0	83	78		422 had a broken AtomicCms, plus some Pex-related failure
11/18/2011 16:08:53	422	Textile	manual	931	625	0	84	85		
11/18/2011 16:12:18	422	Textile	visibility-state	853	518	0	83	76		
11/18/2011 16:17:25	422	Textile	visibility	853	549	0	83	76		
11/18/2011 16:40:15	422	KeePassLib	visibility-state	7567	3478	0	81	70		

Timestamp	Batch #	Project	Strategy	TSP	VSP	PtC	MI IUT	MI MT	Percent coverage	Notes
11/18/2011 17:06:10	424	Textile	base	853	510	0	83	78	59.79%	
11/18/2011 17:10:51	422	KeePassLib	visibility	7567	3658	0	81	80		
11/18/2011 17:18:47	424	Textile	manual	931	624	0	84	85	67.02%	
11/18/2011 17:22:24	424	AtomicCms	base	1025	515	0	87	84	50.24%	
11/18/2011 17:26:50	424	AtomicCms	manual	1052	581	0	87	90	55.23%	
11/18/2011 17:28:38	424	Textile	visibility	853	547	0	83	76	64.13%	
11/18/2011 17:37:44	424	Textile	visibility-state	853	517	0	83	76	60.61%	
11/18/2011 17:38:27	424	AtomicCms	visibility	1025	593	0	87	75	57.85%	
11/18/2011 17:44:39	424	AtomicCms	visibility-state	1025	515	0	87	84	50.24%	
11/18/2011 18:03:06	424	KeePassLib	base	7567	3473	0	81	70	45.90%	
11/18/2011 18:19:51	424	KeePassLib	visibility-state	7567	3481	0	81	70	46.00%	
11/19/2011 18:42:57	424	KeePassLib	manual	7627	3751	0	81	81	49.18%	
11/19/2011 18:55:16	424	KeePassLib	visibility	7567	3657	0	81	80	48.33%	
12/2/2011 17:22:36	private	Textile	base	853	517	0	83	78 vs. 82	60.61%	Experimenting with computing the MI of the Pex-generated tests
12/2/2011 17:22:37	private	Textile	manual	931	625	0	84	85 vs. 82	67.13%	
12/2/2011 17:22:37	private	Textile	visibility-state	853	518	0	83	76 vs. 82	60.73%	
12/2/2011 17:22:37	private	Textile	visibility	853	549	0	83	76 vs. 81	64.36%	
12/2/2011 19:35:23	private	AtomicCms	base	1025	515	0	87	84 vs. 78	50.24%	
12/2/2011 19:35:24	private	AtomicCms	manual	1052	581	0	87	90 vs. 78	55.23%	
12/2/2011 19:35:24	private	AtomicCms	visibility-state	1025	518	0	87	84 vs. 78	50.54%	
12/2/2011 19:35:24	private	AtomicCms	visibility	1025	595	0	87	75 vs. 77	58.05%	MI Pex is the one after "vs."

Bibliography

[1] Extract Method, 2005. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/0s21cwxk%28VS.80%29.aspx.

[2] Summary of the Cluster Mission, February 2005. Accessed April 2012 from http://sci.esa.int/science-e/www/object/index.cfm?fobjectid=31258.

[3] 5. Data Structures – Python v2.7.2 documentation, October 2008. Accessed April 2012 from http://docs.python.org/tutorial/datastructures.html.

[4] Deep Zoom File Format Overview, 2008. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/cc645077(VS.95).aspx.

[5] Extract Method, 2008. Accessed April 2012 from http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm.

[6] CA1502: Avoid excessive complexity, 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/ms182212.aspx.

[7] Explicit Interface Implementation (C# Programming Guide), April 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/ms173157.aspx.

[8] internal (C# Reference), April 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/7c5ka91b%28v=VS.100%29.aspx.

[9] InternalsVisibleToAttribute Class (System.Runtime.CompilerServices), April 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.internalsvisibletoattribute.aspx.

[10] ref (C# Reference), April 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/14akc2c7.aspx.

[11] Setting the InternalsVisibleTo Attribute, 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/bb385840.aspx.

[12] Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, page 418, 15 2010. Accessed April 2012 from http://dx.doi.org/10.1109/IEEESTD.2010.5733835.

[13] Tuple Class (System), April 2010. Accessed April 2012 from http://msdn.microsoft.com/en-us/library/system.tuple%28VS.100%29.aspx.

[14] Visual Studio Code Metrics PowerTool 10.0, January 2011. Accessed April 2012 from http://www.microsoft.com/download/en/details.aspx?id=9422.

[15] M. Aberdour. Open source .NET developer testing tools, 2007. Accessed April 2012 from http://opensourcetesting.org/unit_dotnet.php.

[16] A. Baresel, H. Sthamer, and M. Schmidt. Fitness Function Design to Improve Evolutionary Structural Testing. In *Proceedings of the 2002 Conference on Genetic and Evolutionary Computation*, GECCO '02, pages 1329–1336, New York, USA, July 2002.

[17] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[18] K. Beck. Simple Smalltalk testing. *The Smalltalk Report*, 4(2):16–18, October 1994. Accessed April 2012 from http://www.xprogramming.com/testfram.htm.

[19] R. V. Binder. *Testing Object-Oriented Systems*, chapter 19: Test Harness Design, pages 957–1063. Addison-Wesley, 2000.

[20] R. V. Binder. *Testing Object-Oriented Systems*, chapter 10: Classes, pages 347–523. Addison-Wesley, 2000.

[21] R. V. Binder. *Testing Object-Oriented Systems*, chapter 17: Assertions, pages 807–916. Addison-Wesley, 2000.

[22] R. V. Binder. *Testing Object-Oriented Systems*, chapter 4: With the Necessary Changes: Testing and Object-oriented Software, pages 63–110. Addison-Wesley, 2000.

[23] R. V. Binder. *Testing Object-Oriented Systems*, chapter 6: Combinational Models, pages 121–173. Addison-Wesley, 2000.

[24] O. Bühler and J. Wegener. Evolutionary functional testing. *Computers & Operations Research*, 35(10):3144–3160, October 2008. Part Special Issue: Search-based Software Engineering.

[25] L. Chabant. Textile.NET source code at changeset 26030, March 2009. Accessed April 2012 from http://textilenet.codeplex.com/SourceControl/changeset/changes/26030.

[26] C. M. Christensen. Reinventing IT, October 2011. Accessed April 2012 from http://gartner.mediasite.com/mediasite/play/9cfe6bba5c7941e09bee95eb63f769421d?t=1320659595.

[27] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, May 2006.

[28] E. W. Dijkstra. Structured programming. In J. N. Buxton and B. Randell, editors, *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee.* Scientific Affairs Division, NATO, October 1969.

[29] E. W. Dijkstra. On the cruelty of really teaching computing science, December 1988. Accessed April 2012 from http://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html.

[30] M. Feathers. A Set of Unit Testing Rules, September 2005. Accessed April 2012 from http://www.artima.com/weblogs/viewpost.jsp?thread=126923.

[31] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, July 1999. Also available at http://www.refactoring.com/.

[32] M. Fowler. Extract Method, 2000. Accessed April 2012 from http://martinfowler.com/refactoring/catalog/extractMethod.html.

[33] M. Fowler. Inline Temp, 2000. Accessed April 2012 from http://martinfowler.com/refactoring/catalog/inlineTemp.html.

[34] M. Fowler. Introduce Explaining Variable, 2000. Accessed April 2012 from http://martinfowler.com/refactoring/catalog/introduceExplainingVariable.html.

[35] R. P. Gabriel. The Rise of Worse is Better, 1989. Accessed April 2012 from http://www.jwz.org/doc/worse-is-better.html.

[36] E. Gamma and K. Beck. Test Infected: Programers Love Writing Tests, July 1998. Accessed April 2012 from http://junit.sourceforge.net/doc/testinfected/testing.htm.

[37] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification.* Addison-Wesley, third edition, 2005. Accessed April 2012 from http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html.

[38] P. Graham. *On Lisp.* Prentice Hall, 1993.

[39] P. Haahr. A Programming Style for Java, October 1999. Accessed April 2012 from http://www.webcom.com/~haahr/essays/java-style/single-page.html.

[40] M. Harman. Refactoring as Testability Transformation. Keynote paper for Refactoring and Testing Workshop, March 2011. Accessed April 2012 from http://www.cs.ucl.ac.uk/staff/mharman/reftest.pdf.

[41] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[42] M. Harman, L. Hu, R. M. Hierons, A. Baresel, and H. Sthamer. Improving Evolutionary Testing By Flag Removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1359–1366, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[43] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated Test Data Generation for Aspect-Oriented Programs. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development*, AOSD '09, pages 185–196, New York, NY, USA, March 2009. ACM.

[44] M. Harman, M. Munro, L. Hu, and X. Zhang. Side-Effect Removal Transformation. In *In 9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 310–319. IEEE Computer Society Press, 2001.

[45] M. Hevery. Flaw: Brittle Global State & Singletons, November 2008. Accessed April 2012 from http://misko.hevery.com/code-reviewers-guide/flaw-brittle-global-state-singletons/.

[46] M. Hevery. Static Methods are Death to Testability, December 2008. Accessed April 2012 from http://misko.hevery.com/2008/12/15/static-methods-are-death-to-testability/.

[47] M. Hevery. Testability Explorer, April 2008. Accessed April 2012 from http://code.google.com/p/testability-explorer/.

[48] M. Hevery. How to think about OO, July 2009. Accessed April 2012 from http://misko.hevery.com/2009/07/31/how-to-think-about-oo/.

[49] A. C. Kay. The Early History Of Smalltalk. *SIGPLAN Notices*, 28(3):69–95, March 1993.

[50] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[51] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data Dependence Based Testability Transformation in Automated Test Generation. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 245–254, Washington, DC, USA, 2005. IEEE Computer Society.

[52] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83(12):2379–2391, December 2010. TAIC PART 2009 - Testing: Academic & Industrial Conference - Practice And Research Techniques.

[53] N. Leveson. *Safeware: System Safety and Computers*, chapter Appendix A: Medical Devices: The Therac-25, pages 1–49. Addison-Wesley, 1995.

[54] K. J. Lieberherr and I. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6:38–48, September 1989.

[55] J.-L. Lions, L. Lübeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. ARIANE 5 Flight 501 Failure. Report, Ariane 501 Inquiry Board, Paris, July 1996. An HTML version is available at `http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report`.

[56] T. J. McCabe. A Complexity Measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, December 1976.

[57] P. McMinn. Search-based Software Test Data Generation: A Survey. *Journal on Software Testing, Verification, and Reliability*, 14(2):105–156, June 2004.

[58] P. McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield, January 2005.

[59] P. McMinn, D. Binkley, and M. Harman. Empirical Evaluation of a Nesting Testability Transformation for Evolutionary Testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3):11:1–11:27, June 2008.

[60] P. McMinn and M. Holcombe. The State Problem for Evolutionary Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, volume 2724 of *Lecture Notes in Computer Science*, pages 2488–2498. Springer-Verlag, 2003.

[61] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. The Addison-Wesley Signature Series. Addison-Wesley, 2007.

[62] C. Morrison. Maintainability Index Range and Meaning, November 2007. Accessed April 2012 from `http://blogs.msdn.com/b/codeanalysis/archive/2007/11/20/maintainability-index-range-and-meaning.aspx`.

[63] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, August 2008.

[64] R. Osherove. Write Maintainable Unit Tests That Will Save You Time And Tears, January 2006. Accessed April 2012 from http://msdn.microsoft.com/en-us/magazine/cc163665.aspx.

[65] R. Patton. *Software testing.* Safari Books Online. Sams, 2005.

[66] S. H. Peter Van-Roy. *Concepts, techniques, and models of computer programming.* MIT Press, 2004.

[67] K. Pilch-Bisson and J. Bazuzi. Enforcing Immutability in Code, November 2007. Accessed April 2012 from http://blogs.msdn.com/b/kevinpilchbisson/archive/2007/11/20/enforcing-immutability-in-code.aspx.

[68] C. Poole, J. Cansdale, G. Feldman, J. W. Newkirk, A. A. Vorontsov, M. C. Two, and P. A. Craig. NUnit, September 2000. Accessed April 2012 from http://nunit.org/.

[69] U. S. Reddy. Imperative Functional Programming. *ACM Computing Surveys*, 28(2):312–314, June 1996.

[70] J. Reehal. Why and what is Arrange Act Assert?, June 2010. Accessed April 2012 from http://www.arrangeactassert.com/why-and-what-is-arrange-act-assert/.

[71] D. Reichl. KeePass Password Safe source code for version 2.10, March 2010. Accessed April 2012 from http://sourceforge.net/projects/keepass/files/KeePass%202.x/2.10/KeePass-2.10-Source.zip/download.

[72] A. Shapovalov. Atomic CMS 2.0, April 2010. Accessed April 2012 from http://atomiccms.codeplex.com/releases/view/43118.

[73] G. Shaw, I. MacLean, S. Hernandez, and G. Driesen. NAnt - A .NET Build Tool, October 2006. Accessed April 2012 from http://nant.sourceforge.net/.

[74] E. Sink. Advocating the use of code coverage, September 2006. Accessed April 2012 from http://www.ericsink.com/articles/Code_Coverage.html.

[75] J. Spolsky. The Joel Test: 12 Steps to Better Code, August 2000. Accessed April 2012 from http://www.joelonsoftware.com/articles/fog0000000043.html.

[76] S. Stewart. Test Sizes, December 2010. Accessed April 2012 from http://googletesting.blogspot.com/2010/12/test-sizes.html.

[77] H. Sthamer, J. Wegener, and A. Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *In Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis and Review*, AsiaSTAR 2002, pages 22–24, July 2002.

[78] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. In *Proceedings of the 2nd international conference on Tests and proofs*, volume 4966 of *TAP'08*, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[79] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[80] E. VanDoren. Maintainability Index Technique for Measuring Program Maintainability, January 1997. Accessed April 2012 from `http://web.archive.org/web/20080515074252/http://www.sei.cmu.edu/str/descriptions/mitmpm_body.html`.

[81] M. Veanes, C. Campbell, W. Grieskamp, L. Nachmanson, and N. Tillmann. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Verlag, 2008.

[82] L. Vogel. JUnit - Tutorial, 2007. Accessed April 2012 from `http://www.vogella.de/articles/JUnit/article.html`.

[83] P. Waldschmidt. NCover, January 2004. Accessed April 2012 from `http://www.ncover.com/`.

[84] D. Wampler. *Functional Programming for Java Developers.* O'Reilly Media, July 2011.

[85] S. Wappler and J. Wegener. Evolutionary Unit Testing of Object-Oriented Software using Strongly-Typed Genetic Programming. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1925–1932, New York, NY, USA, July 2006. ACM.

[86] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, December 2001.

[87] E. W. Weisstein. Great Circle, 1999. Accessed April 2012 from `http://mathworld.wolfram.com/GreatCircle.html`.