Universiteit Antwerpen
Faculteit Wetenschappen
Departement Wiskunde – Informatica

2005 – 2006

Coevolution of Software and Tests

An Initial Assessment

Joris Van Geet

Proefschrift ingediend met het oog op het behalen van de graad van
licentiaat in de Wetenschappen

Promotor: Prof. dr. Serge Demeyer
Begeleider: Andy Zaidman

Dankwoord

Graag had ik volgende mensen bedankt om me, al dan niet rechtstreeks, te steunen bij het schrijven van deze thesis.

Andy Zaidman om me alles zelf uit te laten zoeken en me daar toch wonderbaarlijk wel in te begeleiden. Professor Serge Demeyer om me reeds vijf jaar te inspireren met zijn interessante lessen en visies. Professor Roel Wuyts voor zijn mateloos en zeer besmettelijk enthousiasme over bijna alles.

Mijn ouders om me de kans(en) te geven mijn eigen weg te zoeken en te gaan, ook al ligt die niet steeds in *aanvaard* terrein.

Al mijn vrienden, vriendinnen, (ex-)kotgenoten en studiegenoten zonder wie ik het waarschijnlijk ook had gekund, maar zeker niet zo comfortabel en aangenaam.

In het bijzonder Koen om een immer aangename projectpartner te zijn, Pieter om me tussen al dit (volgens hem) academische gekakel toch met minstens één voet in de *echte* wereld (en in de sporthal) te houden, Sven voor al zijn intelligente en minder intelligente opmerkingen, Ellen om me de oren van het hoofd te zagen en mijn vriendin Liesbeth om het al die tijd met (en zonder) mij uit te houden.

Veel lees plezier![1]

[1] Zij die verder geen affiniteit met computer wetenschap hebben, kunnen hier gerust stoppen met lezen.

Samenvatting

In een tijd waarin software het dagelijkse leven draaiende houdt, is de kwaliteit ervan een heikel punt. Eén van de technieken om die te bewaren is testen, maar zoals Dijkstra [7] ons laat weten kunnen testen alleen het bestaan van fouten bewijzen, niet de afwezigheid ervan. Een systeem is incorrect als een test een ander resultaat geeft dan verwacht, maar een eindig aantal testen kan nooit garanderen dat het systeem correct is. Toch kan testen bijdragen tot betere software omdat werkende testen het vertrouwen in het systeem verhogen en een goede test wel fouten kan vinden. We onthouden enkel dat testen op zich geen absolute zekerheid biedt.

Door een veelheid aan factoren die terug te reduceren zijn tot een gebrek aan tijd en geld is software meestal niet robuust bij oplevering. De belangrijkste oorzaak hiervan is het gebrek aan testen. Omdat aanpassingen exponentiëel 'duurder' worden naarmate het project vordert, moeten we zo vroeg mogelijk testen, bij voorkeur tijdens – of zelfs voor – het coderen.

Hoewel er al vele testraamwerken zoals xUnit geschreven zijn om testen comfortabeler te maken, ervaren veel programmeurs het nog steeds als een last. Unit tests schrijven duur te lang, de testen laten lopen wordt veelal vergeten en ze up-to-date houden vergt te veel werk. Wij denken echter dat het beter integreren van test- en ontwikkelingsproces veel van deze problemen kan oplossen. Door afhankelijkheden van de test code en de product code expliciet te gebruiken in de ontwikkelingsomgeving kunnen we bijvoorbeeld aanduiden welke tests moeten aangepast worden als er code verandert. Hierdoor wordt de coevolutie van test en product code beter bewaard. Als een eerste stap in de richting van dit integratieproces zullen wij ons bezig houden met het gebruik van die afhankelijkheden om de coevolutie van bestaande projecten onder de loep te nemen.

We bestuderen afhankelijkheden op methode niveau en op basis van dynamische analyse. We zeggen dat een test methode een andere methode test als deze

methode opgeroepen wordt tussen begin en einde van de test methode (al dan niet rechtstreeks door de test methode zelf). Analoog zeggen we dat een methode getest wordt door een test methode als ze tussen begin en einde van de test methode opgeroepen wordt.
Hiervoor gebruiken we dynamische analyse omdat we het feitelijke gedrag van de tests willen bestuderen, niet alle mogelijke uitvoeringsscenario's die we kunnen afleiden uit statische analyse technieken. Merk op dat we niet gaan kijken of het de bedoeling van een test was om een specifieke methode te testen, we kijken enkel waar de test methode overal doorloopt.

Om de coevolutie van test en product code, i.e. de mate waarin test code samen evolueert met product code, op een hoog niveau te bestuderen, stellen we een paar metrieken voor die gebaseerd zijn op de dynamische afhankelijkheden:

- **Het aantal methodes per test methode** duidt het aantal methodes aan die getest worden door een willekeurige methode.
- **Het aantal test methodes per methode** duidt het aantal test methodes aan die een willekeurige methode testen.
- **Het totale aantal test methodes** die uitgevoerd werden.
- **Het totale aantal methodes** die getest werden.
- **Het totale aantal methode oproepen**.

We formuleren volgende hypothese:

> Heuristische metrieken op dynamische afhankelijkheden geven een maat voor de coevolutie van test en product code.

Om dit te valideren maken we eerst een tool om dynamische test afhankelijkheden te extraheren uit Java programma's met werkende test scenario's die gebaseerd zijn op het JUnit testraamwerk. We gebruiken hiervoor de Java Virtual Machine Profiler Interface om de uitvoering van de test scenario's op de voet te volgen. Uit deze informatie halen we dan onze dynamische afhankelijkheden die we opslaan in een XML formaat om makkelijk verder te onderzoeken.

Vervolgens laten we deze tool los op verschillende versies uit twee fasen van het Apache Ant project. Een stabiele fase, vooral opgemaakt uit *bug fixes*, en een minder stabiele fase waar toevoegen van functionaliteit centraal stond. Voor elk van deze versies berekenen we de voorgestelde metrieken en bestuderen we bepaalde eigenaardigheden in meer detail. Om de coevolutie te bestuderen hebben we echter een extra (statisch) gegeven nodig dat we vinden in de test coverage. Op basis van deze informatie concluderen we dat de test code in Apache Ant meestal niet lang na de product code geschreven wordt maar om dit in meer detail te bekijken is verder onderzoek nodig.

We concluderen dat de voorgestelde metrieken goed zijn om de *evolutie* van test code te bestuderen, maar om de *coevolutie* van test en product code te bestuderen hebben we nog iets extra nodig. We hebben dit gevonden in de test coverage dus besluiten we het volgende:

> Heuristische metrieken op dynamische afhankelijkheden *in combinatie met coverage informatie* geven een maat voor de coevolutie van test en product code.

Voor verder onderzoek stellen we voor deze coevolutie in meer detail te bespreken op basis van dezelfde dynamische test afhankelijkheden. Verder zijn we ervan overtuigd dat die ook gebruikt kunnen worden bij het integreren van test afhankelijkheden in de ontwikkelinsomgeving om hiermee de efficiëntie van programeurs en bijgevolg de kwaliteit van software in het algemeen te verbeteren.

Abstract

Unit testing is the first line of defence against software failure. To make the most of this technique the test code should evolve simultaneously with the product code.

First, this dissertation explores the possibilities of using dynamic analysis to extract test dependencies. Then we investigate whether heuristic metrics on these dynamic test dependencies provide a measure for the degree to which the test code evolves with the product code.

As a case study we use Apache Ant and look specifically at two different phases in the history of this open source project.

We conclude that dynamic test dependencies alone do not suffice to provide such a measure and we propose an alternative solution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

'Regression Testing'? What's that?
If it compiles, it is good. If it boots up, it is perfect.
— **Linus Torvalds**

1.1 Software Testing

In this era of increasing ubiquitous computing, the need for quality software is enormous. One of the many techniques that aid in preserving software quality is verification of software by testing. Dijkstra states the goal of testing very clearly by reminding us that:

> "testing can only show the presence of errors, not their absence". [7]

If a test delivers a result that is different than expected, the system is incorrect. However, no finite number of tests can ever guarantee correctness of the system in general.

Fortunately, this does not make testing utterly useless since running tests increases the overall confidence in a system and a good test is still able to find errors. We should just keep in mind that testing on its own cannot provide absolute certainty.

1.2 Testing in Practice

Savoia [20] wrote a small article that situates current testing practices in the context of quality assurance. Due to time and budget constraints, software is often not robust on delivery, with lack of testing as the root cause. As the cost of changes increases exponentially as a project proceeds, testing should be done as early as possible, i.e. during – or even prior to – development. That is why this dissertation will focus on developer testing: it is the *first line of defence* against software degradation. When used consistently, it can – and will – save a lot of grief later on in the project.

Although automatic testing frameworks such as xUnit have been created to provide a more comfortable testing experience, many developers still consider testing a *burden*. Creating unit tests is too time consuming, running them is often forgotten and keeping them up-to-date as product code evolves is too much of a hassle. These are just some of the complaints developers struggle with regarding to testing their own code.

We believe that a tighter integration of the development and the testing process can overcome many of these complaints. Explicitly incorporating dependency information of test and product code into the development environment offers a plethora of possibilities. It could, for example, help indicate which tests to update when changes appear in product code, thereby preserving the coevolution of software and tests.

As a first step towards integrating test dependencies into the development process this dissertation will evaluate the use of test dependency information to *assess* the testing code of an existing software project in terms of coevolution with the product code.

Before we continue, let us agree on terminology.

1.3 Dynamic Test Dependencies

We define a dynamic test dependency as a link between a test method and a 'normal' method, provided by dynamic call information. We say a test method *tests* a method if this method is called within the test method, i.e. called between entry and exit of the test method. Analogously, we say that a method is *tested by*

a test method if the method is called between entry and exit of a corresponding test method.

We rely on dynamic analysis to obtain these dependencies because we are only interested in the *actual behaviour* of the tests. As opposed to product code, test code is designed to have a fixed execution scenario. By using dynamic analysis we can capture this fixed scenario by merely running the tests. This way we avoid the need to reason about all possible execution paths that can be deduced with static analysis. As the behaviour of, for example, a possibly polymorphic call or a method call within a classic conditional, cannot be unambiguously deduced with static analysis.

Please note that we do not attempt to verify whether a test method was meant to really *test* a certain method, we are merely using dependency information based on the calls that where made during test scenarios. Furthermore, as we are focusing our efforts towards dependencies *on a method level*, a better name might have been 'dynamic test *method* dependencies'. However, for brevity, the remainder of this dissertation will use the terms 'dynamic test method dependencies', 'dynamic test dependencies' and 'test dependencies' interchangeably. Unless explicitly stated otherwise, they all denote the concept explained in this section.

1.4 Metrics on Dynamic Test Dependencies

In an attempt to quantify the *overall coevolution* of test code and product code, i.e the degree to which the test base *evolves with* the code base, we propose some heuristic metrics based on the dynamic test dependencies.

- **The number of methods per test** denotes the number of methods a certain test method *tests*[1].
- **The number of tests per method** denotes the number of test methods that *test* a certain method.

By aggregating our test dependencies we propose three more metrics.

- **The total number of test methods** that were executed,

[1]When a method is tested multiple times by the same test method, it is only counted once.

- **The total number of methods** that were tested,
- **The total number of method calls**.

We believe that these simple metrics provide a measure for the overall coevolution of test and product code. We therefore formulate the following hypothesis.

1.5 Hypothesis

> Heuristic metrics on dynamic test dependencies, like the ones stated above, provide a measure for overall coevolution of test and product code.

1.6 Validation

To validate our hypothesis we first create a tool to extract the dynamic test dependencies. Then, we perform a case study on two different phases of the Apache Ant project. To evaluate the coevolution quality within each phase, we extract the dynamic test dependencies for different versions of each phase. Finally, we compare the heuristic metrics on the test dependencies, supported by anecdotal evidence.

1.7 Structure

In chapters 2 and 3 we elaborate on unit testing and dynamic program analysis as they are the key components in this dissertation. The tool we created to obtain the test dependencies is explained in chapter 4. In chapter 5 the experimental setup of our case study is presented in more detail. The results of the case study are presented in chapter 6. In chapter 7 we conclude and give directions for the future.

Chapter 2

Unit Testing

Optimism is an occupational hazard of programming: feedback is the treament.
— **Kent Beck**

2.1 Introduction

Testing comes in many forms and can be classified in various ways. The 'Guide to the Software Engineering Body of Knowledge' (SWEBOK) [1] provides some interesting classifications. One of them is based on the *granularity of testing*:

- **Component/Unit testing** is concerned with verifying functionality of small and (clearly) separable components. Such tests are typically conducted by the developer(s) of the component(s).

- **Integration testing** aims at verifying the interaction between components. Usually these components have already been tested by the previous strategy. Such tests can also be conducted by the developer(s).

- **System testing** tests the system as a whole. This strategy is considered useful for testing non-functional requirements, as the functional requirements should have been tested by the previous two strategies. Such tests are preferably not conducted by the developers of the system.

Another interesting classification is based on the *objective of testing*. It classifies tests in strategies such as acceptance testing, performance testing and conformance testing. Again, a 'complete' listing is provided by the SWEBOK [1], but of particular interest for this dissertation is regression testing.

- **Regression testing** is the 'selective retesting of a system or component to verify that modifications have not caused unintended effects' [13] and it can be conducted at all levels for functional and non-functional testing.

As mentioned in the introduction (section 1.2) we are mainly interested in developer testing. The remainder of this chapter is entirely devoted to unit testing as it is by far the most popular methodology/framework for automated developer testing.

2.2 Initial Philosophy

The initial unit testing strategy, as proposed by Beck [3], was created from a dislike of user interface testing which was deemed "too brittle to be useful" as it was based on user interface scripts. The goal of this new strategy was to set up a simple pattern system for developers to write their own tests in the same language and environment as they were developing in, namely Smalltalk. Beck's main concerns were:

- simplicity,
- full automation,
- preventing interference between tests.

In order to achieve this, he introduced several concepts.

Fixture The main purpose of a fixture is to define the context for a test case, a fixture defines the state of the *unit under test*. In Object Oriented systems this implies an instance variable for each known object in the fixture and a `setUp` method to initialise these variables, thereby representing the test data. Furthermore, a fixture has the ability to be run by a test case whereby the test case implements the behaviour of the test. This implies that several test cases can run the same fixture resulting in testing one context (fixture) in several ways.

Test Case A test case represents a *single unit of testing*, the smallest unit of testing commonly used by testers. The purpose of a test case is to stimulate the fixture to provoke a result and then to check whether that result was what you had predicted. Thus, a test case represents a predictable reaction of a fixture. To prevent interference between tests each test case is represented as an object with its own fixture. In fact, a fixture is initialised with a specific test case.

Failures and Errors When running a test case a distinction should be made between failures and errors. A failure represents a deviation between the specification (as defined by the failing test case) and the behaviour of the system. An error, on the other hand, is a manifestation of a defect causing abnormal behaviour of the system, such as a crash.

Test Suite A test suite is a collection of test cases (with their corresponding fixtures). In the same way as a test case can be run (through a fixture), a test suite can be run resulting in running every test case in sequence and aggregating the results. This similarity enables a test suite not only to contain test cases but also other test suites, thereby creating a fully automated testing framework.

2.3 An Implementation

With the concepts of section 2.2 in mind, figure 2.1 depicts a class diagram of a possible design of the unit testing framework. The `TestFixture` class has a `run` method which invokes the `setup` method to initialise its objects under test, executes the `pluggableTestCase`, fills in the `TestResult` object and invokes the `tearDown` method to clean up for a next run. In this design the test case is interpreted as a method. In Smalltalk, for example, this is possible since everything is modelled as an object. A method is simply an object that *understands* the message `perform`. This way you can easily test several test scenarios on the same fixture by assigning different test cases to the `pluggableTestCase` variable before invoking the `run` method on the fixture. This design pattern is known as the *plugable selector* as explained by Beck [2, p.70].

To be able to use the unit testing methodology in more languages, figure 2.2 depicts a design that overcomes some of the language specific constructs of the

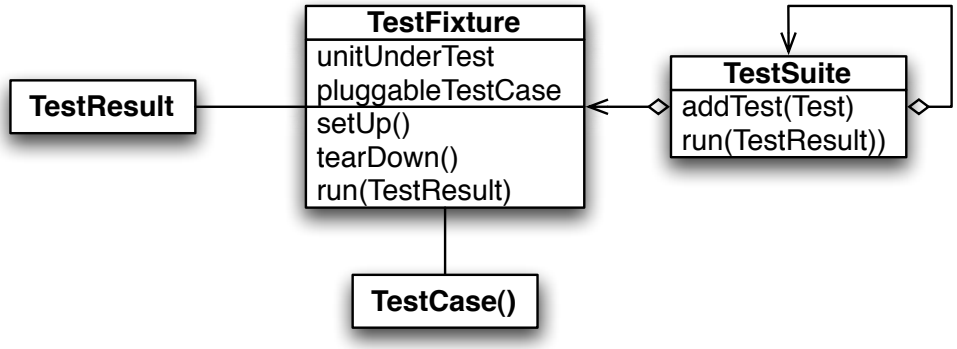


Figure 2.1: Possible Smalltalk design of a unit testing framework modelled as a class diagram in UML.

original Smalltalk implementation. `TestSuite` and `TestFixture` have a common interface so that a test suite cannot only contain individual tests but also other test suites. Another difference is that `TestCase` is a subclass of `TestFixture`. This way the fixture can also be reused for several test cases.
Furthermore, in Java, for example, the plugable selector behaviour can be accomplished by using reflection. Instead of a test case variable, the test class contains all the test cases that test that fixture in the form of public methods. When invoking the `runTest` method on the test class, it searches for all its public (test) methods and runs them in sequence while preserving the `setUp` and `tearDown` calls before and after each test case.

2.4 The xUnit Framework

Currently, unit tests are available in practically all flavours. Frameworks have been written in languages ranging from Smalltalk, C++ and Java over .NET and Visual Basic to JavaScript, PHP and even PL/SQL and MATLAB.[1] All of these xUnit frameworks are based on the philosophy described in section 2.2. In practice most of these implementations have the following minimal set of properties [4, ch. 29].

[1] A more complete listing of xUnit frameworks can be obtained from http://www.xprogramming.com/software.htm

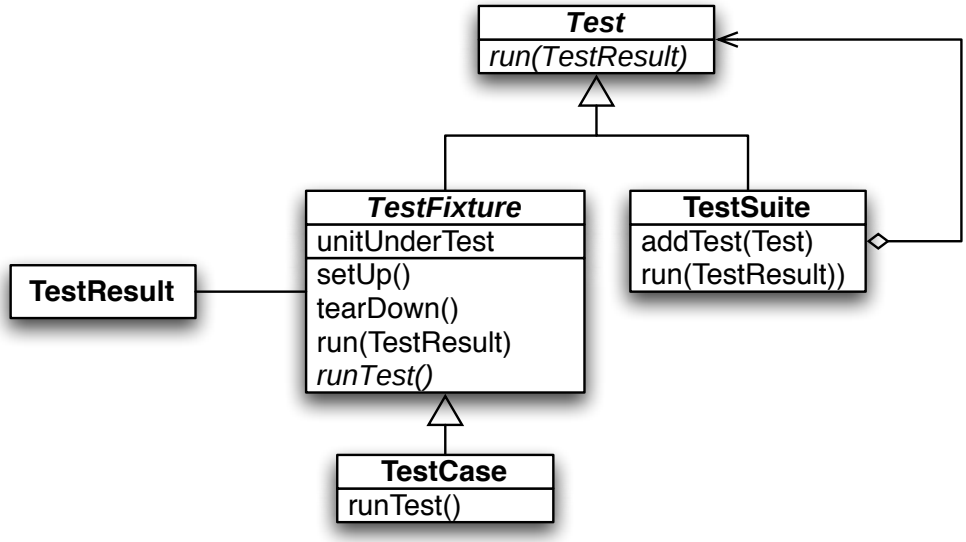


Figure 2.2: Possible Java design of a unit testing framework modelled as a class diagram in UML.

Assertion In order to fully automate a testing framework, the evaluation of the results should be free of human interaction and judgement. This suggests that evaluation of a test case is represented as a boolean decision and that it is performed by the computer. A simple `assert()` method that checks a boolean condition should suffice. Most xUnit frameworks provide several variants of such a method.

Fixture A class, declaring a `setUp()` and `tearDown()` method, which can be sub classed to create a fixture containing test data to be used by several tests. The `setUp()` and `tearDown()` methods are crucial for preventing interference between tests as they guarantee a fresh context for each test (figure 2.3). In most xUnit frameworks this class is called `TestCase` (instead of `TestFixture`).

Test Method As a fixture is typically represented as a class, the test cases are simply represented as methods of that class. By convention, these methods are public, take no parameters and their name starts with "`test`". This results in a number of test methods grouped into a `TestCase` class.

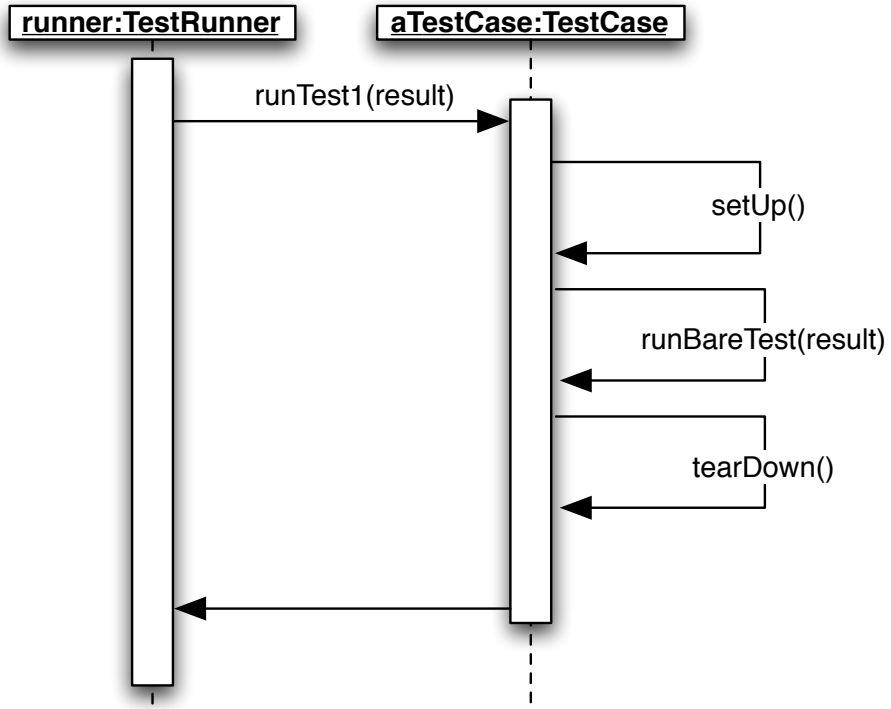


Figure 2.3: Sequence diagram of running one test method.

Test Suite To further enhance test automation, an xUnit framework should provide the possibility to group several `TestCase` classes and run them all at once.

2.5 Unit Testing in Practice

A clear distinction should be made between unit testing as a framework, as described above, and unit testing as a level of testing granularity.

In section 2.1 we stated unit testing and integration testing to act on different levels of granularity. However, the boundary between the two is blurred for object oriented systems as objects are used at all stages of the software process [21]. The key difference is that unit testing should test units *in isolation*. To accomplish this, the objects interacting with the *unit under test* should be replaced by stubs: dummy objects that mimic the behaviour of their real objects for the sole purpose of testing the unit under test.

However, this is not something the xUnit testing framework *imposes on*. The framework merely *facilitates* well structured automated testing. Therefore, it can be – and is – used for more than just unit testing in the strict sense of the word.

2.6 Uses of the Framework

Extreme programming [5] is a refreshing deliberate and disciplined approach to software development based on very simple rules and practices. One of these core practices is *Test Driven Development* [4]. As the name implies, tests are the *driving force* behind this methodology: first write the tests, then write the code to pass the tests, and finally clean up the code without affecting the behaviour. This leads to a very focused way of developing. The xUnit framework is the most commonly used framework for test driven development as it enables to capture requirements in simple test cases and to run them fully automated.

Not only software engineering, but also software *reengineering* benefits from testing frameworks such as xUnit. "Automated tests are the foundation of reengineering" [6] as they provide reproducible and verifiable information. As opposed to written documentation, running tests – and unit tests in particular – are an always up-to-date description of the system. Of course, not all reengineering projects have tests to start with. A testing framework allows you to quickly capture your understanding of (a part of) the system as executable test cases. Furthermore, having a running test suite is essential during refactoring, as you want to make sure the behaviour of the system does not change. Essential here is *limited execution time*, in order to run them after every change, and *error localisation*, to quickly find the problem [10]. Both can be accomplished by using unit testing in the strict sense.

On a slightly different note there is the concept of *daily builds* [22], used in big companies like Microsoft. The idea is to automate the building process and run a clean build on daily basis. When using fully automated testing, this idea can easily be extended to also running all the tests.

Chapter 3

Dynamic Program Analysis

There is nothing either good or bad, but thinking makes it so.
— **William Shakespeare**, "Hamlet"

3.1 Definition

> Dynamic program analysis is the process of investigating a program through its *run-time* information.

As this definition implies, dynamic program analysis is nothing more than executing a program (according to an execution scenario) and observing that execution. Testing, for example, is a form of dynamic analysis where a program is executed according to a certain (test) scenario and the outcome of that execution verified against the expectation. Of course, dynamic analysis allows us to observe more than just the outcome of a certain scenario, we can have access to all executed instructions and intermediate states of the program.

The counterpart of dynamic analysis is static analysis, which is performed without access to run-time information. It usually gathers information from the program code but also from documentation, design, architecture and anything else that does not involve executing the program.

Static Analysis	Dynamic Analysis
Abstract Domain *slow if precise*	Concrete Execution *slow if exhaustive*
Conservative *due to abstraction*	Precise *no approximation*
Sound *due to conservatism*	Unsound *does not generalise*

Table 3.1: Static and Dynamic Analysis Comparison by Ernst and Perkins [8].

3.2 Why Dynamic Analysis?

As Ernst [9] explains, static and dynamic analysis have been viewed as separate domains as they emerged from different communities. They are regarded as fundamentally different techniques each used for their own set of tasks. However, Ernst argues that the difference between these two techniques is not as big as it appears to be. Static analysis examines program code to reason about all possible behaviour that *might* arise at run-time. As there are typically infinite possible executions, feasibility issues force static analysis to abstract away some details thereby losing precision but retaining soundness. Because of the same feasibility issues, dynamic analysis examines execution (traces) for a relatively small set of specific execution scenarios, thereby losing soundness but retaining precision. This way, Ernst uncovers a duality between these two techniques that make many of the same trade-offs. They investigate the same program properties but approach them in a different way. That is why Ernst proposes to acknowledge and exploit the synergy between static and dynamic analysis by *enhancing* each other through pre- and post-processing, *inspiring* each other to develop analogous analysis approaches and even *combining* static and dynamic analysis into a *hybrid* analysis.

In short, static and dynamic analysis can both investigate program properties such as structure, behaviour, performance or quality, but their approach is different (table 3.1), consequently obtaining different yet *complementary results*. However, due to the limited scope of this dissertation, we will be focusing our efforts towards dynamic analysis as we are interested in extracting test dependencies from software projects with readily available test scenarios.

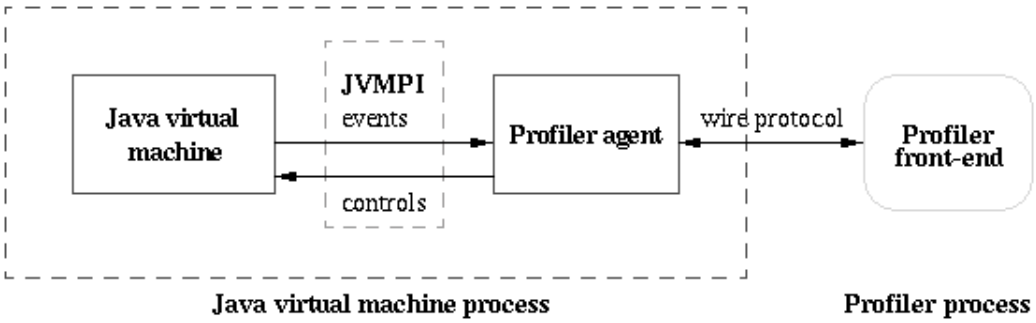


Figure 3.1: An example of virtual machine hooking capabilities: the JVMPI architecture. [25]

3.3 Profiler Driven Program Analysis

Program profiling is a popular subset of dynamic analysis. Although Larus [14] made a clear distinction between program *tracing* and program *profiling*, the term profiling is commonly used for extracting both traces and profiles from running systems. A *trace* is a complete listing of all instructions and data references encountered, whereas a *profile* measures, for example, the frequency of program statements resulting in a statistical summary.

Profilers can use a variety of techniques to collect run-time information. A widely accepted technique is *code instrumentation* which inserts small pieces of code, be it on source code level or byte code level, into the program that is to be analysed. The inserted code then outputs the desired analysis data during execution. ATOM [23], introduced at the PLDI[1] conference of 1994, is a framework for instrumentation based program analysis, effectively turning the program that is to be analysed into its own profiler.
A slightly different approach can be accomplished by using operating system or virtual machine *hooks*. A so called *profiler agent* can be created and hooked into the virtual machine, thereby establishing a two way communication path between the virtual machine and the profiler agent (figure 3.1): the virtual machine can notify the profiler agent of useful events and the profiler agent can in turn request more information.
Another technique for gathering run-time data uses *sampling*. Such a profiler *probes* the internals of the target program at regular intervals using, for example,

[1]The conference on Programming Language Design and Implementation is regarded as the most important conference of the ACM SIGPLAN.

operating system interrupts. Needless to say that this technique is less accurate and less specific, ultimately resulting in statistical approximations of the run-time data.

One of the first profiler implementations was gprof [11], a call graph execution profiler. During execution of a routine it gathers three pieces of information: call counts, execution times per call and the arcs of the dynamic call graph. Afterwards, it combines this information into a dynamic call graph of that particular execution with execution times propagated along the edges. This results in an representation of elapsed time between different calls, which is very useful for performance analysis of routines. More specifically for identifying poorly coded routines, evaluating and optimising new algorithms, etc.

For this dissertation, however, we only use the dynamic call relation, as we are merely interested in call dependencies.

3.4 Dynamic Test Dependencies Revisited

Let us take a look at a visual representation of our dynamic test dependencies as defined in section 1.3. Figure 3.2 illustrates an extract of a dynamic call tree from a toy 'Shopping Cart' example. As you can see, two test methods are called within the execution of the test suite. `testRemoveItem` calls two methods: `removeItem` and `getItemCount`. The `removeItem` method calls `equals` on an object `P` which in turn calls `getTitle` on the same object.

We are interested in *all* the methods that are called after the test method, however, not in the order or the multitude they are called in. This results in a *flattened call tree* for each test method as depicted in figure 3.3(a). As you can see, all the methods that are called appear exactly once.

To inverse this relation, we search for equal methods in the flattened sub trees. As you can see in figure 3.3(a), the `getItemCount` appears in both test methods, thus figure 3.3(b) shows that the `getItemCount` method is tested by both `testRemoveItem` and `testAddItem`.

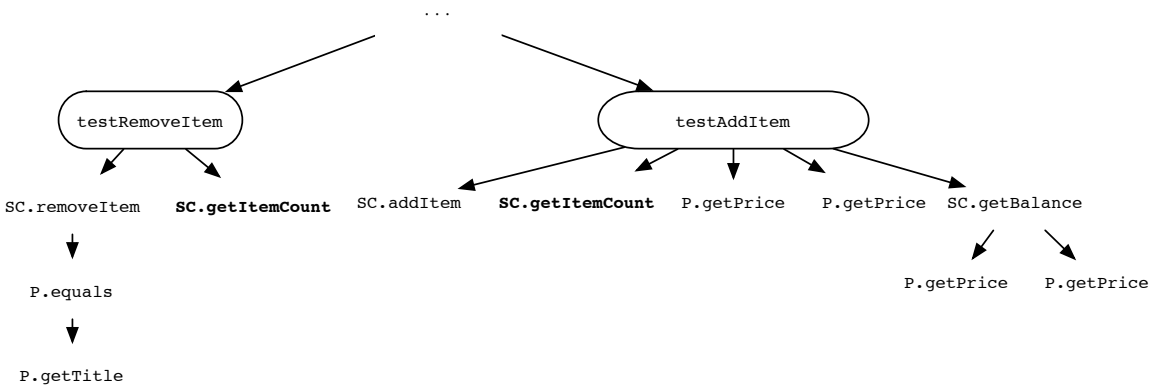
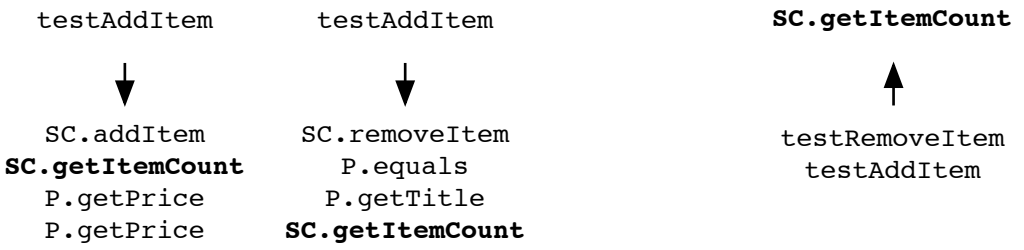


Figure 3.2: Dynamic Call Tree Example



(a) flattened call tree

(b) inverse dependency

Figure 3.3: Dynamic Test Dependencies Example

Chapter 4

Test Dependency Tool

The only place where success comes before work is in the dictionary.
— Donald Kendall

4.1 Introduction

For extracting the test dependencies we created a tool with a pipe and filter architecture. As depicted in figure 4.1 it starts by tracing the execution of the test scenario(s), followed by an analysis of that trace data to eventually result in two xml files that both contain the same test dependency information, albeit in a different form.

4.2 Selective Tracing

To trace the different execution scenarios, we created a *profiler agent* based on the Java Virtual Machine Profiler Interface (JVMPI) [25], which is written in C(++). This agent can be hooked into the Java Virtual Machine, providing a two way communication path (figure 3.1). As the virtual machine sends out various events during execution, our agent *listens* to the `method_entry` and `method_exit` events, which provide the crucial information for a dynamic call graph. Whenever such an entry or exit occurs, some identification information is written to the trace file

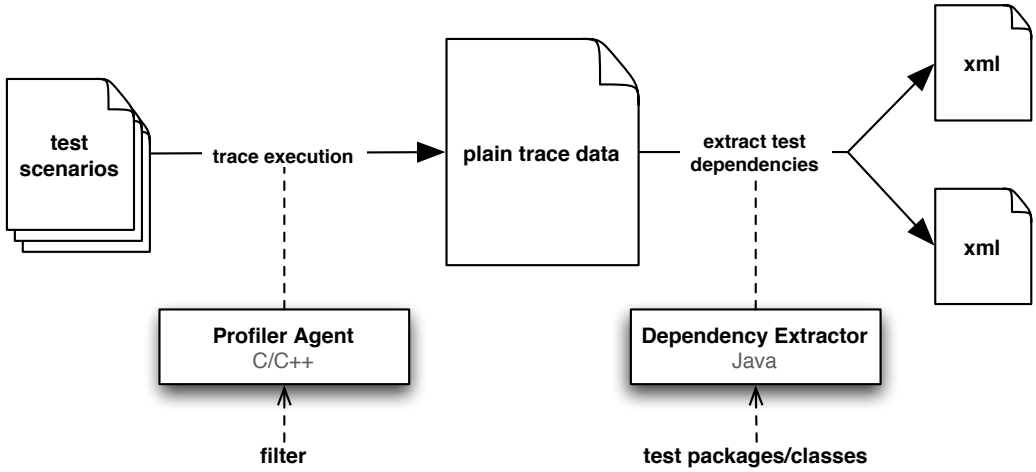


Figure 4.1: Architecture of the dependency tool.

```
(...)
Entry:org.apache.tools.ant.taskdefs.Available:<init>:()V
Entry:org.apache.tools.ant.Task:<init>:()V
Entry:org.apache.tools.ant.ProjectComponent:<init>:()V
Exit:org.apache.tools.ant.ProjectComponent:<init>:()V
Exit:org.apache.tools.ant.Task:<init>:()V
Exit:org.apache.tools.ant.taskdefs.Available:<init>:()V
Entry:org.apache.tools.ant.ProjectComponent:setProject:(Lorg/apache/tools/ant/ Project;)V
Exit:org.apache.tools.ant.ProjectComponent:setProject:(Lorg/apache/tools/ant/Project;)V
Entry:org.apache.tools.ant.Task:setTaskType:(Ljava/lang/String;)V
Exit:org.apache.tools.ant.Task:setTaskType:(Ljava/lang/String;)V
(...)
```

Figure 4.2: Extract from an Ant 1.4.1 trace file.

containing the fully qualified name of the method, its formal parameters and its return type[1]. To give an impression, figure 4.2 shows an extract of a trace we obtained during our research.

Because the virtual machine sends out these events for all methods, including the ones from system classes and third party libraries, we performed a basic form of filtering to only trace packages or classes that are of interest. An inclusion and exclusion filtering mechanism is provided in the form of a simple string based comparison between the filter and the fully qualified class name. This comparison

[1]The return type of a method is not necessary to uniquely identify a method. However, the Java Virtual Machine provides this data together with the parameters, we keep it for human readability.

```
1  (...)
2  Exit:ShoppingCart:addItem:(LProduct;)V
3  Exit:ShoppingCartTest:setUp:()V
4  Entry:ShoppingCartTest:testRemoveItem:()V
5  Entry:ShoppingCart:removeItem:(LProduct;)V
6  Entry:Product:equals:(Ljava/lang/Object;)Z
7  Entry:Product:getTitle:()Ljava/lang/String;
8  Exit:Product:getTitle:()Ljava/lang/String;
9  Exit:Product:equals:(Ljava/lang/Object;)Z
10 Exit:ShoppingCart:removeItem:(LProduct;)V
11 Entry:ShoppingCart:getItemCount:()I
12 Exit:ShoppingCart:getItemCount:()I
13 Exit:ShoppingCartTest:testRemoveItem:()V
14 Entry:ShoppingCartTest:tearDown:()V
15 (...)
```

Figure 4.3: Extract from a Shopping Cart trace file.

simply checks whether the class name starts with the given filter. For example, for the trace in figure 4.2 we used '`org.apache.tools.ant.`' as an inclusion filter, forcing the profiler to only trace the methods that start with the string '`org.apache.tools.ant.`', effectively ignoring all methods that are not part of the ant package (or its subpackages).

Note that at this stage we merely store the trace data for further analysis (offline analysis), instead of analysing the trace data on the fly (online analysis).

4.3 Extracting Test Dependencies

The trace file from the profiler agent provides us with the necessary raw data to extract test dependencies as it lists the entry and exit of all calls in chronological order. The dependency extractor takes a regular expression to identify the test packages or classes. Methods of such a test class are identified as a test method if they take no arguments and their name starts with '`test`', as this is the convention in the JUnit testing framework.

Once we have identified the test methods we can easily deduce all methods that are *tested* by a certain test method, as they appear between entry and exit of that test method. To obtain all the test methods that test a particular method, we inverse this relationship. In practice, we extract both relationships in one pass

```
<?xml version="1.0"?>
<testtomethod>
    <testmethod name="ShoppingCartTest.testRemoveItem">
        <method name="ShoppingCart.removeItem" params="(Product)" return="void" />
        <method name="Product.getTitle" params="()" return="java.lang.String" />
        <method name="ShoppingCart.getItemCount" params="()" return="int" />
        <method name="Product.equals" params="(java.lang.Object)" return="boolean" />
    </testmethod>
    <testmethod name="ShoppingCartTest.testRemoveItemNotInCart">
        <method name="ShoppingCart.removeItem" params="(Product)" return="void" />
        <method name="Product.getTitle" params="()" return="java.lang.String" />
        <method name="Product.equals" params="(java.lang.Object)" return="boolean" />
        <method name="ProductNotFoundException.ProductNotFoundException" params="()" />
        <method name="Product.Product" params="(java.lang.String,double)" />
    </testmethod>
    <testmethod name="ShoppingCartTest.testEmpty">
        <method name="ShoppingCart.empty" params="()" return="void" />
        <method name="ShoppingCart.getItemCount" params="()" return="int" />
    </testmethod>
</testtomethod>
```

Figure 4.4: Shopping Cart xml output for the test-to-method relation.

```
<?xml version="1.0"?>
<methodtotest>
    <method name="ShoppingCart.removeItem" params="(Product)" return="void">
        <testmethod name="ShoppingCartTest.testRemoveItem" />
        <testmethod name="ShoppingCartTest.testRemoveItemNotInCart" />
    </method>
    <method name="ShoppingCart.getItemCount" params="()" return="int">
        <testmethod name="ShoppingCartTest.testRemoveItem" />
        <testmethod name="ShoppingCartTest.testEmpty" />
    </method>
    <method name="Product.Product" params="(java.lang.String,double)">
        <testmethod name="ShoppingCartTest.testRemoveItemNotInCart" />
    </method>
    <method name="ShoppingCart.empty" params="()" return="void">
        <testmethod name="ShoppingCartTest.testEmpty" />
    </method>
    <method name="Product.getTitle" params="()" return="java.lang.String">
        <testmethod name="ShoppingCartTest.testRemoveItem" />
        <testmethod name="ShoppingCartTest.testRemoveItemNotInCart" />
    </method>
    <method name="Product.equals" params="(java.lang.Object)" return="boolean">
        <testmethod name="ShoppingCartTest.testRemoveItem" />
        <testmethod name="ShoppingCartTest.testRemoveItemNotInCart" />
    </method>
    <method name="ProductNotFoundException.ProductNotFoundException" params="()">
        <testmethod name="ShoppingCartTest.testRemoveItemNotInCart" />
    </method>
</methodtotest>
```

Figure 4.5: Shopping Cart xml output for the method-to-test relation.

over the trace file. Finally, we store this information in a proprietary XML[2] format, thereby making the test dependencies explicit in both directions.

Let us look at a small example of this process, based on a toy 'Shopping Cart' program. Figure 4.3 shows an extract of the trace as obtained by executing (and tracing) the unit test. Lines 4 and 13 entry and exit of the `testRemoveItem()` test method. Lines 3 to 8 in figure 4.4 show that the four methods, called between entry and exit of the `testRemoveItem` method (fig 4.3: lines 4–13), are extracted as tested methods for `testRemoveItem()`. In addition, test methods `testRemoveItemNotInCart()` (line 9) and `testEmpty()` (line 16) were executed as well. Figure 4.5 shows the inverse dependency between tests en methods. For example, the method `getItemCount()` (line 7) is *tested* by `testRemoveItem()` (line 8) and `testEmpty()` (line 9). Of course, the same information is visible in figure 4.4: both `testRemoveItem()` and `testEmpty()` test the `getItemCount()` (line 6 and 18 resp.).

Note that, for the test methods, only the name is stored, since a test method never takes any arguments. For the methods, both name and parameter list (and return type) is stored, as they are needed to uniquely identify a method. Furthermore, method calls that appear more than once within the same test method are only listed once, as this tool provides a *flattened call graph* resulting in a *set* of methods for each test method and vice versa.

4.4 Issues

Because of the prototype nature of our tool, we did not add support for multiple threads as the cost/benefit ratio was considered too small. This issue does, however, substantially limit the use of our tool.

When tracing larger execution scenarios[3], the trace data appeared to contain a small amount of *random noise*, enough to crash our dependency extractor. We traced this problem back to either an operating system I/O error or an instability in JVMPI. The latter is the most likely of the two as JVMPI, released with Java 1.1, has always been labelled as an experimental technology. We worked around this issue by simply ignoring the corrupted line(s).

[2]More information of Extensible Markup Language at http://www.w3.org/XML/

[3]We first noticed this when tracing execution scenarios that produced trace files of 7GB.

As with most dynamic analysis techniques scalability and performance is an issue. Obviously, this is also reflected in the use of our tool. If possible, consider limiting research to specific test scenarios as it requires a lot of patience and disk space to trace all test scenarios.

4.5 Room for Improvement

Besides the lack of multi threading support, there are some smaller practical issues that can be improved.

- The filter on the profiler agent is a simple string based comparison. It would be more user friendly to extend this to regular expressions as is the case in the dependency extractor.
- The XML format we use is by no means standard. For querying the dependency information more easily the structure could be changed to explicitly contain package and class names as nodes, instead of implicitly in the fully qualified method name. This way methods are grouped per package and class, which facilitates querying the information.
- When extraction the test dependencies *all* method calls are taken into account, not just the ones that are called directly from the test method. While this can produce interesting information, it can also generate a lot of noise. It might be interesting to introduce a cut-off parameter that defines to what depth the nested calls should be traced (or extracted). This way, the tool is more customisable to fit the user's needs.

4.6 Related Work

An interesting (natural) extension of extreme programming is Saff and Ernst's notion of *continuous testing* [18] [19], which basically continuously runs an xUnit test suite in the background to provide constant feedback. Obviously, running all the tests all the time is not scalable. This is where *test selection and prioritisation* come into play. Rothermel and Harrold [17] provide a nice overview of, both static and dynamic, formal methods for selection and prioritisation of regression

tests, whereas Srivastava and Thiagarajan [24] take this problem to a business environment.

More related to our tool is the combination of dynamic analysis and source code changes as proposed by Wong et al. [29]. They use execution traces from previous runs and current source code changes to prioritise future test runs. Clearly, our tool could provide these test dependencies as well. An even more thorough investigation of *change impact analysis* in general is provided by Ren et al. [16] as they implemented a change impact analysis tool (chianti) based on both dynamic and static information.

Another interesting area is *controlling the size* of a test suite [12] by eliminating *redundant tests*. Our test dependencies can give an indication of this redundancy.

Finally, Winger [28] proposed to use smalltalk pragmas[4] to explicitly link test and methods in order to run the corresponding tests on method compile. Although a very nice idea, applying this to an existing project would require annotating all the methods with their corresponding test(s). Our test dependencies could aid in (semi) automatically annotating such methods. Since our tool is based on JVMPI it can only trace java programs, however, this notion of annotating methods can easily be extended to java. We could, for example, use JavaDoc tags or Java 1.5 annotations to explicitly tag methods and integrate this information with the development environment to run the corresponding tests on compile.

[4] Pragmas provide an annotation mechanism in Smalltalk.

Chapter 5

Experimental Setup

There is no certainty, only opportunity.
— **V**, "V for Vendetta"

5.1 Case Choice

For this study, the choice of a valid software project is restricted in several ways. Some of these restrictions are inherent to the setting of the experiments, others are technical limitations of our tool. This section discusses these requirements for our case study.

5.1.1 Requirements Inherent to the Experiments

A fully automated unit test suite Since we are aiming at uncovering dynamic test dependencies between product code and test code, we need running test scenarios. Although it is not strictly necessary for these scenarios to be automated and structured as unit tests, this is of course the best way to represent reproducible test scenarios in a systematic manner. Furthermore, the test suite should provide *acceptable* product code coverage[1]. By 'acceptable' we mean, at least, existent

[1] Code coverage is a measure describing the degree to which the source code of a program has been tested. This can be measured at different levels. On method level, for example, the code coverage is defined as the number of tested methods divided by the total number of methods in the system.

as we need running test scenarios to extract our dynamic test dependencies[2]. As an indication we would like the code coverage to be no less than 50%, but this is no hard limit.

Access to the source code Based on the dependencies between the product code and the test code, we can extract some speculations and preliminary results. Of course, we would like to verify these findings in the source code, therefore we need unrestricted access to it.

Access to the history As we are investigating the (co)evolution of software and tests, we obviously need different *snapshots* of the system. The easiest way to obtain these is via a (publicly accessible) code repository.

Evolution Accessing the history of a project is only possible if the project *has* a history. To validate our hypothesis we need to investigate a project that has significantly evolved over time. Ideally we would like to be able to investigate two different phases in the project:

1. a phase of *relative stability* where the main focus would be on bug fixes and the like.
2. a phase of *relative instability*, perhaps caused by the addition of a substantial amount of functionality or by a rather intrusive restructuring.

5.1.2 Requirements Inherent to Tool Restrictions

Since our tool is built on top of the JVMPI we are, of course, restricted to *Java* applications. Furthermore, our prototype tool restricts us to *single threaded* applications (see chapter 4) .

[2]Of course, 'no test dependencies' provide results as well, albeit to trivial for our purposes.

class, %	method, %	block, %	line, %
80% (515/640)	65% (3739/5745)	59% (81558/137669)	60% (18967.4/31489)

Table 5.1: Ant 1.6.5's overall test coverage as generated by Emma.

5.1.3 The Selected Candidate

Apache Ant[3] meets all of these requirements as it is a single threaded, java-based build tool which has been released to the public as an open source project in 2001. It is supported by an active development community and has (almost) flawlessly running unit test suites. With an overall method coverage[4] of 65% (see table 5.1), it will most likely provide us with sufficient research opportunities. Table 5.2 gives an overview of the coverage distributed over the different packages, showing that most packages have an even higher method coverage. That is, of course, because other packages have no coverage at all. The `listener` package, for example, implements some extensions on logging frameworks such as Log4J. The `loader` package, which seems very important by name, just contains one deprecated empty `AntClassLoader` class, the actual `AntClassLoader` is located in the top `ant` package. The `mail` package is not even part of the actual Ant project, however, the Ant `email` task does rely on it. But, whatever the reason for these untested packages, they are of little importance to our further investigations as they will not even show up in our test dependencies.

5.2 Distinguishing Two Phases

As mentioned in section 5.1.1, one of the requirements for our case study was the presence of two different phases of evolution. Hence, we conducted two experiments, one for each phase. Each of these experiments involved several subsequent snapshots of a particular development phase.

For our first experiment we investigated the three most recent stable releases of Ant, namely versions 1.6.3, 1.6.4 and 1.6.5. By doing so, we examined a relatively stable phase in the history of Ant's development, as those releases mainly consisted

[3] More information about Apache Ant in appendix A and at http://ant.apache.org/.

[4] This coverage was calculated by the Java code coverage tool Emma, more information at http://emma.sourceforge.net/.

name	class, %	method, %	block, %	line, %
org.apache.tools.ant.listener	0% (0/2)	0% (0/10)	0% (0/705)	0% (0/161)
org.apache.tools.ant.loader	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/1)
org.apache.tools.ant.types.mappers	100% (1/1)	0% (0/4)	0% (0/70)	0% (0/19)
org.apache.tools.mail	0% (0/4)	0% (0/42)	0% (0/749)	0% (0/192)
org.apache.tools.ant.taskdefs.cvslib	27% (3/11)	1% (1/83)	0% (7/2055)	0% (2/522)
org.apache.tools.ant.taskdefs.email	38% (3/8)	9% (7/80)	19% (339/1772)	20% (81.3/407)
org.apache.tools.ant.taskdefs.compilers	33% (4/12)	36% (21/58)	23% (615/2678)	23% (128.1/560)
org.apache.tools.ant.input	100% (4/4)	73% (11/15)	41% (108/263)	45% (34/75)
org.apache.tools.ant.helper	53% (10/19)	61% (63/104)	46% (1381/2975)	52% (349.6/669)
org.apache.tools.ant.util.regexp	100% (3/3)	75% (9/12)	49% (88/179)	49% (25/51)
org.apache.tools.ant.taskdefs.rmic	86% (6/7)	73% (24/33)	52% (640/1242)	50% (110.9/222)
org.apache.tools.ant.taskdefs	76% (169/222)	57% (1254/2203)	55% (29318/53560)	55% (6698.7/12099)
org.apache.tools.ant.types.resources	91% (20/22)	71% (142/201)	56% (1897/3406)	59% (499.7/853)
org.apache.tools.tar	100% (5/5)	54% (51/94)	58% (1461/2532)	63% (350.4/555)
org.apache.tools.ant.dispatch	100% (2/2)	83% (5/6)	61% (174/285)	77% (47/61)
org.apache.tools.ant	85% (45/53)	75% (510/683)	62% (11626/18678)	63% (2647/4169)
org.apache.tools.ant.types.resources.selectors	100% (13/13)	70% (43/61)	62% (439/705)	67% (116.4/175)
org.apache.tools.ant.types.selectors	86% (25/29)	64% (135/210)	65% (2880/4430)	65% (734.3/1124)
org.apache.tools.ant.util	80% (41/51)	70% (246/353)	65% (4645/7092)	66% (1112.7/1682)
org.apache.tools.ant.types	93% (52/56)	82% (520/635)	69% (8454/12195)	71% (2075.6/2908)
org.apache.tools.ant.types.resources.comparators	100% (8/8)	85% (17/20)	72% (110/152)	69% (24.3/35)
org.apache.tools.zip	92% (11/12)	68% (96/141)	73% (2475/3377)	74% (551.7/750)
org.apache.tools.ant.taskdefs.condition	95% (20/21)	88% (100/114)	74% (1322/1797)	77% (326.8/427)
org.apache.tools.ant.taskdefs.repository	100% (15/15)	79% (117/148)	76% (1511/1994)	81% (406.8/501)
org.apache.tools.ant.filters	90% (38/42)	84% (218/259)	76% (3785/4971)	77% (990.9/1291)
org.apache.tools.ant.types.selectors.modifiedselector	100% (9/9)	83% (62/75)	77% (1163/1519)	76% (301.4/394)
org.apache.tools.ant.filters.util	100% (1/1)	89% (8/9)	78% (247/316)	82% (66/80)
org.apache.tools.bzip2	100% (5/5)	85% (67/79)	86% (6747/7843)	85% (1254.8/1474)
org.apache.tools.ant.util.facade	100% (2/2)	100% (12/12)	100% (126/126)	100% (32/32)

Table 5.2: Ant 1.6.5's test coverage per package as generated by Emma.

of bug fixes[5].

Based on *Ant's release notes*[6], we uncovered the transition from Ant 1.4.1 to Ant 1.5 to be a period of tremendous development activity. To investigate this phase more closely, we checked out four versions from the Ant repository: 1.4.1, 1.5 and two intermediate revisions. To define these intermediate revisions we took a presumably more intelligent approach than simply partitioning the transition in three equidistant parts based on time stamps or revision numbers. We checked out the repository history and filtered out

- all changes on files other than `.java` files and
- some non intrusive changes, based on keywords in the commit messages, such as 'java doc' and 'check style'.

We then partitioned this set of changes into three equidistant parts and checked out the revisions responsible for the changes at the two points of separation. By doing

[5]Information about each release can be found at http://ant.apache.org/antnews.html.

[6]The significant list of changes from Ant 1.4.1 to Ant 1.5 can be found at http://archive.apache.org/dist/ant/binaries/RELEASE-NOTES-1.5.html#section-4..

(a) First Experiment

Ant release	revision number
1.6.3	278192
1.6.4	278297
1.6.5	278395

(b) Second Experiment

	Ant release	revision number
v1	1.4.1	269774
v2	n/a	271314
v3	n/a	272441
v4	1.5	273395

Table 5.3: Listing of all revisions used in our experiments.

this, we hope to have partitioned the development efforts during the transition from Ant 1.4.1 to Ant 1.5 in a more balanced way, thereby decreasing the chance of investigating two subsequent versions with little or no evolution in between.

5.3 Obtaining Test Dependencies in Four Steps

The core information for our case studies are the dynamic test dependencies between product code methods and their unit test methods. Obtaining this information for a specific Ant version implies running that version, tracing it and extracting the dependencies. In what follows we will explain these steps in more technical detail.

5.3.1 Obtaining a Version

The first thing you need is a specific version of Ant that you would like to trace. Because of the necessity of different *intermediate* builds, the code repository is the most appropriate source for these versions. We checked out the project from the Ant subversion repository, located at `http://svn.apache.org/repos/asf/ant/core/trunk`, using the `svn co` command. For each version, we obtained the corresponding *revision number* and updated our version using the `svn up -r[revision]` command. Table 5.3 lists all the revision numbers used in our case studies.

5.3.2 Bootstrapping a Version

After obtaining the source code, you need to build the version and check whether the tests run in an acceptable manner. The biggest problem here is how to separate the Ant installation on your system from the one you are trying to run. Luckily, the Ant distribution comes with a bootstrap script. The original purpose of this script is to be able to build this version of Ant without needing a working version of Ant. It builds a basic version of Ant, which is used to build the complete Ant version. Since this version is completely separated from the rest of your system, you can use it to run the tests, thus not interfering with the currently installed version. This is important because you do not want to test (or trace) your current installation of Ant.

Since Ant does not come with a standard JUnit distribution, you should place a copy of it in the `lib` folder of the bootstrapped version before running the tests.

5.3.3 Tracing a Version

The two main issues with tracing a version of Ant are hooking the profiler agent into the virtual machine and making sure you only trace the tests, since you also need to run Ant to start the tests. For this purpose we created a separate Ant build file. As you can see in figure 5.1 we call the `run-tests` task (line 3) of the original Ant build file and set the `ANT_HOME` environment to the bootstrap folder (line 8). The `jvm.arg` argument (line 4) hooks the profiler agent into the virtual machine used to run the tests. The `junit.fork` (line 6) and `test.haltonfailure` (line 5) arguments make sure all tests are run using the same virtual machine and that the trace process does not stop if one of the tests should fail.

Furthermore, the profiler agent needs some properties to filter out relevant information from the execution. We do this by passing on `org.apache.tools.ant` as only element in the inclusion list, located in the `tracefile.in` file. This forces the profiler to only trace the Ant classes.

5.3.4 Extracting Dependencies from a Trace

Now that we have the trace of our entire test suite, we can extract the dependencies using our 'dependency extractor'. It only takes two arguments: the trace as

```
<target name="trace-all-tests">
    <exec executable="ant">
        <arg value="run-tests"/>
        <arg value="-Djvm.arg=-Xrunthesisprofiler"/>
        <arg value="-Dtest.haltonfailure=no"/>
        <arg value="-Djunit.fork=true"/>
        <env key="CLASSPATH" value=""/>
        <env key="ANT_HOME" value="./bootstrap"/>
    </exec>
</target>
```

Figure 5.1: Trace task extracted from our own build file.

generated by the profiler and a regular expression denoting the test classes. We cannot just use package names, since the tests are in the same package as the source code they test. Therefore, we use `.*Test` as regular expression, effectively extracting all classes that end with `Test`. This is a frequently used naming convention that also applies to the Ant project.[7]

5.4 Querying Test Dependencies

As our test dependencies are stored in XML format, we can easily query them with a combination of XSLT[8] and XPath[9]. XPath provides us with a convenient way to navigate through XML files, whereas XSLT presents us some general purpose functions, such as counting nodes and simple string comparisons.

Querying for Metrics

Let us give some examples of XPath queries by explaining how we can calculate the metrics as proposed in section 1.4

[7] We verified this by listing all the classes that subclass `TestCase`. There were some test classes that did not adhere to this convention, however, closer investigation revealed that these tests were not run, thus of no importance to us.

[8] More information on Extensible Stylesheet Language Transformations at http://www.w3.org/TR/xsl/.

[9] More information on XML Path Language at http://www.w3.org/TR/xpath.

- **The number of methods per test** is easily deducible from the XML files we obtained from the dependency tool. Figure 4.4 showed us an XML file that presents us with the *test-to-method* information. All we need to do is find the test method and count its children. For example, the following XPath query returns the number of methods tested by the `testEmpty` in figure 4.4.

```
count(//testmethod[@name=ShoppingCartTest.testEmpty]/method)
```

- **The number of tests per method** is calculated in a similar way, but on the *method-to-test* XML file, which holds the inverse dependencies of the method-to-test file. For example, the following XPath query returns the number of test methods that test the `ShoppingCart.getItemCount` method in figure 4.5.

```
count(//method[@name=ShoppingCartTest.getItemCount]/testmethod)
```

  If there was more than one `getItemCount` method, the `@params` attribute should be used as well to locate the correct method.

- **The total number of test methods** that were executed can be calculated by simple counting all the `testmethod` nodes in the test-to-method dependencies.

```
count(//testmethod)
```

- **The total number of methods** that were tested can be calculated similarly on the method-to-test dependencies.

```
count(//method)
```

- **The total number of method calls** can be calculated by looping over all test methods, and for each test method counting the number of methods they test. In XPath, this can easily be accomplished by executing, for example, the following query on the test-to-method dependencies.

```
count(//testmethod/method)
```

Since the two dependency files contain essentially the same information, we can extract all this information from both the test-to-method and the method-to-test dependencies. Of course, we use the one that is most convenient in a certain situation.

Querying for Specifics

To query the dependencies for more specific information, similar queries can be used. The following query, for example, returns all the methods that are tested by more than 100 test methods.

```
//method[count(testmethod)>100]
```

As a last example, this query returns all methods in the `resource` package.

```
//method[starts-with(@name,'org.apache.tools.ant.types.resources.')]
```

Chapter 6

Results

People often say that motivation doesn't last.
Well, neither does bathing — that's why we recommend it daily.
— Zig Ziglar

6.1 Introduction

As mentioned in section 5.1.1, one of the requirements for our case study was the presence of two different phases of evolution. Hence, we conducted two experiments, one for each phase.

This chapter presents the results for both experiments, each starting with statistical observations and gradually zooming in on anecdotal evidence. Based on these results, the last section of this chapter discusses the similarities and differences between both experiments in terms of coevolution.

6.2 First Experiment

6.2.1 Statistical Indications

First we calculated the the *number of methods* that were tested, the *number of test methods* executed and the *total number of method calls* executed as explained

ant version	methods	tests	calls
1.6.3	4467	1330	286499
1.6.4	4472	1337	288363
1.6.5	4767	1407	324250

Table 6.1: First Experiment: Total count of methods, tests methods and method calls.

(a) Average number of test methods for an arbitrary method.

version	mean	σ
1.6.3	68.02	190.35
1.6.4	64.48	183.49
1.6.5	64.14	182.40

(b) Average number of methods that an arbitrary test method runs through.

version	mean	σ
1.6.3	230.45	146.10
1.6.4	215.68	136.68
1.6.5	215.41	137.01

Table 6.2: First Experiment: Averages of methods and test methods.

in section 5.4. The results are shown in table 6.1. Based on this information we calculated the average number of test methods that test an arbitrary method (table 6.1(a)) and the average number of methods an arbitrary test method runs through (table 6.1(b)). We can see that, on average, a method is tested by approximately 64 test methods and a test method tests approximately 215 methods. These numbers are shocking in contrast with the *ideal* one to one relation between method and test method. However, the enormous standard deviation[1] of the averages we calculated, suggests that the actual values are highly variable, indicating that further investigation is needed.

To get a better view on the distribution of tested methods and test methods, we represented them in a box plot[2] which uses more robust measurements such as the *median* and other *quartiles* instead of the unstable mean.
Figure 6.1 illustrates the distribution of the number of methods that are tested by a test method. For version 1.6.5, for example, you can see that half of the test methods test more (and the other half tests less) than 212 methods, since 212 is

[1] According to Wikipedia [27] standard deviation is the most common measure of *statistical dispersion*. Simply put, standard deviation measures how *spread out* the values in a data set are. Traditionally this measure is represented as σ.

[2] Please note that we use a stripped version of the traditional box plot [26]. Whereas a standard box plot has a parametrised *acceptable range* to define what an outlier is (usually 3/2 times the *inter quartile range*), our range simply extends to the minimum and the maximum values, thus not explicitly specifying outliers.

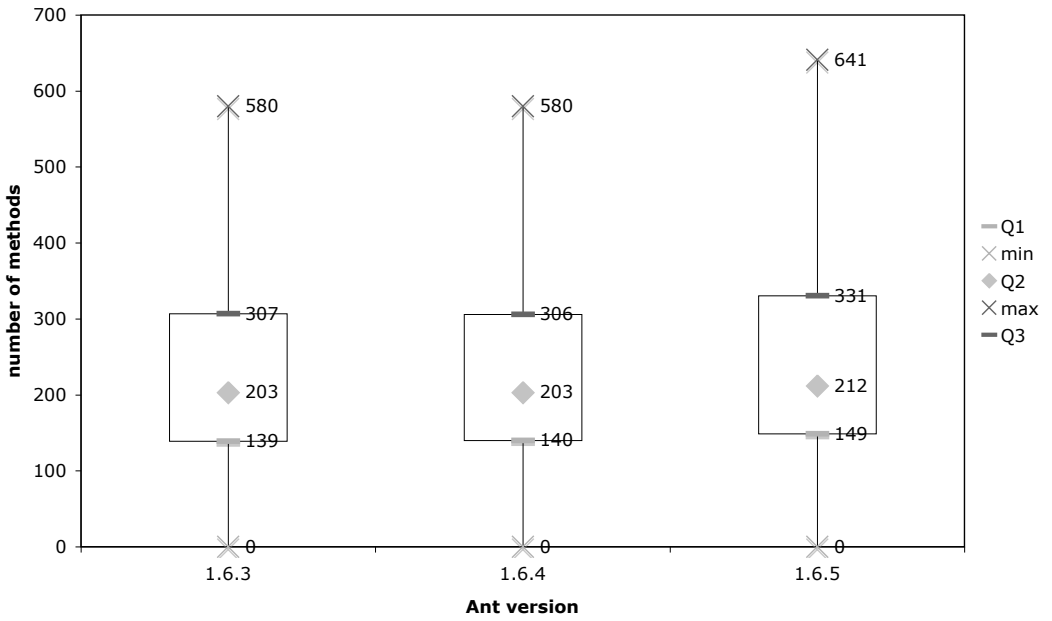


Figure 6.1: First Experiment: Box plot of the distribution of the number of methods tested by an arbitrary test method.

the median (= the second quartile Q2). For the same version you can see that half of the test methods test no more than 331 (the third quartile) and no less than 149 (the first quartile) methods. The distribution is almost symmetric around the median, leaving us with similar results as the ones we obtained from the averages. However, for the number of test methods per method, we do get a different perspective on the distribution. In the box plot of figure 6.2, we used a logarithmic scale on the Y axis to visualise the results in a sensible way. This means that, although the box plot looks almost as symmetric as the one in figure 6.1, it is most definitely not. For version 1.6.5, for example, half of the methods are tested by no more than seven test methods. A quarter of the methods are even tested by no more than two test methods. As opposed to the average of 64, we can see that 75% of all methods are tested by no more than 38 test methods.
This box plot also illustrates how the average can be so high when most tests are far below it. Taking into account the logarithmic scale of the Y axis, the tail of the box plot is enormous. There are outlier methods that are tested by more than 1200 test methods. The average is very sensitive to such outliers, which explains why few methods (less than 25%) can have such an impact on the average.

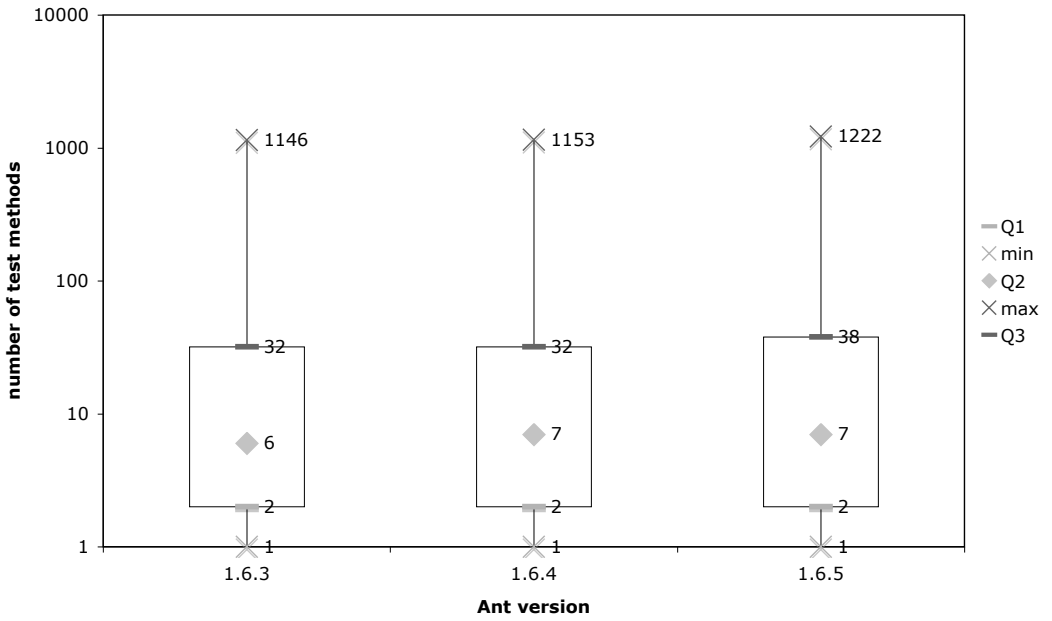


Figure 6.2: First Experiment: Box plot of the distribution of the number of test methods that test an arbitrary method (logarithmic scale).

6.2.2 Anecdotal Evidence

Let us dig a little deeper, based on these statistical observations. In table 6.1 we can see that Ant 1.6.5 runs approximately 1400 test methods and the box plot in figure 6.2 shows us at least one method that is tested by *more than 1200* of these test methods. Based on the rather high average (table 6.1(b)) of methods that an arbitrary test method runs through, we suspect even more of these methods that are tested by almost all test methods. Two possible explanations come to mind.

1. Some form of *generic setup code* is executed at every run. This code would have to be located in the test methods themselves[3], since the dependencies of the `setUp()` and `tearDown()` methods are not extracted from the original trace.

2. Some form of *generic test code* provides an execution scenario that is similar for all tests. This would indicate an integration testing strategy or at least a lack of stub usage.

[3]Setting up test data in the test method is not uncommon as it is the only way to initialise different test data for test methods in the same class.

```
public class RenameTest extends BuildFileTest {
    public void setUp() {
        configureProject("src/etc/testcases/taskdefs/rename.xml");
    }
    public void test1() {
        expectBuildException("test1", "required argument missing");
    }
    public void test2() {
        expectBuildException("test2", "required argument missing");
    }
    public void test3() {
        expectBuildException("test3", "required argument missing");
    }
    public void test4() {
        expectBuildException("test4", "source and destination the same");
    }
    public void test5() {
        executeTarget("test5");
    }
    public void test6() {
        executeTarget("test6");
    }
}
```

JUnit Test Case

```
<project name="xxx-test" basedir="." default="test1">
    <target name="test1">
        <rename/>
    </target>
    <target name="test2">
        <rename src=""/>
    </target>
    <target name="test3">
        <rename dest=""/>
    </target>
    <target name="test4">
        <rename src="testdir" dest="testdir"/>
    </target>
    <target name="test5">
        <rename src="template.xml" dest="."/>
    </target>
    <target name="test6">
        <rename src="template.xml" dest="template.tmp"/>
        <rename src="template.tmp" dest="template.xml"/>
    </target>
</project>
```

Test Build File 'rename.xml'

Figure 6.3: Test structure for the Rename Task

```
1 /** this test fails but we ignore the return value;
2  *  we verify that failure only matters when failonerror is set
3  */
4 public void testRunFail() {
5     if(runFatalTests) {
6         executeTarget("testRunFail");
7     }
8 }
```

Figure 6.4: An untested test as extracted from the `JavaTest` class.

Further investigation of the source code revealed the latter option to be true. We queried our dependencies for classes containing those *often called* methods and briefly navigated through the source code with a code browser. Figure 6.3 nicely illustrates our findings. The top of figure 6.3 is a source code extract of the unit test of the Ant `rename` task. As you can see in line 1, this test extends `BuildFileTest`, an abstraction of a unit test that uses a build file as test data. Line 3 shows that, for each test run, a project is configured based on `rename.xml` (bottom of figure 6.3), a build file specifically designed for testing the `rename` task. As you can see, each test method has its own *target* in the build file. Executing a test method is nothing more than calling its corresponding target on the newly created project and checking whether or not the task produces the correct exception. When searching for that specific build file, we found similar build files for almost all other tests. This indicates tight coupling[4] between different modules in the system, thereby endangering future coevolution of code and tests.

An interesting observation to be made from our previous findings, is that the methods of the abstract `BuildFileTest` class appear as methods called by several test methods, not as test methods themselves as the class name might suggest. This is of course obvious as the `BuildFileTest` is a helper class rather than a test class. However, in some contexts of extracting test dependencies this would be considered noise to be filtered out, whereas for our study it proved valuable information.

On a final remark, you might have noticed that the minima in box plot 6.1 are `0`, whereas the minima in box plot 6.2 are `1`. This is not due to the logarithmic scale in the latter. The `1`s in figure 6.2 represent methods that are tested by exactly one test method. It is clear that this has to be the minimum, since methods that

[4] Coupling is a measure of interconnection among modules in a software structure [15].

are not tested by a test method do not show up in our dependencies. The 0s in figure 6.1, however, represent test methods that call no other methods, or at least no methods that are part of the Ant distribution[5], indicating that the test method does not test Ant functionality. However, closer investigation reveals a test method designed to fail. As explained in the comments of figure 6.4, this test methods only fails when a certain variable is set when running the test suite. Clearly, we did not set that variable, resulting in the test method being run but not going past the initial check (line 5), thus not calling any method.

6.3 Second Experiment

6.3.1 Statistical Indications

For the second experiment, which is focussed on a more active development phase in the Ant project life-cycle, we gathered the same kind of statistical information as for our first experiment. Table 6.3 shows that the test base, and with it the amount of unique methods that are tested, grows consistently with each version. This suggests that newly created test methods test priorly untested functionality. A visual representation of the same results (figure 6.5) reveals an almost straight line for the number of test methods. This shows that our approach for partitioning this phase into different versions, has led to well balanced parts with regard to the relative increase of the test base size. Furthermore, figure 6.5(b) shows that the number of tested methods increases more rapidly than the number of test methods. Since their relation is still linear, this merely indicates that the trend of testing more than one method with one test method remains over time. However, figure 6.5(a) shows that the number of method calls increases in a more exponential manner, indicating that, on average, a test method runs through *more* methods with each subsequent version. The increasing averages in table 6.3(a) and table 6.3(b) support this observation. Although all this might just indicate a few test methods testing a massive amount of methods while most test methods only test one or two methods, the fact that the increase of all three counts is so well balanced over time suggests otherwise. When taking the observations of our first experiment into account as well, these numbers could indicate that the

[5]It is possible that such a test method calls methods that are not part of the Ant distribution, as they are not traced and thus not represented in our test dependencies.

ant version	methods	tests	calls
v1	819	206	13713
v2	1192	340	24851
v3	1623	435	34616
v4	2050	564	51663

Table 6.3: Second Experiment: Total count of methods, tests methods and method calls.

(a) Average number of test methods for an arbitrary method.

version	mean	σ
v1	16.74	35.13
v2	20.85	51.65
v3	21.33	57.89
v4	25.20	69.01

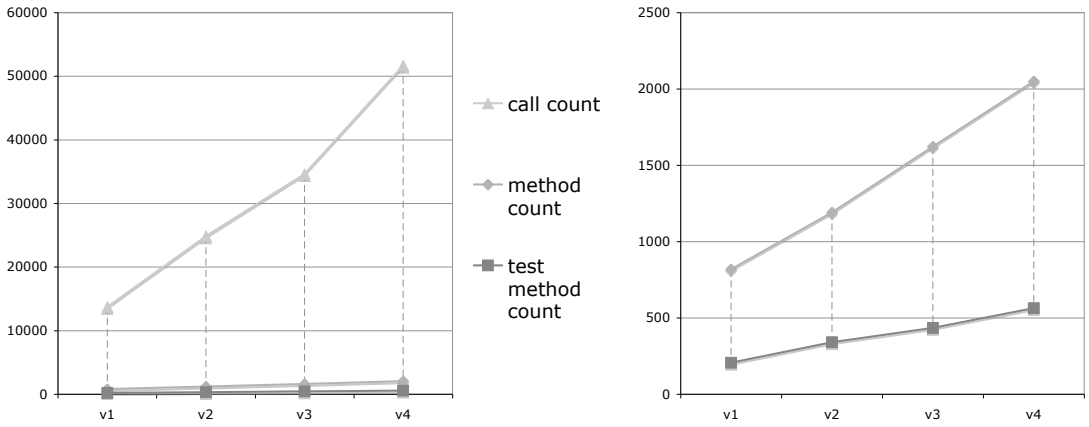
(b) Average number of methods that an arbitrary test method runs through.

version	mean	σ
v1	66.57	44.50
v2	73.09	46.70
v3	79.58	57.97
v4	91.60	73.29

Table 6.4: Second Experiment: Averages of methods and test methods.

integration testing strategy is gaining popularity in the evolution of these earlier versions.

Although less explicitly, box plot 6.6 seems to support this theory as well. Since `Q2` and `Q3` increase with each version (while `Q1` remains roughly the same) the box plot shows that at least 50% of the test methods run through an increasing amount of methods. Box plot 6.7, on the other hand, seems to undermine our theory because the number of tests that run through 75% of the methods stays roughly the same. However, this does not necessarily indicate a contradiction. For example, adding a test class with the purpose of testing new functionality will most likely be done in a similar way as previous test classes. The methods that provide this new functionality will be tested by 2 to 13 test methods of that newly created test class. The test count for these *new functionality methods* will not increase by adding test classes for other functionality, only the test count for the *general framework methods* will increase. However, the statistical importance of these framework methods decreases as new functionality is tested. As a consequence, the only visible effect on the box plot is the increasing maximum.



(a) including method calls

(b) zooming in on methods and test methods

Figure 6.5: Second Experiment: Evolution of total number of unique methods, test methods and method calls.

	method coverage	
ant version	percentage	bare count
v1	32%	1052/3298
v2	42%	1477/3510
v3	35%	1825/5729
v4	38%	2055/5369

Table 6.5: Second Experiment: Method coverage as generated by Emma.

6.3.2 Anecdotal Evidence

Based on the statistical information, we have been able to detect a steady increase of the test base size. However, we cannot make any assumptions on the size of the code base. It is possible that the first version already contained all the Ant functionality and that the subsequent versions only consisted of gradually testing more of this already existent functionality. To investigate this more thoroughly, we calculated the test coverage for each version. If this coverage would remain (roughly) the same for each version, we could conclude that extra functionality was added because the only way that increasing the number of test methods (that test previously untested methods) does not result in an increased test coverage, is by adding even more functionality.

The percentages in table 6.5 suggest that the transition from `v1` to `v2` introduced

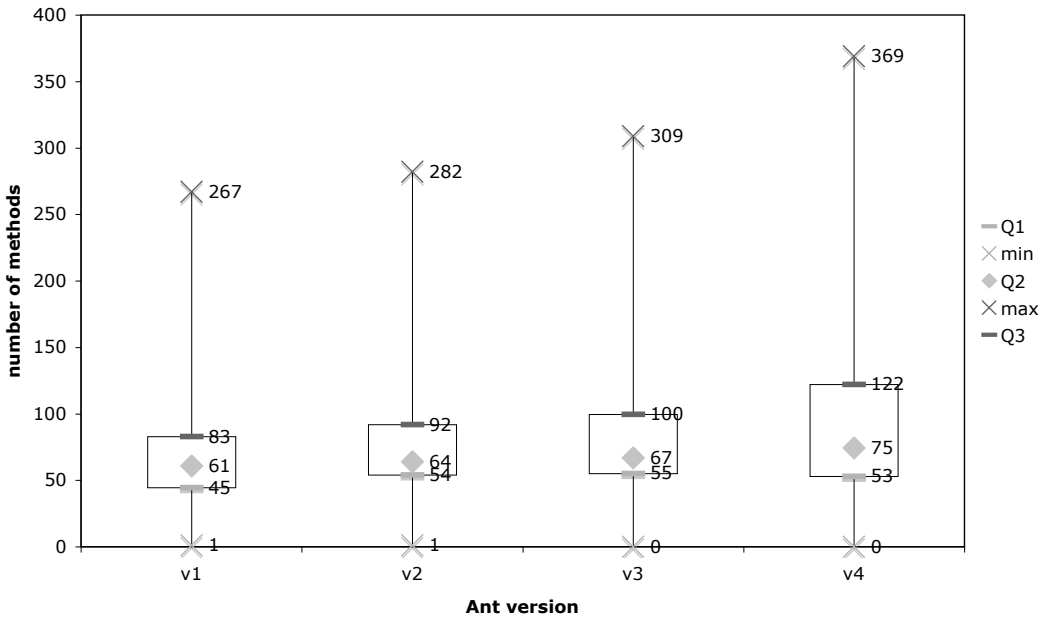


Figure 6.6: Second Experiment: Box plot of the distribution of the number of methods tested by an arbitrary test method.

more new test than new functionality, whereas the transition from v2 to v3 introduced the opposite. The last column in that same table denotes the same percentages but represents them as *the number of covered methods* over *the total amount of instrumented methods*[6]. Between v1 and v2 the relative increase of covered methods is larger than the relative increase of instrumented methods, whereas between v2 and v3 it is smaller. However, the enormous increase of instrumented methods from v2 to v3 seemed curious, especially in combination with the slight decrease in the transition to v4. Although it could be the result of an enormous functionality boost in the transition to v3, perhaps followed by some restructuring in the transition to v4, further investigation of the coverage report revealed a flaw in the distribution of v3. Some packages of third party libraries found their way into the ant package, thereby erroneously being instrumented together with the real ant classes. This resulted in at least 1000 instrumented methods that should not have been taken into account when calculating the

[6]Instrumentation in Emma has the same effect as filtering in our tool, it indicates which methods should be taken into account. However, since this instrumentation is done prior to execution (as opposed to filtering during tracing) it provides a static measure of all methods in the code base.

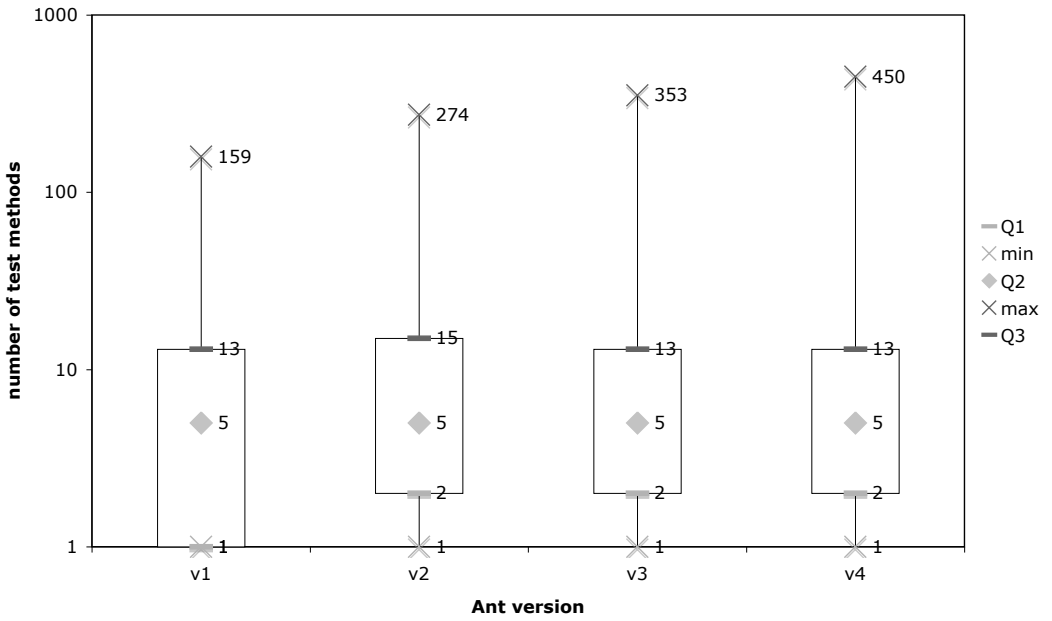


Figure 6.7: Second Experiment: Box plot of the distribution of the number of test methods that test an arbitrary method (logarithmic scale).

coverage for v3. Keeping this flaw in mind, the coverage results indicate that each version, although in variable amounts, added functionality *and* tests. However, it makes no claims as to whether these new tests test the newly created functionality or already existent functionality.

To further investigate our *evolving integration testing strategy* assumption, we queried our dependencies for the `BuildFileTest` class, as it is the basis of the testing framework in version 1.6.x. The dependencies revealed the presence of this class in all versions except for v1. Closer investigation showed that in v1 similar functionality was available in the `TaskdefsTest` class. As the name indicates, this was only used to test Ant's `taskdef` constructs. In the transition to v2 this class was renamed to `BuildFileTest` to be used by all Ant constructs. To investigate the evolution of this testing strategy we queried the dependencies for all test methods that call at least one of these framework methods and for all test methods that call none of them. The fourth column of table 6.6 shows the percentages of test methods that rely on the `BuildFileTest` (or the `TaskdefsTest` for v1). A remarkable result at first sight, as we might have expected this percentage to grow in subsequent versions. However, this merely indicates that the integration

	bare count			percentage	
ant version	yes	no	total	yes	no
v1	153	53	206	74.27%	25.73%
v2	259	81	340	76.18%	23.82%
v3	328	107	435	75.40%	24.60%
v4	414	150	564	73.40%	26.60%

Table 6.6: Second Experiment: Test methods based on `BuildFileTest`.

	method coverage	
ant version	percentage	bare count
1.6.3	61%	3247/5351
1.6.4	63%	3399/5363
1.6.5	65%	3739/5745

Table 6.7: First Experiment: Method coverage as generated by Emma.

testing framework was already in place in `v1` (in the form of the `TaskdefsTest`) and that the increased usage of that framework has been consistent over the different versions: for every four new test methods, three were based on the `BuildFileTest`.

On a final remark, the box plot in figure 6.6 shows a jump from `1` to `0` in the minima, indicating that the two tests methods that run through no methods, which we uncovered in our first experiment, were introduced during the transition from `v2` to `v3`. Further investigation of the dependency data revealed that it were indeed the same methods. However, when verifying this in the source code we noticed that the methods were already introduced in the transition from `v1` to `v2`. To investigate this oddity we queried the logs of our test runs and noticed that, although the test methods were already present in `v2`, they only became part of the running test suite in `v3`.

6.4 Discussion

As you can verify in table 6.1, the test base of our stable experiment does not grow significantly in the transition from 1.6.3 to 1.6.4. But strangely enough, the transition introduces seven new test methods and only five new tested methods,

which is different from the trend in our second experiment where the increase of methods is consistently larger than the increase of test methods. This can have a multitude of causes, but based on the knowledge that 1.6.4 is a bugfix release, it suggests that some bug fixes introduce new test methods to test already existent methods in a different manner, thereby increasing the number of test methods but not the number of tested methods. The transition from 1.6.4 to 1.6.5 shows larger differences, which could indicate that more tests were introduced to test previously untested functionality. But the coverage information represented in table 6.7 shows that, although it was mainly a bug fix release, some new functionality was added as well.

When comparing the evolution of both experiments we can say that, during the phase of high development activity, product code and test code evolve simultaneously which suggests that tests are written during development and not just before every release. In the bug fix phase, the focus is more on the test methods alone, indicating that the functionality is already in place but not tested well enough.

Please note that the term *integration testing* as used in this chapter is slightly different from the original meaning (as defined in section 2.1). Integration testing is focussed on verifying interaction between components, whereas the test suite of Ant is focussed on testing the behaviour of separate components. However, while testing these components they interact with the entire system, as a consequence each test also implicitly tests interactions in the entire system. A better description of the test strategy of Ant would be *unit testing with an extreme lack of stub usage* resulting in a testing strategy that is tightly coupled with the entire system.

It is clear that the goal of Ant's testing framework is providing a simple way for regression testing. In doing so they created a framework that will not easily withstand drastic evolution of the product code. However, as long as the design of Ant stays the same, this is an adequate framework for providing the necessary regression tests.

6.5 Lessons Learned

These experiments have shown us that, solely based on our test dependencies, we can reason about the quality of a single test base in isolation. Thereby enabling

us to *speculate* about the ease of (future) evolution and coevolution of test base and code base.

In order to specifically investigate the coevolution of known history, however, we need to analyse static data as well. We mainly used the static method count gathered by the Emma coverage tool and came to the conclusion that the code base and the test base grow simultaneously. However, further research is needed to investigate whether unit tests are committed prior to, simultaneously with or after their corresponding product code[7]. Whatever approach is used in Ant, based on the overall simultaneous growth of test and code base, we assume that the tests are written in close iteration.

In addition we provided some insights on how focus on possible caveats can be gained quickly by performing some statistics on the massive amount of test dependency data. This information can be the basis for filtering out unwanted noise and focussing on specific oddities.

[7] However, we are not sure whether a test driven approach, where the tests are written before the product code, would be visible on repository level.

Chapter 7

Conclusion

A scientist is happy, not in resting on his attainments,
but in the steady acquisition of fresh knowledge.
— **Max Planck**

7.1 Hypothesis Revisited

We validated that the heuristic metrics on dynamic test dependencies, as proposed in section 1.4, provide a measure for the overall *evolution* of test code. However, to provide a measure for the *coevolution* of test code and product code, we also need a metric that is not solely based on dynamic information. Thus, heuristic metrics on dynamic test dependencies alone do *not* provide a measure for coevolution.

As we added code coverage information to enable investigation of the coevolution of test and product code, we validated and therefore conclude that:

> Heuristic metrics on dynamic test dependencies *in combination with code coverage information* provide a measure for overall coevolution of test code and product code.

7.2 The Bigger Picture

Although we provided a measure for overall evolution and coevolution of test and product code and indicated that our technique is able to quickly locate abnormalities, further research is needed to investigate these abnormalities in more detail. However, we have strong indications that our test dependencies can be a good starting point.

Because of the indications that our test dependencies can *assess* coevolution, we believe that they can be successfully used to *preserve* it in an earlier stage as well. By integrating them into the development environment we hope to improve developer efficiency and overall software quality.

Appendix A

Apache Ant

Introduction. Apache Ant[1] is a Java build tool. It is widely used in open source development and also in industrial projects. Because of this, it is integrated in many popular Integrated Development Environments (IDE's), such as Eclipse, IntelliJ IDEA, ... Ant heavily relies on XML, as the propriety build files are written entirely in XML. A number of extensions to the basic Ant distribution have been written (e.g. GUI's) and there has even been a complete port to the .NET environment (called nANT).

Size and metric overview. The source-file distribution of Apache Ant 1.6.1 contains 1216 Java classes. Only 403 (of these classes around 83 KLOC) are Ant-specific, as most of the classes in the package belong to general purpose libraries of frameworks, such as Apache ORO (for regular expressions) or Apache Xerces (XML parser).

Architectural overview. Now, with the help of the freely available design documentation[2], we will discuss the role the five classes that are considered important by the architects, play in the execution of a build.xml file:

1. `Project`: Ant starts in the Main class and immediately creates a `Project` instance. With the help of subsidiary objects, the `Project` instance parses

[1] http://ant.apache.org/

[2] The design documentation of Ant can be found at: http://codefeed.com/tutorial/ant_config.html

the build.xml file. The xml file contains *targets* and *elements*.

2. `Target`: this class acts as a placeholder for the *targets* specified in the build.xml file. Once parsing finishes, the build model consists of a project, containing multiple targets — at least one, which is the implicit target for top-level events.

3. `UnknownElement`: all the elements that get parsed are temporarily stored in instances of `UnknownElement`. During parsing the `UnknownElements` objects are stored in a tree-like datastructure in the `Target` to which they belong. When the parsing phase is over and all dependencies have been determined, the `makeObject()` method of `UnknownElement` gets called, which instantiates the right kind of object for the data that was kept in the placeholder `UnknownElement` object.

4. `RuntimeConfigurable`: each `Unknown- Element` has a corresponding `Runtime- Configurable`, that contains the element's configuration information. The `RuntimeConfigurable` objects are also stored in trees in the `Target` object they belong to.

5. `Task` is the superclass of `UnknownElement` and is also the baseclass for all types of tasks that are created by calling the `makeObject()` method of `UnknownElement`.

We tried to record the relationship between those 5 classes in Figure A.1. Besides these 5 key classes, the design documentation also mentions five other (helper)classes:
`IntrospectionHelper`
`ProjectHelper2`
`ProjectHelperImpl`
`ElementHandler`
`Main`

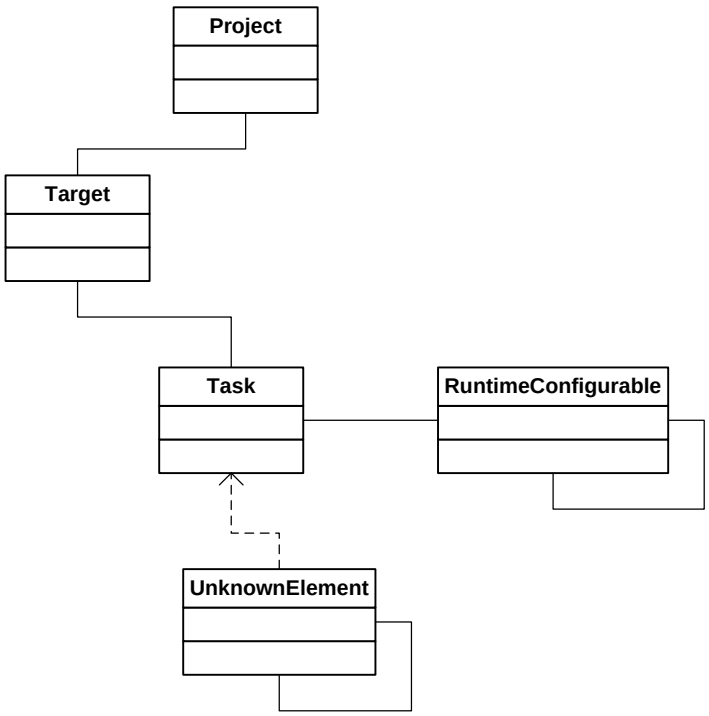


Figure A.1: Simplified class diagram of Apache Ant.

Bibliography

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2001. ISBN 0769510000.

[2] Kent Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-476904-X.

[3] Kent Beck. Simple smalltalk testing: With patterns. *Smalltalk Report*, 4(2), October 1994. Retrieved from http://www.xprogramming.com/testfram.htm on May 14th 2006.

[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[6] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1558606394. Foreword By-Ralph E. Johnson.

[7] Hoare C.A.R. Dijkstra, Dahl O.J. *Notes on Structured Programming*. Academic Press, London, 1972.

[8] Michael Ernst and Jeff H. Perkins. Learning from executions: Dynamic analysis for software engineering and program understanding. In *Tutorial at 21st Annual International Conference on Automated Software Engineering*, November 2005. URL http://pag.csail.mit.edu/~mernst/pubs/dynamic-tutorial-ase2005-abstract.html.

[9] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 2003.

[10] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[12] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3): 270–285, 1993. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/152388.152391.

[13] IEEE Std 610.12-1990 (R2002). Ieee standard glossary of software engineering terminology. Technical report, IEEE, 1990.

[14] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, 1993.

[15] R S Pressman. *Software engineering: a practitioner's approach (5th ed.)*. McGraw-Hill, Inc., New York, NY, USA, 2000.

[16] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[17] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.536955.

[18] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 76–85, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-820-2. doi: http://doi.acm.org/10.1145/1007512.1007523.

[19] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, November 17–20, 2003.

[20] Alberto Savoia. Prevent bug propagation with unit tests. *International Developer*, November 2005. URL http://www.agitar.com/downloads/InternationalDeveloperNovember-05.pdf. Retreived May 24th 2006.

[21] Ian Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-39815-X.

[22] Joel Spolsky. Daily builds are your friend, January 27th 2001. URL http://www.joelonsoftware.com/articles/fog0000000023.html. Retrieved on May 27th 2006.

[23] Amitabh Srivastava and Alan Eustace. Atom - a system for building customized program analysis tools. In *PLDI*, pages 196–205, 1994.

[24] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–106, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-562-9. doi: http://doi.acm.org/10.1145/566172.566187.

[25] Sun. The Java Virtual Machine Profiler Interface documentation, 2004. URL http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html. Retrieved May 7th 2006.

[26] John W. Tukey. *Exploratory data analysis*. MA: Addison-Wesley, 1977. ISBN 0-201-07616-0.

[27] Wikipedia. Standard deviation, May 2006. URL http://en.wikipedia.org/wiki/Standard_deviation. Retrieved May 14th 2006.

[28] Eric Winger. Pragmas: Running tests on method change. Technical report, Cincom Smalltalk, June 24th 2004. URL http://www.cincomsmalltalk.com/userblogs/eric/blogView?entry=3265627283. Retrieved May 24th 2006.

[29] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *ISSRE '97: Proceedings*

of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97), page 264, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8120-9.