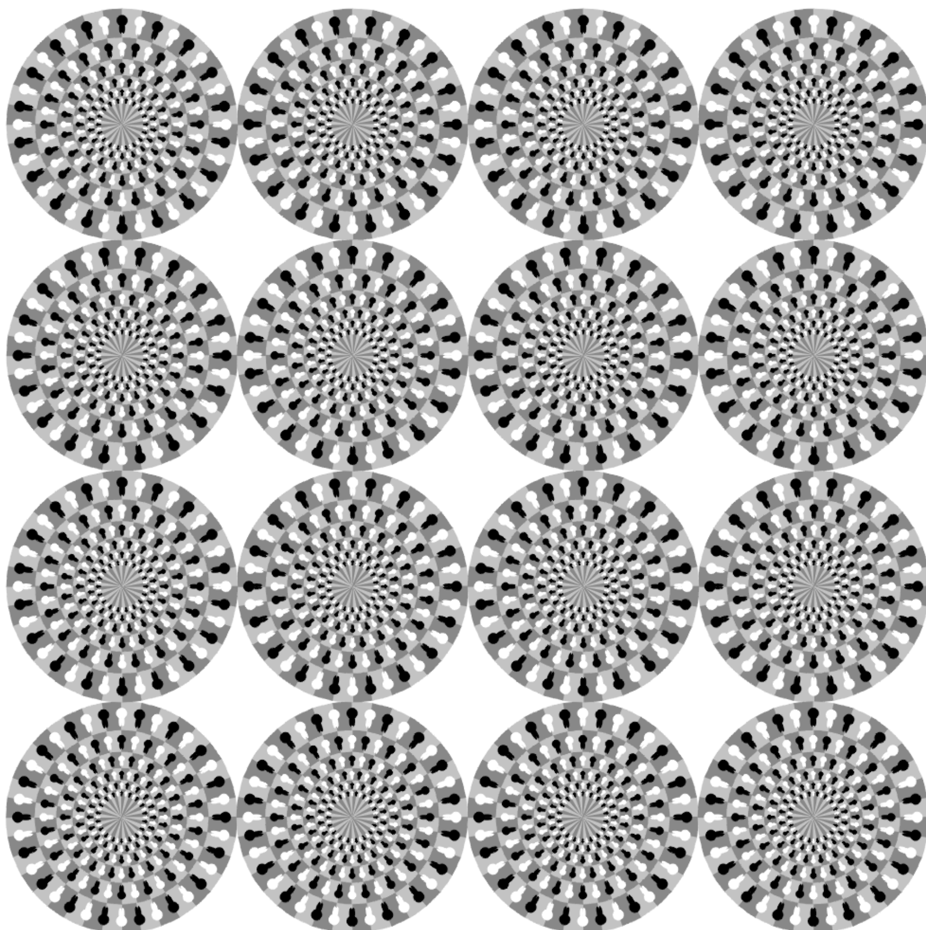
Testability of Object-Oriented Systems: a Metrics-based Approach

Magiel Bruntink



© 2003 Magiel Bruntink

Cover image 'Uzumaki' © 2003 Akiyoshi Kitaoka, used with permission.

> *'U' means rabbits, 'zu' means figure, 'maki' means rotation, and 'Uzumaki' means spirals or swirls. Yellow represents the color of the moon in Japan and it is imagined (though not seriously) that rabbits live in the moon and make mochi (food made from rice).*
>
> Akiyoshi Kitaoka describing the meaning of 'Uzumaki' on his web site. The drawing appeared in his Japanese book *Trick Eyes 2*, and quickly became popular on the Internet.

This work has been typeset using Leslie Lamport's LaTeX, in Donald Knuth's Computer Modern Roman font, 11 pt.

UNIVERSITEIT VAN AMSTERDAM

Testability of Object-Oriented Systems: a Metrics-based Approach

Magiel Bruntink
September 2003

Master's Thesis

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Sectie: Programmatuur
Afstudeerdocent: Prof. dr. Paul Klint
Begeleiders: Dr. Arie van Deursen & Dr. Tobias Kuipers

Preface

When I first saw the ‘Uzumaki’ drawing, it struck me how well it captured my state of mind at the beginning of the work on this thesis. The topic I had chosen to explore appeared to be huge and very difficult, and many things of interest were scattered throughout it. I was unsure where to focus my attention. One cannot help but feel the same way when looking at ‘Uzumaki’; wherever one looks, there is always something going on in another part of the drawing. Hence I decided to put ‘Uzumaki’ on the cover of my thesis.

However, the analogy does not stop there. If one can resist the urge to keep looking at the moving wheels of ‘Uzumaki’, and focus on only one, all the wheels will eventually stop moving. This thesis is the result of focusing on one particular aspect of the topic, testability. I thank my coaches, Tobias Kuipers and Arie van Deursen, for supporting me in resisting the urge to keep looking at the moving wheels.

The topic of this thesis was conceived by Arie, who is never short of interesting ideas or new perspectives. Together with Tobias and Paul Klint, it was decided that the project would be performed at the Software Improvement Group. I was amazed how readily Tobias agreed to be my coach, and allowed me to work at the SIG. Arie, Paul and Tobias, thanks a lot for your feedback and support.

Working at the SIG has been an excellent experience, for which I’d like to thank my colleagues. In particular, I am grateful to Alex van den Bergh, Gerard Kok, Gerjon de Vries, Joost Visser, Patrick Duin and Peter van Helden for allowing me a glimpse of the real world of software engineering. A great advantage of working at a company is the possibility to explore the industry. In my case, I visited a number of people of other companies and interviewed them about their testing efforts. I’d like to thank Joris Alkema, Rutger Buckmann and Vincent van Moppes all of Epictoid, René Clerc of Tryllian and Sylvia Verschueren of ABN-AMRO for their patience and time to answer my questions.

Finally, I am grateful to my friends and family for their support and keeping me in touch with the world outside of research. In particular, Adam Booij was a great help with statistics and Steven de Rooij was brave enough to read early versions of this work. However, none of this would have been possible without the support of my parents, Hiljan and Roel, to whom I dedicate this thesis.

Contents

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Software Testing . . . 1
- 1.3 Testability . . . 4
- 1.4 Assessing Testability Using Metrics . . . 6
- 1.5 Overview . . . 7

2 Testability 9
- 2.1 The Fish-bone In More Detail . . . 9
- 2.2 A Model Of Testability . . . 12
  - 2.2.1 Test Case Generation Factors . . . 13
  - 2.2.2 Test Case Construction Factors . . . 16

3 Related Work 19
- 3.1 Fault Sensitivity . . . 19
- 3.2 Information Loss . . . 21
- 3.3 Visibility . . . 24
- 3.4 Observability And Controllability . . . 28
- 3.5 Test-critical Dependencies . . . 31
- 3.6 Conclusion . . . 33

4 Metrics 35
- 4.1 Source-based Metrics . . . 35
  - 4.1.1 Notation . . . 35
  - 4.1.2 Depth Of Inheritance Tree (DIT) . . . 37
  - 4.1.3 Fan Out (FOUT) . . . 37
  - 4.1.4 Lack Of Cohesion Of Methods (LCOM) . . . 37
  - 4.1.5 Lines Of Code Per Class (LOCC) . . . 38

4.1.6 Number Of Children (NOC) 38
4.1.7 Number Of Fields (NOF) 38
4.1.8 Number Of Methods (NOM) 38
4.1.9 Response For Class (RFC) 38
4.1.10 Weighted Methods Per Class (WMC) 38
4.2 Setting Up The Experiments 38
4.3 Methods 40
4.4 Implementation 43

5 Case Studies 45
5.1 DocGen 45
5.2 Apache Ant 47
5.3 Results 48
5.4 Discussion 50
5.5 Conclusion 57

6 Conclusions 59
6.1 Contributions 59
6.2 Limitations 61
6.3 Future Directions 61

A Results Of The Case Studies 63

B Report On The Interviews 75
B.1 Introduction 75
B.2 Results 75
B.3 Discussion 81
B.4 Conclusion 82

Bibliography 83

List of Figures

1.1 The testability fish-bone. 5

2.1 The *sign* method. 13
2.2 Control-flow graph of method *sign*. 14
2.3 Signature of the *integerAdd* method. 15
2.4 Possible implementation of the *integerAdd* method. 16
2.5 Alternate implementation of the *sign* method. 17

3.1 Example of a component specification. 21
3.2 Plot of the function $(m-1)^n/(m^n-1)$. 23
3.3 Overview of the inputs and outputs of a method. 26
3.4 Example of a non-observable method. 29
3.5 Example of an observable method 29
3.6 Example of a non-controllable method 30
3.7 Example of a controllable method 30

4.1 Methods overview. 40

A.1 DocGen: DIT 64
A.2 DocGen: FOUT 64
A.3 DocGen: LCOM 65
A.4 DocGen: LOCC 65
A.5 DocGen: NOC 66
A.6 DocGen: NOF 66
A.7 DocGen: NOM 67
A.8 DocGen: RFC 67
A.9 DocGen: WMC 68
A.10 Ant: DIT 68
A.11 Ant: FOUT 69
A.12 Ant: LCOM 69
A.13 Ant: LOCC 70
A.14 Ant: NOC 70
A.15 Ant: NOF 71
A.16 Ant: NOM 71

A.17 Ant: RFC 72
A.18 Ant: WMC 72

List of Tables

5.1 Measurement results for DocGen. 48
5.2 Measurement results for Ant. 49
5.3 Hotelling's t values for the source-based metrics versus LOCC. 51

A.1 Spearman correlations between the metrics. 73

Chapter 1

Introduction

1.1 Problem Statement

In this thesis we investigate factors of the testability of object-oriented software systems. The starting point is given by a study of the literature to obtain an initial model of testability, and related software metrics. Subsequently, the metrics are evaluated by means of two case studies of large Java systems. The goal of this thesis is to define and evaluate a set of metrics that can be used to assess the testability of the classes of a Java system.

1.2 Software Testing

Programmers are human beings. Human beings are prone to make errors during most of their activities, and software development is no exception. Thus the need arises for verification of the (half-) products of software development. Software testing is the practice of running a piece of software in order to verify that it works as expected.

The errors made by programmers have the potential of introducing *faults* in the program. Typically faults are confined to a single program statement, but more complex and distributed faults can occur too. Faults in a program have the capability of causing the program to fail. *Failure* happens when the program produces an output that is different from the expected output. In short, programmers make errors and introduce faults in their programs, which become prone to failure. The terms we use here are defined more thoroughly by the IEEE in [12].

Software testing occurs (or should occur!) during multiple phases of the construction of a software system. Typically the software development methodology determines both the kind of testing, and the phase(s) during which testing is done. Since methodology is not our focus here, we will suffice by briefly describing the different kinds of testing that are common in practice. It is useful to consider the several aspects of testing separately. The

following overview of software testing is based on the Software Engineering Body of Knowledge (SWEBOK) [2].

First, we look at the *level* at which testing can take place.

- *Unit* testing is concerned with verifying the behavior of the smallest isolated components of the system. Typically, this kind of testing is performed by developers or maintainers and involves using knowledge of the code itself. In practice, it is often hard to test components in isolation. Components often tend to rely on others to perform their function.

- *Integration* testing is focused at the verification of the interactions between the components of the system. The components are typically subjected to unit testing before integration testing starts. A strategy that determines the order in which components should be combined usually follows from the architecture of the system.

- *System* testing occurs at the level of the system as a whole. On the one hand, the system can be validated against the non-functional requirements, such as performance, security, reliability or interactions with external systems. On the other hand, the functionality implemented by the system can be compared to its specification.

Second, testing can have several objectives. Of course, the base objective of testing is verification of the developed code, however, the reference to be used for verification can be different.

- *Acceptance* testing is done to verify that the system implements the customer's requirements correctly. Usually the testing is done by (future) users of the system. In addition to verifying whether or not the required functionality is present in the system, (future) users are also likely to be concerned about the user-interface and performance characteristics.

- *Functional* testing is done to determine if the system has correctly implemented the specification of functionality. Typically, a team separate to the development or maintenance teams would perform this task.

- *Reliability evaluation* is sometimes done by executing test cases obtained from a typical operational profile for the system. The rate of failure observed during such a test session can then be used to derive statistical measures of the reliability of the system.

- *Regression* testing is performed to make sure that a modification of a certain part of the system has not inadvertently broken other parts of the system. For example, a regression test could entail the execution

of every unit test. Larger projects will likely require a more selective approach if regression testing is to remain viable.

Finally, we discuss the ways in which test cases can be selected.

- *White-box* testing refers to the creation of test cases by exploiting knowledge of the implementation (ie. the source code) of the system under test. Therefore, white-box techniques are typically applied by the same developers that wrote the code.

  Several aspects of the source code can be targeted by white-box techniques. For example, possible techniques are based on the control-flow, data-flow or call behavior of the code being tested. Observing the effects of modifications made to certain parts of the code, so-called mutation analysis, can also be classified as a white-box technique.

- *Black-box* testing is the opposite of white-box testing, in the sense that no knowledge of the implementation is used to generate test cases. Instead, black-box testing focuses on the input/output behavior of the system. This approach enables people without knowledge of the internals of a system to apply these techniques.

  Many black-box techniques take the specification of the system as a starting point. The specification should provide information about the domains of inputs and outputs of the system, and describe the implemented functionality. Using this information, the tester should be able to generate input/output pairs that represent correct executions of the system. In other words, for every pair, the system should result in the specified output value when given the specified input value. Clearly, one such pair exactly represents a test case.

  In general, a system that would pass all possible test cases implements its specification correctly. However, exhaustively testing a system is not a feasible practice, since most interesting systems will likely involve input and output domains that are very large. Therefore, a number of techniques exist to reduce this problem. These provide ways to select a set of test cases that will provide a reasonable level of confidence in the correctness of a system that passes the tests. Examples of these techniques are partitioning of the domains in equivalence classes, boundary-value analysis, random testing, and statistical testing based on an operational profile.

- *Use cases* [13] can also serve as the starting point for a test case. A use case typically gives an informal description of the way a user interacts with the system in order to reach a certain goal. A project applying a use case-centered approach would collect a large number of use cases during requirements gathering and analysis phases. Subsequently each

of the use cases would be used as input for a design and implementation phase, resulting in an implementation of the functionality described in the use case. Given that the use cases have been written at an appropriate level of abstraction, it should be fairly easy to generate a number of test cases from them.

Due to the fact that use cases capture the required functionality in a way that is understandable for both customers and developers, this approach to generating test cases is very suitable for use during acceptance testing.

Of course, the aspects described above are not completely orthogonal, and some combinations of level, objective and technique will not yield a useful testing method. On the other hand, some techniques are likely to be used at a specific level, and in the context of specific testing objective. For example, functional testing would typically occur at the system level using black-box techniques.

1.3 Testability

A software system's testability is defined by the ISO model [21] as "*attributes of software that bear on the effort needed to validate the software product.*" In other words, the testability of a software system is indicative of the amount of effort needed to test the system. This thesis investigates testability from the perspective of unit testing, where our units consist of the classes of an object-oriented software system.

In the world of software engineering, a software system exists by grace of the process supporting it. The process takes care of maintenance of the software and development of new features, among other things. The validation of the software system i.e. testing, is an activity that is tightly coupled to both the software and its supporting process. The amount of effort that one should expect to spend on testing, given a certain desired degree of validity, is therefore a result of properties of both the process and the software. Figure 1.1, adapted from the fish-bone figure in [3], gives an overview of the facets that influence the test effort.

The overview presented in Figure 1.1 allows us to specify the focus of our work. The goal of this thesis is to find a number of source-based metrics, or measures, that can be used to assess a software system's testability. In terms of Figure 1.1, we generally focus on the *implementation* 'major bone', and specifically on the *source code factors* 'minor bone'. The 'bones' of Figure 1.1 each represent an important aspect of a software project with respect to the testing effort. First, the required degree of *validity* sets the testing goal of the project. The higher the goal, the higher the expected effort. Second, *representation* regards factors of the requirements and specification,

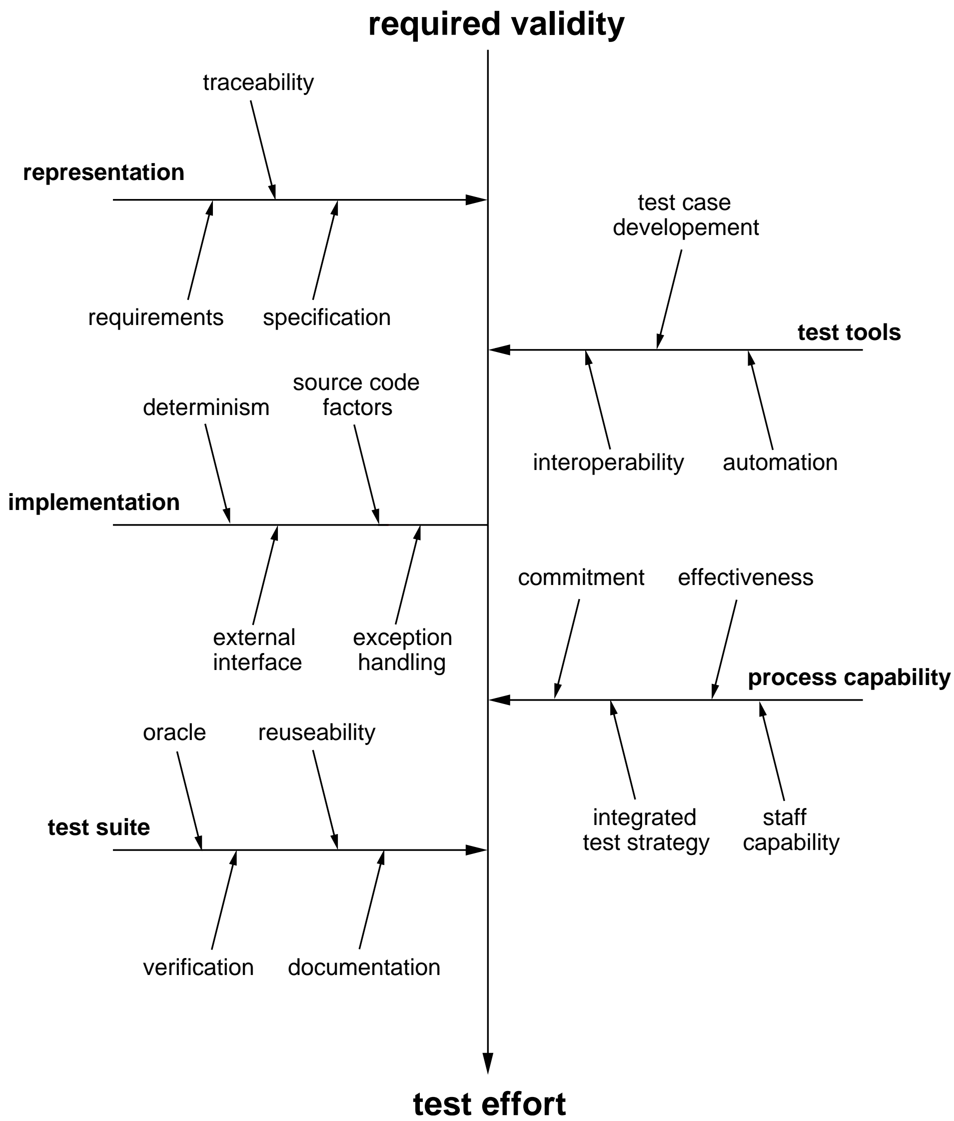


Figure 1.1: The testability fish-bone.

and their influence on the testing effort. Third, *implementation* deals with factors of the actual software, including determinism and interaction with external systems. Fourth, *test suite* takes care of quality aspects of the test suite itself, i.e. its documentation, reuseability and validity. Fifth, *test tools* takes into account the presence of tools that support the project's testing effort. Finally, *process capability* regards the capability of the (testing) staff, and how the project's organization supports the testing effort. The 'major bones' of Figure 1.1 will be described in more detail in Chapter 2.

1.4 Assessing Testability Using Metrics

In this thesis we investigate factors of the testability of object-oriented software systems. Our approach is to evaluate a set of source-based metrics with respect to their capabilities to assess the effort needed for testing. We choose this approach because metrics are a good driver for the investigation of aspects of software. The evaluation of metrics that are thought to have a bearing on the testing effort allows us to, on the one hand, gain insight into the factors of testability, and obtain refined metrics, on the other. Both results can subsequently be used in further studies.

The goal of this thesis is to define and evaluate a set of metrics that can be used to assess the testability of the classes of a Java system. Metrics that have this property are useful for a number of reasons.

First, putting the focus on extraction of facts from source code, as opposed to using design documents, has several advantages of a practical nature. Practice shows that documentation is either severely lacking or simply not present at all. Furthermore, documentation typically consists of natural language texts or diagrams, which can both be ambiguous and hard to process automatically. In contrast, the source code of a software system provides a direct rendering of the implementation, and is very suitable for automatic processing.

Second, being able to determine which parts (classes) of the system cause it to become significantly harder to test allows for the possibility of modifying specific parts of the system in order improve the testability. From the perspective of development, those programming practices or constructions that are known to result in badly testable software can then be avoided.

Finally, in order to support planning and resource allocation, being able to assess the testability based on software metrics is desirable. For example, a system that is in early stages of development typically has no test effort data available yet. The same would be the case for a process where no (accurate) time or effort bookkeeping is done.

We investigate testability from the perspective of object-oriented software systems. Object-oriented programming is a new programming paradigm, which has not yet been subject to the same amount of investigation as

older paradigms have been. In case of the topic of testability, object-oriented programming has received little attention of the literature.

1.5 Overview

In Chapter 2 we further specify the topic of this thesis, and provide some details on related issues. Furthermore, we describe an initial model of testability. Related approaches to our topic are presented and discussed in Chapter 3. At the end of the chapter we discuss the relationships between those approaches and our own approach. In Chapter 4 we define the metrics that we have selected for evaluation, and describe the experiments we do to evaluate them. A detailed description of the methods used for the experiments concludes Chapter 4. Two case studies are described in Chapter 5, followed by a presentation of the results for both case studies. The chapter is concluded by a discussion of the results. Finally, in Chapter 6 we present the concluding notes of this thesis.

Chapter 2

Testability

In this chapter we specify the focus of this thesis, and describe an initial model of testability. This model is used as a starting point for the experiments described in Chapters 4 and 5.

2.1 The Fish-bone In More Detail

The 'fish-bone' presented in Figure 1.1 attempts to specify the focus of this thesis, and put it in context. To provide some additional detail on related issues, we will discuss how each of the 'major bones' influences the testing effort. Parts of this discussion are based on work by Binder [3].

A major input of the test effort picture is the degree of validity that the software is required to have. In general, software that is required to have a high degree of validity will need to be tested thoroughly before it can be claimed the requirement is met.

For some software development projects the required degree of validity may be known explicitly, while for most others the software will simply be expected to 'work'. For example, safety-critical systems are often required to meet very strict validity requirements; maximally allowable failure rates are typically stated explicitely. On the other hand, a word processor application will likely not be required to meet the same degree of validity.

Let's assume that a project intends to verify the validity of the software by means of testing. If the required degree of validity is specified, the goal of testing is clear; to evaluate whether or not the software meets the specified validity requirement. It will depend on the other aspects of the project how much effort will be required to complete the testing.

If the required degree of validity is not specified, the project will need to agree on some kind of testing criterion that indicates whether adequate testing has been performed. In the context of white box testing, such a criterion is typically called a code coverage criterion, because it indicates the extent to which a certain aspect of the code has been 'covered' by testing.

For example, a code coverage criterion could be that all lines of code of the software are executed at least once. In effect, a testing criterion will establish a lower bound on the validity of the software, and an upper bound on the number of test cases required. A project will thus have to make a trade-off between the verification of validity on the one hand, and the required amount of testing on the other. Again, it will depend on the other aspects of the project how much effort will be required to complete the testing according to the selected testing criterion.

In practice, the moment that testing is complete will typically be determined by the amount of effort a project is capable of spending on testing. The 'spine' of Figure 1.1 would thus start at 'available test effort', and point upwards to 'resulting validity'.

Representation

Ideally there is more to a software system than its source code. According to various industry standards, documentation should cover the requirements the software needs to implement and the specification of the chosen solution. The quality of these documents has its bearing on the test effort.

Requirements capture the expectations of the customer, and thus are a crucial source of test cases that determine whether the implementation is correct and complete. From a testing viewpoint, good requirements are unambiguous and quantifiable.

A specification details the architecture and design of the solution that was selected to implement the requirements. Complete and current specifications describe the intended behavior of the implementation. Knowing the intended behavior is valuable if one wants to derive test cases that validate the implementation.

The separation of concerns inherent in modern software documentation raises the issue of traceability. A software system and its documentation is traceable if the relations between the components of the requirements and those of the specification, and those of the specification and implementation, are clear. In other words, it should be easy to point to the components involved in solving a certain requirement. Vice versa, it should be clear which requirement a certain component implements.

A non-traceable software system cannot be effectively tested, since relations between required, intended and current behaviors of the system cannot easily be identified.

Implementation

The core of each software system consists of its implementation. Typically, the implementation is the target of all testing, and thus the extent to which

the implementation allows itself to be tested is a key factor of the required amount of testing effort.

The major part of the implementation of an application consists of source code expressed in one or more programming languages. Factors of the source code that relate to the testability of the implementation, and thus the testing effort, are the topic of this thesis. We will explore this category in the other sections of this chapter.

An implementation that is driven by an external interface can be hard to test. For example, a graphical user interface (GUI) is a source of events that the underlying application must respond to. If no special care has been dedicated to separating the GUI code from the application code, a myriad of problems arises. First, controlling the application to execute certain components would require the generation of GUI events in an appropriate way. Not only will it be needed to have a thorough knowledge of the GUI, but tools supporting the automated generation of GUI events will be needed as well. Second, obtaining result data from an application that is tightly coupled to a GUI can be problematic. In the worst case, values need to be fetched from a GUI component.

Being able to repeat tests is a very desirable asset. Sources of non-determinism present in the implementation or its environment can be obstacles to repeatability. Examples are concurrency-related issues like a non-fixed order of events and contention on resources, but also the use of (pseudo) non-deterministic control or hardware-related problems.

Most applications have some way of dealing with invalid input data, calculation errors and other exceptional conditions. The code responsible for dealing with the condition is a good candidate for testing. However, depending on the nature of an exceptional condition, it may be difficult to reproduce the cause during testing, and thus exercise the responsible code.

Test Suite

Aspects of the test suite itself also determine the effort required to test. First, test cases should be created to allow for automated execution. It should be possible to compare observed output values to expected output values in an automated way, preferably by employing a mechanism called a test oracle. A test oracle is a simple abstraction of the mapping from valid input values to correct output values.

Second, reusing test suites for different revisions and configurations of the system under test must be possible. Test suites should thus be subject to configuration management along with the software itself.

Third, test cases that contain errors are as harmful as buggy code. If they are to be of any use, test suites had better be subject to a verification process of their own.

Finally, test suites need documentation detailing the implemented tests, a test plan, test results of previous test runs and reports.

Test Tools

The presence of appropriate test tools can alleviate many problems that originate in other parts of the 'fish bone' figure. For example, easy-to-use tools will demand less of the staff responsible for testing. Test case definition in the presence of graphical user interfaces is another example where tooling can significantly reduce the required effort.

Obviously, testing benefits from automation of repetitive and error-prone tasks as much as any other activity does. A good set of test tools is capable of interoperating with related tools. For example, a test runner that encounters a failed test is capable of producing a trace which can subsequently be read be debugger or profiler tools, which in turn are linked with an editor, and so forth.

Process Capability

The organizational structure, staff and resources supporting a certain activity are typically referred to collectively as a (business) process. Properties of the testing process obviously have great influence on the effort required to perform testing. Important factors include a commitment of the larger organization to support testing, through funding, empowerment of those responsible, and provision of capable staff.

In order for the process to perform effective testing, i.e. testing the right thing, requirements and specification should be taken as a starting point. Furthermore, the order of testing the various components of the system under test, the test strategy, should conform to the order of development. For example, if the system is developed in a bottom-up fashion, testing should also commence at the lowest level.

2.2 A Model Of Testability

Now that we have specified our focus, it is time to deal with the topic of this thesis. We investigate how factors of the source code of an object-oriented software system relate to its testability. More specifically, we investigate how factors of a class of the system influence the effort needed to unit test the class.

In this section we will introduce the source code factors that will be subject to further investigation in later chapters. We distinguish between two categories of source code factors: factors that influence the *number of test cases required* to test the system, and factors that influence the effort required to develop *each individual test case*. We will refer to the former

category as test case generation factors, and to the latter category as test case construction factors.

2.2.1 Test Case Generation Factors

The number of test cases that has to be created and executed is determined by factors of the source code, on the one hand, and a testing criterion, on the other. Furthermore, the testing criterion determines which source code factors actually influence the number of required test cases. For example, suppose that a project has chosen to create at least one test case for every method. In that case, the total number of methods will clearly determine the required number of test cases. Thus, whether or not a source code factor is a test case generation factor depends on the testing criterion.

A source code factor that is often targeted by testing methodologies is the control-flow of a program. In fact, one of the best known testing methodologies, McCabe's structured testing[27][1], generates test cases based on a program's control-flow. We will give a small example of McCabe's methodology in action. Consider the piece of Java code in Figure 2.1.

```
class Example {
    int sign(int input) {
        int result;
        if(input > 0)
            result = 1;         // 2
        else {                  // 1
            if(input < 0)
                result = -1;    // 3
            else
                result = 0;     // 4
        }
        return result;
    }
}
```

Figure 2.1: The *sign* method.

The method `sign` calculates the sign of its input and yields 1, -1 or 0 if the input is greater than zero, smaller than zero or exactly zero, respectively. According to the structured testing criterion, we have to "Test a basis set of paths through the control-flow graph of each module." Thus, we need the control-flow graph corresponding to `sign`, which is depicted in Figure 2.2. The graph edges in Figure 2.2 have numerical labels that correspond to the

[1]The cited work is a later version of the original work, which was published in 1976.

numbered code statements in Figure 2.1.

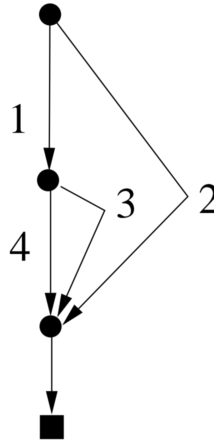


Figure 2.2: Control-flow graph of method *sign*.

A basis set of paths for this control-flow graph is given by:

$$\{\langle 2\rangle, \langle 1,3\rangle, \langle 1,4\rangle\}.$$

In order to comply with the structured testing criterion we will have to create test cases that will cause the paths in the set to be executed. In this simple example, finding suitable test cases is not hard; it is clear that a positive number, a negative number and zero will suffice.

The number of paths in the basis set corresponds to the minimum number of test cases that has to be created to fulfill the structured testing criterion. McCabe terms this number the 'cyclomatic complexity number,' where 'cyclomatic' originates from the graph theoretical 'cyclomatic number'. Instead of by its full name, the cyclomatic complexity number has become known as (the) 'McCabe complexity'.

The example shows how both the testing criterion and the appropriate source code factors determine the number of required test cases. The structured testing criterion is almost directly stated in terms of source code factors, which makes structured testing a distinct white-box approach. Conversely, the black-box approach to unit testing does not target source code factors directly, by definition. However, there are still source code factors that could determine the number of required test cases in a black-box setting. One distinct example is given by work of Voas et. al. [26][23][24], which regards the fault sensitivity of software components. Stated briefly, fault sensitivity is the probability that a fault will be revealed by a randomly selected test case, given that a fault is indeed present. Thus, a low fault sensitivity indicates that many test cases will be required to uncover

potential faults, and vice versa. The topic of fault sensitivity is explored further in various sections of Chapter 3.

Now that we have discussed how the number of required test cases is influenced by generation factors in general, we will look at two features that are specific to object-oriented programming.

Inheritance

Inheritance is a mechanism which allows classes to share their methods and fields. The set of methods and fields of a class is thus the union of the methods and fields that the class inherits, and those defined by the class itself. Depending on the object-oriented language, classes can also redefine the methods they inherit.

The (unit) testing of a class will typically consist of testing its methods. However, through the use of the inheritance mechanism, classes can contain methods that are defined in other classes. Should these methods be tested in the class that inherits them? Should they be tested in the class that defines them? Should they be tested in both? Answers to these questions should be provided by the testing criterion that the project uses.

It is easy to see that inheritance is a source code factor that can influence the number of required test cases. For example, given that the project has decided to test all – inherited and defined – methods of each class, clearly the number of inherited methods of a class will influence the number of required test cases.

Polymorphism

Polymorphism is a feature of object-oriented languages that allows objects to belong to multiple classes. Consider two classes, A and B. Let's say that A is the superclass of B, thus B inherits from A. Now, objects of class B are also objects of class A. In practice this means that objects of class B will be able to fill the role of objects of class A.

The use of polymorphism has possible implications for the number of required test cases. For example, suppose that the signature of method `integerAdd` is specified in Figure 2.3.

```
input:  Number, Number
action: Compute the integer addition of the Numbers
output: int
```

Figure 2.3: Signature of the *integerAdd* method.

In Java, the Number class has several subclasses, which all represent different types of numbers. The method `integerAdd` claims to implement

its operation for all of the subclasses of Number.[2] As a result, a testing criterion will have to tell the testers whether or not to create test cases for objects of all the different subclasses, or what subset of subclasses would be sufficient. Again, the actual testing criterion will determine how use of polymorphism influences the number of required test cases.

Other types of polymorphism lead us beyond the scope of unit testing. For example, consider a possible implementation of the `integerAdd` method in Figure 2.4.

```
class Example {
    int integerAdd(Number a, Number b) {
        return(a.intValue() + b.intValue());
    }
}
```

Figure 2.4: Possible implementation of the *integerAdd* method.

This implementation exploits the fact that all objects of class Number (and thus also all objects of subclasses of Number) contain the `intValue` method, which returns the value of that Number in integer form. However, Number itself does not define an implementation of the `intValue` method; all its subclasses provide an implementation that is suitable for the type of number they represent.

The code in Figure 2.4 does not fix the implementation of the `intValue` method that will be executed; it depends on the specific classes of the objects `a` and `b`. In other words, it is clear that a method by the name `intValue` will be called, however it is not yet known which exact method it will be. McCabe et. al. [27] term this phenomenon 'implicit control-flow'. Since the target code of the implicit control-flow lies outside of the unit-under-test, this use of polymorphism brings us outside of the scope of unit testing.

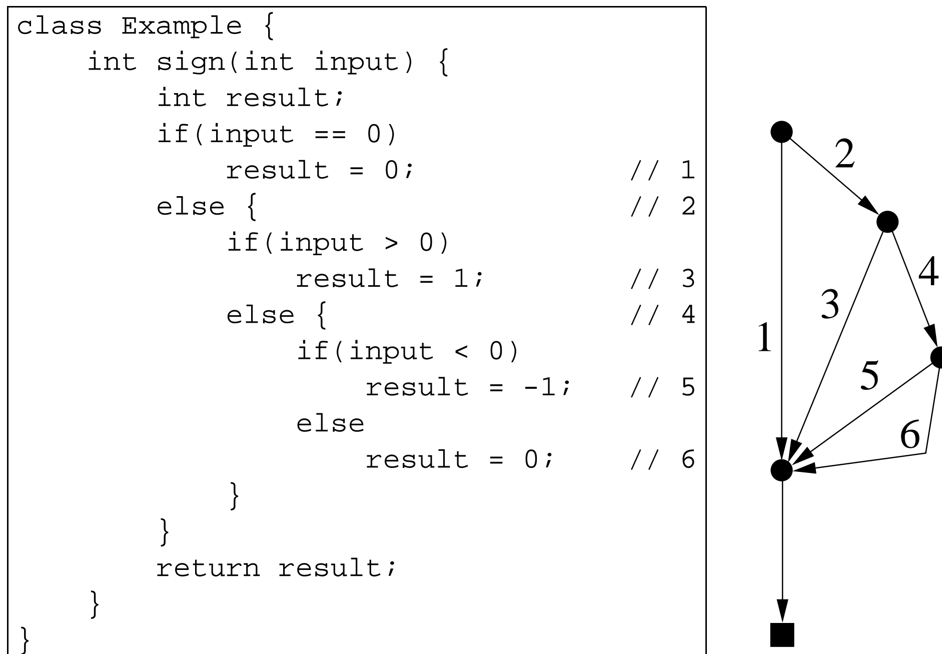
2.2.2 Test Case Construction Factors

If you know what needs to be tested according to your testing criterion, it may seem that creating the required test cases is a trivial task. However, as it turns out, the construction of test cases is at least as difficult a problem as finding out what you need to test. For example, consider an alternate implementation of the `sign` method and its corresponding control-flow graph in Figure 2.5.

A basis set of paths for this control-flow graph is given by

$$\{\langle 1\rangle, \langle 2,3\rangle, \langle 2,4,5\rangle, \langle 2,4,6\rangle\}.$$

[2] Number itself is an abstract class.



```
class Example {
    int sign(int input) {
        int result;
        if(input == 0)
            result = 0;           // 1
        else {                    // 2
            if(input > 0)
                result = 1;       // 3
            else {                // 4
                if(input < 0)
                    result = -1;  // 5
                else
                    result = 0;   // 6
            }
        }
        return result;
    }
}
```


Figure 2.5: Alternate implementation of the *sign* method.

This set of paths poses a problem; there is no input for which the path $\langle 2, 4, 6 \rangle$ will be followed. We can see why this is the case by looking at the code. Edge 2 will be followed if the input *is not* equal to 0, while edge 6 will only be followed if the input *is* equal to 0.

Although the example may appear to be rather artificial, inconstructible test cases are a very real problem. Selection of a testing criterion will have to be done carefully in order to prevent the requirement of test cases that are inconstructible. However, it may be possible to extract useful information from the fact that a certain part of the code cannot be exercised by any test case. In the example above, there is a clear presence of redundant code: the second occurrence of the `if(input == 0)` statement. Indeed, it will depend on the testing criterion whether or not inconstructible test cases indicate true problems of the code.

Even if a test case is constructible in theory, there are source code factors that influence the effort needed to construct it. The unit under test will need to be initialized such that effective testing can be done. In our case, this entails that the fields of (an object of) a class are set to the right values before a test case can be executed. Furthermore, if the class under test depends on other classes, because it used members of those classes, those will need to be initialized. A class which deals with external interfaces (see Section 2.1) will typically require the external components to be initialized as well. Due

to our focus on source code factors, we will not consider the latter source of initialization work. In Chapters 4 and 5 we investigate whether or not these source code factors influence the testing effort.

Chapter 3

Related Work

Several approaches to testability assessment have been proposed in the literature. In this chapter we describe a number of them, and provide our view in a discussion of each approach. The relation of these approaches to our own work is provided at the conclusion of this chapter.

3.1 Fault Sensitivity

Voas et. al. define software testability as the probability that a piece of software will fail on its next execution during testing, provided it contains a fault [26]. In Section 1.2 we described how faults are related to the failure of a program. It is important to realize that a fault will not always cause a failure. The location of the fault, its nature and the program itself are all factors that determine whether or not the fault will lead to a failure. Additionally, the distribution of inputs presented to the program during testing should be taken into account.

In their work, Voas et. al. aim at estimating the probability that a fault at a program location will cause the program to fail during testing. They assume that a black-box testing strategy is used and that the test inputs are uniformly distributed. Furthermore, their analysis assumes that faults are limited to a single location each, and that faults occur on their own.

Sensitivity analysis is a technique to estimate the fault sensitivity of a program location. In order for a fault to be exposed during testing, the location containing the fault must be executed, the fault must cause the data state to become corrupted, and finally the corrupted data state needs to propagate to the output. Sensitivity analysis calculates the probabilities of the occurrence of these events separately. They are named the execution probability, the infection probability and the propagation probability, respectively.

The product of these three probabilities yields the fault sensitivity of the program location: the probability that a fault at that location will be

exposed during testing. A program's overall testability is defined as the minimum sensitivity of all program locations.

Voas et. al. propose techniques for the implementation of the different phases of their sensitivity analysis. First, the execution probability of a location can be estimated by instrumenting the source code of the program with a flag for every program location. Initially all the flags would be down, and only if the corresponding location is executed during a program run would a flag be raised. The instrumented program should be run a number of times, taking input selected randomly from a certain distribution. Subsequently, an estimate for the execution probability of a location is obtained by the number of times the flag for the location was up divided by the number of program runs.

Second, estimating the probability that a fault at a location will corrupt the data state can be done using mutation analysis. The idea is to deliberately introduce different types of faults at a location, and executing the location a number of times given randomly selected data states. Comparing the data states that result from executing the faulty location to the correct data state yields the number of times a fault has corrupted the data state. Finally, dividing this number by the total number of executions gives an estimate for the infection probability of a location.

Third, the propagation probability can be estimated using a combination of data flow analysis and data state perturbation. Given a location, data flow analysis is used to determine the set of 'live' variables. A variable is live if its value potentially influences the output of the program. Now, the set of live variables is perturbed, meaning that the value of one of the variables is deliberately changed. Its resulting value is selected from a specified distribution. The estimation process amounts to the following: first a data state is selected at random, and the set of live variables is determined. One of the live variables is chosen. For a number of times, the value of this variable is perturbed, and the remainder of the program is executed. The remainder of the program is also executed given the original data state, i.e. the one that was not perturbed. Finally, the output that resulted from executing the program with a perturbed data state is compared to the output of the normal execution of the program. An estimate for the propagation probability is now obtained by dividing the number of times a different output was encountered by the total number of executions.

The techniques we have described above are refined in further work by Voas et. al., including [22]. A tool implementing these techniques has also been developed [25].

Discussion

The sensitivity analysis technique was developed mainly for use with procedural programs. Object-oriented programs mainly consist of methods,

which are procedural in nature. Sensitivity analysis could thus be used to determine a method's fault sensitivity, given that the three stages of the analysis can be applied. The techniques described above all have their solutions in the OO domain, which allows the sensitivity analysis technique to be implemented for the OO paradigm.

3.2 Information Loss

A different approach to determining a program's fault sensitivity is presented by Voas and Miller in [23] and [24]. In contrast to the sensitivity analysis presented in Section 3.1, this approach focuses on the analysis of semantic information contained in program specification and design documents. The link with testability is the same; a high fault sensitivity indicates a high testability and vice versa.

Let us consider a program component that accepts an input, performs some specified computation, and produces an output. An upper-bound on this components fault sensitivity is given by the amount of information loss occurring within the component. Information loss can appear in roughly two guises: *Explicit* information loss occurs because the values of variables local to the component may not be visible at the system level, and thus cannot be inspected during testing. *Implicit* information loss is a consequence of the *domain/range ratio* (DRR) of the component. The DRR of a component is given by the ratio of the cardinality of the input to the cardinality of the output. For example, consider the component in Figure 3.1.

```
input:  an integer from the range 1...10
action: if input value is less than 6; return true
        else; return false
output: true or false
```

Figure 3.1: Example of a component specification.

The DRR of this component is $10/2 = 5$. Since this component does not refer to additional (global) variables and does not cause any side-effects, a DRR of 5 indicates that the execution of this component will result in loss of information. Intuitively speaking, the amount of information that flows into the component is larger than the amount of information that flows out. Thus, if a fault would be present in the implementation of the component the loss of information occurring because of the DRR might include information that indicated the presence of a fault.

Discussion

Consider again the component in Figure 3.1. Suppose that the component has been implemented incorrectly. Instead of computing the predicate 'less than 6' it computes the predicate 'less than 5'. Assume that testing the component consists of selecting a number of inputs at random, and comparing the produced output values with the expected values. In this setting, the component will not reveal its fault easily: of the ten possible test inputs, only one (5) will cause failure and detect the fault.

Now suppose that the implementation suffers from another fault: the predicate computed is 'more than or equal to 6'. In this case, every possible test input will reveal the presence of the fault. Clearly, the probability that a fault will be detected during testing is not merely a consequence of the DRR. The nature of the fault itself has a large influence. Therefore, to expose the relation between the DRR and fault sensitivity we will need to deal with the nature of faults.

The information loss viewpoint taken by Voas et. al., despite being intuitive, does not allow us to reason about the nature of faults easily. In the next subsection we will attempt to provide a remedy by taking a functional viewpoint.

Domain/Range Ratio And Fault Sensitivity

Let's first specify what we mean by faults. The specification of a component details its allowed input, the action it will perform and its possible outputs. In other words, the component is expected to implement a surjective function that maps input values to the correct output values. Note that we *expect* the component to implement the function, in reality it might not live up to our expectation. The reason it might not behave as expected is the presence of faults in the implementation. We will assume that faults do not lead to some catastrophic failure, or non-halting behavior of the program. Instead, faults lead to an input/output mapping that differs from the specified one; a different function has been implemented. Consider the following definition of faults:

Definition 1 *A fault causes the implementation of a component to represent a function that maps at least one input value to an incorrect output value. A fault is unique in the sense that each fault is associated with exactly one function.*

From the testing perspective, it is clear which faults are most easily detected: those that cause the implementation to map each input value to a wrong output value. This is exactly what happened in the second example. We will refer to functions that have this property, i.e. mapping each input to a wrong output value, as maximally fault-exposing functions.

Suppose now that we wish to test an arbitrary component. This time we do not know how the implementation is incorrect. We simply assume that the implementation contains an arbitrary fault, and thus represents a function that is different from the one required in the specification.

The number of alternate functions that the component could implement is given by

$$F = m^n - 1, \tag{3.1}$$

with n the cardinality of the domain and m the cardinality of the range.[1] We are interested, though, in the number of maximally fault-exposing functions. This number is given by

$$F^* = (m-1)^n. \tag{3.2}$$

The ratio F^*/F has been plotted in Figure 3.2.

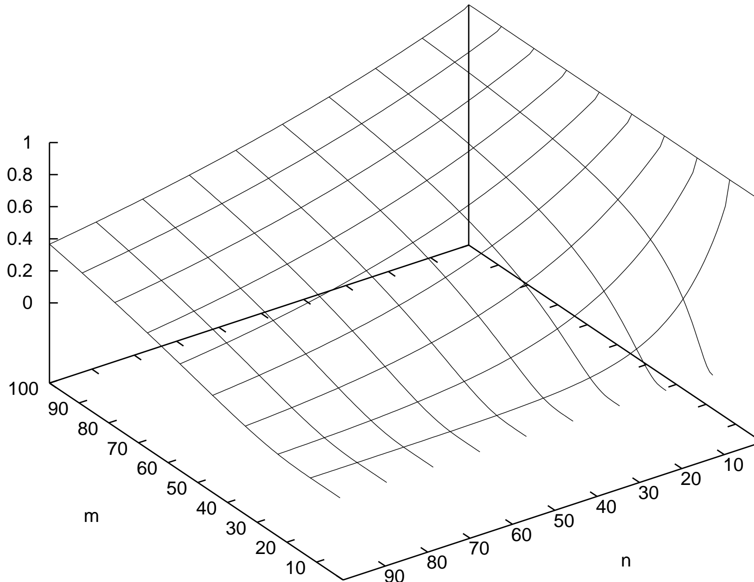


Figure 3.2: Plot of the function $(m-1)^n/(m^n-1)$.

Note that the portion of the plot where $m > n$ is not applicable for our analysis, since we previously assumed that components implement surjective functions. The plot of F^*/F shows that the relative number of maximally fault-exposing functions increases when m increases or n decreases. The DRR (n/m) will thus indicate the relative number of maximally fault-exposing functions for a specified component.

[1] We allow for functions that are not surjective.

We now return to the topic of fault sensitivity. The implementation of the component under test was assumed to suffer from an arbitrary fault. By our previous finding, we can now say that a high DRR will indicate that chances are relatively low that the implementation represents a maximally fault-exposing function. Vice versa, a low DRR indicates that chances are relatively high that a maximally fault-exposing function has been implemented. Finally, the link between the DRR of a component and its fault sensitivity becomes clear by observing that maximally fault-exposing functions are highly fault sensitive: every possible input will reveal the fault! Recall that fault sensitivity is indicative of a component's testability, by definition of Voas et. al. Thus, a high DRR indicates a highly testable component, and vice versa.

3.3 Visibility

McGregor et. al. attempt to determine the testability of an object-oriented system [15]. They introduce the "visibility component" measure (VC for short), which can be regarded as an adapted version of the DRR measure presented in Section 3.2. The VC has been designed to be sensitive to object oriented features such as inheritance, encapsulation, collaboration and exceptions. Furthermore, a major goal of the VC is the capability to use it during the early phases of a development process. Calculation of the VC will thus require accurate and complete specification documents.

The definition of the VC depends on the following terms:

Definition 2 (Explicit parameter) *An object is an explicit parameter of a method iff it is named in the method's signature.*

Definition 3 (Implicit parameter) *An object is an implicit parameter of a method iff it is 'visible' from within the method. The visible objects for a method consist of the fields of the corresponding class and the accessible fields of associated classes.*

In most object oriented languages, access to the fields and methods of a class can be restricted. For example, in Java a field declared as 'private' is only accessible by methods within the same class, whereas a 'public' field would be accessible by all.

Definition 4 (Constant object) *An object is constant with respect to a method iff it is not modified as a result of the execution of the method.*

The VC metric is focused at class specifications, which typically lack information needed to tell which objects are modified by a method. Therefore, constant objects are those made constant syntactically, and are thus not modifiable by any method.

Definition 5 (Constant method) *A method is constant iff its execution does not cause any objects to be modified.*

Again, whether or not a method can be identified as being constant depends on the availability of sufficient information about the method's behavior.

Using these definitions we can give a description of how a method is related to its environment. A method is declared in a certain class, which we will refer to as the declaring class of the method. The method declaration consists of a signature and a body. A method's signature specifies its input and output, while the method's body provides the implementation of the desired functionality. By definition 2, the objects named in the signature make up the method's explicit parameters. Explicit parameters can be divided further into parameters that serve to supply the method with data, parameters that are meant to hold the results of the method, the return value of the method and exception objects thrown by the method. We will refer to the first two types of explicit parameters as explicit input parameters and explicit output parameters, respectively.

A method's signature is not its only source of visible objects. The use of data (objects) that are not local to a method is very common in object-oriented programming. Definition 3 aggregates the non-local objects a method could possibly access, and refers to them as the method's implicit parameters. Again, we have objects that provide the method with data, the implicit input parameters, and objects that are modified by the method, the implicit output parameters. With respect to the calculation of the metric, the set of implicit input parameters would have to be considered equal to the set of implicit output parameters, since it is impossible to tell which objects will be modified given only program specifications. For an overview of a method and its inputs and outputs, see Figure 3.3.

The definition of the VC of a (non-constant) method now follows:

$$\text{VC} = \frac{\text{number of inputs}}{\text{number of outputs}}, \tag{3.3}$$

where the number of inputs is given by the sum of the number of implicit- and explicit input parameters, and the number of outputs is given by the sum of the number of implicit- and explicit output parameters, the number of exceptions thrown and the number of returned values (either zero or one). Constant objects are excluded from the count.

A method's testability is given by the method's VC multiplied by a factor C, which is determined by the number of unique objects that are either explicit input parameters, implicit input parameters, returned objects or exceptions. McGregor et. al. argue that this is a reasonable choice, since the factor C is a count of the objects that may have been modified incorrectly by the method. The assumption is that a large number of (potentially) incorrectly modified objects will allow the method to be easily tested, and

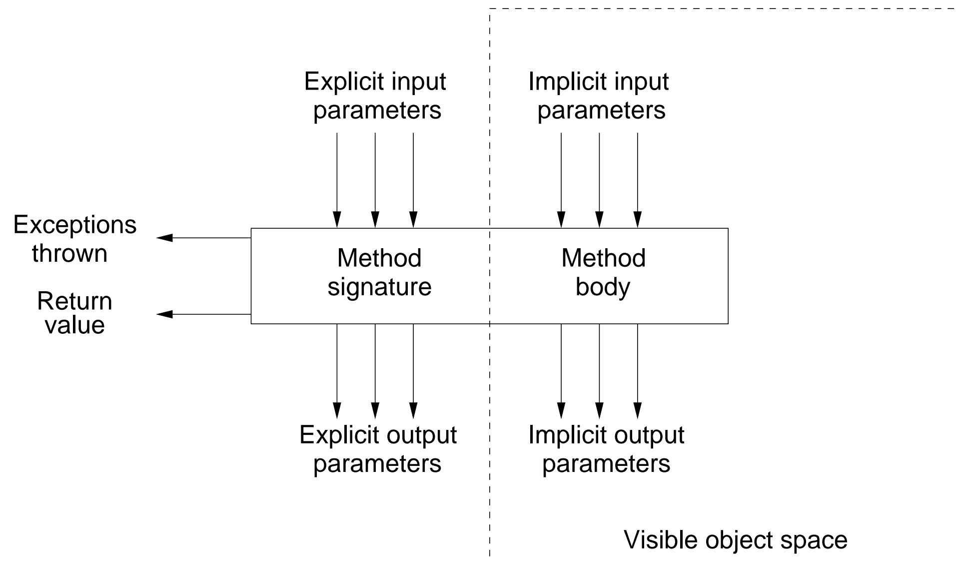


Figure 3.3: Overview of the inputs and outputs of a method.

vice versa, a small number of incorrectly modified objects will cause the method to be hard to test. Subsequently, the testability of a class is given by the minimum testability of its methods, with constant methods being left out of the calculation.

Discussion

By the assumption that a large number of possibly incorrectly modified objects will allow a method to be easily tested, a large visible object space is to be rewarded by the metric. However, the VC itself is not very sensitive to a growing size of the visible object space; a growing object space will cause the VC to approach 1 rather quickly, since both the explicit inputs and explicit outputs will often be relatively few in number. The testability metric is made more sensitive to the size of the visible object space through the introduction of the factor C, which will typically be dominated by the number of implicit input parameters.

Above we mentioned that the VC is sensitive to important features of object-oriented systems. We will now discuss how the VC is influenced by these features.

- *Inheritance* is a mechanism which allows classes to share their fields and methods with each other. Participating classes are involved in an inheritance relation. For each pair of classes that are part of the relation, one, the child, inherits a subset[2] of the fields and methods

[2]The content of the subset is a result of the inheritance rules a specific OO programming

of the other, the parent. Of course the child class is also capable of defining new fields and methods itself. As a result, a class with a large number of parents and ancestors tends to have a large number of fields and methods as well.

In terms of Figure 3.3, a class with a large number of parents and ancestors tends to have large visible object space. As we observed above, a large visible object space will result in a high value for the testability metric. Thus, the testability metric will tend to reward classes that have many parents and ancestors.

- *Encapsulation* is not a concept that originates from the object-oriented paradigm. However, object-orientation capitalizes on the notion of 'shielding' data from modification by otherwise unrelated code. An object is essentially a container for data and operations that are related specifically to the data. Most object-oriented languages support encapsulation, by allowing the programmer to specify access restrictions on both the fields and methods of an object.

  Encapsulation of data and operations limits the visibility of these data and operations to other components of the system. As a result, the visible object space of methods of the other components will be reduced. A smaller visible object space will yield smaller values for the testability metric.

- *Collaboration* occurs when two (or more) objects work together to implement some functionality. A typical construction consists of the collaborating objects having fields that contain references to the other objects. Through these references the objects can access the fields and methods of the others. Note that the set of fields and methods that can be accessed by others can be restricted by encapsulation.

  Much like the effect of inheritance, collaboration expands the visible object space of the methods of each collaborating object. Thus, compared to the situation without collaboration, these methods will receive higher values for the testability metric.

- *Exceptions* are objects that are generated in exceptional (error) conditions. Information about the exceptional condition that occurred is contained in the exception, which facilitates its handling. In a sense, exceptions are outputs of a method, but they are unlikely to be valid output.

  The number of exceptions a method can possibly generate are added to a method's outputs for the calculation of the VC metric. This number is included in the factor C, as well. Therefore, the number of

language uses.

exceptions will positively influence the value of the testability metric for the method.

The behavior of the testability metric based on the VC appears to have some undesirable properties. If one would aim at maximizing the metric during development of a program, one would have to go against good programming practice to reach that goal. For example, properly encapsulated data will reduce the visible object space for other methods, which is considered to be a desirable property according to programming practice, but it does reduce the testability metric. Furthermore, inheritance positively influences the testability metric, while programming practice advises to keep inheritance hierarchies shallow. In their paper [15], McGregor et. al. also remark that classes that have many parents or ancestors are possibly 'complex' to test, yet the metric doesn't seem to incorporate that idea.

The VC and its related testability metric are based on the definition of testability in terms of information loss and fault sensitivity, which was discussed in Section 3.2. In fact, the VC can be seen as an adaptation of the DRR for object-oriented programs. There two problems with the definition of the VC. First, the focus on the use of specification documents – instead of source code – causes the VC to be rather insensitive to the number of implicit parameters; since we cannot determine which implicit parameters will actually be used by a method, the number of implicit input parameters will be equal to the number of implicit output parameters. This problem could be remedied by applying the metric to actual source code.

Second, the metric estimates the amount of information flowing into a component by counting the number of input parameters. Arguably, this estimation is bound to be very inaccurate; objects are very flexible, and could represent structures ranging from a simple number to highly complex composites.

3.4 Observability And Controllability

Freedman applies an approach common in hardware testing to software specifications [8]. Two notions are relevant to Freedman's so-called 'domain testability'. First, the notion of 'observability' intuitively captures the degree to which a component can be observed to generate the correct output for a given input. A component is not observable if so-called 'input inconsistencies' can occur. An input inconsistency is present if a component yields distinct outputs for equal inputs. For example, the piece of Java code in Figure 3.4 contains a method that is not observable.

The method is not observable because calling `notObservable(1)` two times in a row will yield the values 1 and 2. Note that we are not concerned here with the expected behavior of the method! The example shows a possible source of input inconsistencies: the method relies on the state of its

```
class Example {
    int field = 0;

    int notObservable(int input) {
        field = field + input;
        return field;
    }
}
```

Figure 3.4: Example of a non-observable method.

environment. In other words, the method is not strictly a function of its input, but instead it is a function of the union of its input and the state of its current environment. The example in Figure 3.5 shows the same method adapted in such a way that is has become observable.

```
class Example {
    int field = 0;

    int observable(int input, int added) {
        return added + input;
    }
}
```

Figure 3.5: Example of an observable method

Now the method is no longer reliant on the state of its environment, and it has become observable. To use an analog of hardware testing: we have added a pin. The parameter `added` takes over the role of `field` in the method. Having observable components during testing is desirable since observability will guarantee that the component is reliant on its input values only.

Second, the notion of 'controllability' relates to the possibility of a component generating all values of its specified output domain. If no input value exists that causes the component to generate a certain value of the output domain then an 'output inconsistency' is present, and the component is not controllable. For an example, look at the method in Figure 3.6.

Depending on the value of `input` is, the method will return either 1 or 0. Thus, the remaining integer values cannot be generated at all. Adapting the method such that it becomes controllable yet retains its functionality, entails limiting its specified output domain to the values the method is capable of generating. In the case of our example, the output domain should clearly

```
class Example {
    int notControllable(int input) {
        if(input > 0)
            return 1;
        else
            return 0;
    }
}
```

Figure 3.6: Example of a non-controllable method

be changed to the boolean type. Figure 3.7 shows a controllable version of the method.

```
class Example {
    boolean controllable(int input) {
        if(input > 0)
            return true;
        else
            return false;
    }
}
```

Figure 3.7: Example of a controllable method

Adapting (the specification of) a component such that it becomes observable and controllable can be done by introducing extensions. Observable extensions add inputs to account for previously implicit states in the component. Controllable extensions modify the output domain such that all specified output values can be generated. Freedman proposes to measure the number of bits required to implement observable and controllable extensions to obtain an index of observability and controllability.

Discussion

Freedman originally applied his notions of observability and controllability to programs specified and developed using the procedural paradigm. Our focus here is the object-oriented paradigm, and it seems that the applicability of these notions is debatable. Object-oriented programming makes heavy use of (global) state. Objects typically contain data fields and methods are written to access and modify these fields, like in the non-observable example above. An input inconsistency occurred there because of `field` not being considered as part of the input of the method. However, from the object-

oriented viewpoint, it would seem to be more natural if the fields of an object *are* considered to be part of a method's input. Without its containing object, a (non-static) method would not even exist; they are bound together. See Section 3.3 for an approach that does take (global) state into account.

Controllability poses another problem. Consider a method that is specified to return an object of some type. What is the domain of the object type? The answer is not clear-cut, and the next question seems to be: when are two objects different? Typically the programmer is allowed to (re-)define the equality relation of objects. As a result, defining object equality becomes a concern during the specification of the component.

As stated before, the notions of observability and controllability have their origin in the hardware testing industry. There, these notions help to specify components that implement well-behaved, surjective functions. Components of this nature are more easily tested because of the absence of input and output inconsistencies. However, the application of these notions to object-oriented programming is not very natural.

3.5 Test-critical Dependencies

Jungmayr's approach is aimed at finding the dependencies between program components that have a large influence on the program's testability [14]. He defines testability as the effort required to test a component. Dependencies between components require dependee components to be considered during testing of the dependent components, thus increasing effort needed for testing. Removing dependencies between components can therefore improve the testability of a system. Identifying those local dependencies that have a large influence on global testability is worthwhile, since removing dependencies by refactoring is expensive.

A dependency between components (classes) A and B exists if and only if:

- A uses a method or field of B,
- A has a field of type B, or
- A inherits from B.

The components of a system together with the dependency relation can be represented as a directed graph. An edge in such a graph represents one (dependent, dependee) pair from the dependency relation. A dependency graph can contain cycles, which are likely to have a large influence on global testability. During the testing of each component in the cycle it is necessary to consider all other components in the cycle. The dependencies that break dependency cycles when they are removed are elements of a feedback dependency set. Jungmayr hypothesizes that feedback dependencies are good candidates of test-critical dependencies.

In order to identify test-critical dependencies, four system level metrics are defined. These are:

- *Average Component Dependency (ACD)*

  $$\text{ACD} = \frac{1}{n} \times \sum_{i=1}^{n} \text{CD}_i,$$

  where n is the total number of components and CD_i the number of components that component i depends on. Note that both direct and indirect dependencies are included in the count. For example, suppose that component A depends on component B, and component B depends on component C. Now, A is directly dependent on B and indirectly dependent on C. In the absence of other dependencies, the CD of A would be 2.

- *Number of Feedback Dependencies (NFD)*

  $$\text{NFD} = |\text{D}_{\text{fb}}|,$$

  where D_{fb} is a feedback dependency set for the dependency cycles, and $\text{D}_{\text{fb}} \subseteq \text{D}$, the set of all dependencies.

- *Number of Stubs needed to Break Cycles (NSBC)*

  $$\text{NSBC} = |\text{C}_{\text{fb}}|,$$

  where C_{fb} is a feedback component set, i.e. the set of components that are either the source or target of a feedback dependency, and $\text{C}_{\text{fb}} \subseteq \text{C}$, the set of all components.

- *Number of Components within Dependency Cycles (NCDC)*

  $$\text{NCDC} = |\text{C}_{\text{ic}}|,$$

  where C_{ic} is the set of components that are involved in a dependency cycle.

For every identified dependency, above metrics are computed for the system. Then, for every dependency the metrics are recalculated for the system with that particular dependency removed. The relative reduction of the metrics when a particular dependency has been removed is the final indication used to identify the test-critical dependencies.

Based on several case studies, Jungmayr concludes that a small number of dependencies have a large influence on his set of testability metrics. Also, the removal of feedback dependencies leads to a reduction of the metrics that is above average.

Discussion

Jungmayr's approach is based on the assumption that dependencies between components will increase the effort needed for testing. His work does not verify whether or not there is a relationship between component dependencies and testing effort. In our work we try to find our which source code factors are related to the testing effort. The notion of component dependency, albeit in a slightly different form, is included in our initial model of testability (see Chapter 2) and the FOUT and RFC metrics (Chapter 4).

3.6 Conclusion

In this section we will discuss the relation of the work presented in this chapter to our approach, which we described in Chapter 2. We will show how the various source code factors presented in this chapter can be regarded as test case generation factors or test case construction factors.

First, let's consider fault sensitivity (Section 3.1), and derived factors like visibility (Section 3.3) and the DRR (Section 3.2). The fault sensitivity of a component is defined as the probability that the component will reveal a fault during testing, given that a fault is indeed present. The testing of a component consists of running a constant number of test cases, which are selected at random from a uniform distribution. Thus, given that we know the fault sensitivity of a component, we know the probability of finding a fault for the constant number of test cases. If we would require to find faults with a higher probability, the number of test cases should be increased; vice versa, if we would be content with a lower probability, the number of test cases could be decreased. In this sense, the fault sensitivity of a component is a test case generation factor. Since both the DRR and the visibility component measures aim at estimating the fault sensitivity of a component, they belong in the category of test case generation factors as well.

Second, observability (Section 3.4) of a component indicates whether or not the component suffers from input inconsistencies (i.e. the component yields different outputs for equals inputs). Testing a component in the presence of input inconsistencies is not an effective practice, thus it is necessary to compensate such that the component becomes observable. Freedman proposes to adapt the component by introducing observable extensions; a different approach would be to properly initialize the (global) state before testing. Either solution requires additional effort from the tester's part, hence observability belongs in the test case construction category.

Third, controllability (Section 3.4) is related to the inconstructible test cases of Subsection 2.2.2. A component is controllable if and only if all of its specified output values can be generated. Conversely, a component is not controllable if and only if at least one of its specified output values cannot be generated for all of its input values. Thus, it might not be possible to

create a test case for a certain output value of a non-controllable component. Consequently, if we would use the specification of a component to generate test cases, controllability would be a test case construction factor.

Finally, we consider the dependency relation of Jungmayr, which was defined in Section 3.5. Class A depends on class B if either A inherits from B or A accesses a member, i.e. a method or field, of B. We have argued in Section 2.2.1 that inheritance is a generation factor. Regarding the second part of the definition, in Section 2.2.2 we argued that use of members of other classes is a test case construction factor. Consequently, we put Jungmayr's dependency relation in both the test case construction and test case generation categories.

Chapter 4

Metrics

The evaluation of metrics is the core topic of this thesis. In this chapter we define our set of metrics, and set up the experiments to evaluate them. A description of the methods used for the experiments concludes the chapter.

4.1 Source-based Metrics

In this section we describe the metrics we have selected for our experiments. To give each metric a concise and unambiguous definition, we use (a subset of) the notation introduced by Briand et. al. in [5]. For ease of reference, we first provide the relevant notation.

4.1.1 Notation

Definition 6 (Classes) *An object-oriented system has of a set of classes, C. For every class $c \in C$ we have the following:*

- $Parents(c) \subset C$*, the set of classes from which c inherits directly.*
- $Children(c) \subset C$*, the set of classes that inherit directly from c.*
- $Ancestors(c) \subset C$*, the set of classes from which c inherits either directly or indirectly.*

From the definitions it follows that $Parents(c) \subseteq Ancestors(c) \subset C$.

Definition 7 (Methods) *Let $c \in C$, then we have:*

- $M_{In}(c)$*, the set of methods that c inherits.*
- $M_D(c)$*, the set of methods that c newly declares, i.e. $m \in M_D(c)$ iff m is declared in c and $m \notin M_{In}$.*
- $M_{Im}(c)$*, the set of methods that c implements, i.e. c defines a body for m. Clearly, for each $m \in M_{Im}(c)$ either $m \in M_{In}$ or $m \in M_D$.*

- $M(c) = M_D(c) \cup M_{In}(c)$, *the set of methods of* c.
- $M(C) = \cup_{c \in C} M(c)$, *the set of all methods.*

Definition 8 (Method Invocations) *Let* $c \in C$, $m \in M_{Im}(c)$ *and* $m' \in M(C)$, *then we have:*

- $MI(m)$, *the set of methods invoked by* m. $m' \in MI(m)$ *iff the body of* m *contains an invocation of method* m'.

Definition 9 (Fields) *Let* $c \in C$, *then we have:*

- $F_{In}(c)$, *the set of fields that* c *inherits.*
- $F_D(c)$, *the set of fields that* c *newly declares, i.e.* $f \in F_D(c)$ *iff* f *is declared in* c *and* $f \notin F_{In}$.
- $F(c) = F_{In} \cup F_D$, *the set of fields of* c.
- $F(C) = \cup_{c \in C} F(c)$, *the set of all fields.*

Definition 10 (Field References) *Let* $m \in M_{Im}(c)$ *for some* $c \in C$ *and* $f \in F(C)$, *then we have:*

- $f \in FR(m)$ *iff the body of* m *contains a reference to* f.

We now introduce the metrics that we evaluate in our experiments. A similar set of metrics has been proposed by Binder to measure structural testability factors of the source code [3]. However, Binder did not provide an operational definition of the metrics. In the following subsections we define each metric operationally.

Binder distinguishes two structural factors that influence the testability. First, *complexity* factors, which influence the 'complexity' of the development of test cases. Second, *scope* factors, which influence the amount of test cases that need to be developed. Binder's complexity and scope factors are similar to our test case construction and test case generation factors, respectively. The major reason for our selection of these metrics is the close match between our model and Binder's. Compared to several other metrics presented in the literature (see Chapter 2), our metrics have the additional advantage of being easier to implement and understand.

Many of the metrics that Binder proposes originate from an object-oriented metrics suite by Chidamber and Kemerer [7]. Chidamber and Kemerer also suggest that some of their metrics have a bearing on the testing effort; in particular, their Coupling Between Objects (CBO) and Response For Class (RFC) metrics. Other metrics from Chidamber and Kemerer's suite that are included in our set are Depth Of Inheritance Tree (DIT), Number Of Children (NOC), Weighted Methods Per Class (WMC) and Lack Of Cohesion Of Methods (LCOM).

For the following subsections, let $c \in C$ where C is the set of classes for some object-oriented system.

4.1.2 Depth Of Inheritance Tree (DIT)

$$\text{DIT}(c) = |Ancestors(c)|$$

The definition of DIT relies on the assumption that we deal with object-oriented programming languages that allow each class to have at most one parent class; only then will the number of ancestors of c correspond to the depth of c in the inheritance tree. Our subject language – Java – complies to this requirement, however C++ does not.

4.1.3 Fan Out (FOUT)

$$\text{FOUT}(c) = |\{d \in C - \{c\} : uses(c, d)\}|$$

where $uses(c, d)$ is a predicate that is defined by:

$$\begin{aligned} uses(c, d) \quad \leftrightarrow \quad & (\exists m \in M_{Im}(c) : \exists m' \in M_{Im}(d) : m' \in MI(m)) \vee \\ & (\exists m \in M_{Im}(c) : \exists a \in F(d) : a \in FR(m)) \end{aligned}$$

In words, $uses(c, d)$ holds if and only if a method of c either calls a method of d, or a method of c references a field of d.

The FOUT metric we use is an adaptation of Chidamber and Kemerer's CBO metric. In a sense, FOUT is a one-way version of CBO; it only counts the number of classes that c uses, not the classes it is used by. The definition of CBO follows from the definition of FOUT if $uses(c, d)$ is replaced by $uses(c, d) \vee uses(d, c)$.

4.1.4 Lack Of Cohesion Of Methods (LCOM)

$$\text{LCOM}(c) = \frac{\left(\frac{1}{a}\sum_{f \in F_D(c)} \mu(f)\right) - n}{1 - n}$$

where $a = |F_D(c)|$, $n = |M_{Im}(c)|$ and $\mu(g) = |\{m \in M_{Im}(c) : g \in FR(m)\}|$, the number of implemented methods of class c that reference field g.

This definition of LCOM is proposed by Henderson-Sellers in [10]. It is easier to both compute and interpret compared to Chidamber and Kemerer's definition of the metric. The metric yields 0, indicating perfect cohesion, if all the fields of c are accessed by all the methods of c. Conversely, complete lack of cohesion is indicated by a value of 1, which occurs if each field of c is accessed by exactly 1 method of c. It is assumed that each field is accessed by at least one method, furthermore, classes with one method or no fields pose a problem; it is assumed that they do not occur.

4.1.5 Lines Of Code Per Class (LOCC)

$$\text{LOCC}(c) = \sum_{m \,\in\, M_{Im}(c)} \text{LOC}(m)$$

where LOC(m) is the number of lines of code of method m, ignoring both blank lines and lines containing only comments.

4.1.6 Number Of Children (NOC)

$$\text{NOC}(c) = |Children(c)|$$

4.1.7 Number Of Fields (NOF)

$$\text{NOF}(c) = |F_D(c)|$$

4.1.8 Number Of Methods (NOM)

$$\text{NOM}(c) = |M_D(c)|$$

4.1.9 Response For Class (RFC)

$$\text{RFC}(c) = |M(c) \cup_{m \in M(c)} MI(m)|$$

The RFC of c is a count of the number of methods of c and the number of methods of other classes that are potentially invoked by the methods of c.

4.1.10 Weighted Methods Per Class (WMC)

$$\text{WMC}(c) = \sum_{m \,\in\, M_{Im}(c)} \text{VG}(m)$$

where VG(m) is McCabe's cyclomatic complexity number[27] for method m.

4.2 Setting Up The Experiments

Now that we have defined a set of metrics, it is time to set up experiments to evaluate them. Empirical study within the field of software engineering is relatively rare. As a result, no unified approach for such experiments has been adopted yet. We use the GQM/MEDEA[1] framework proposed by Basili et. al. in [6]. First, we state the goal of our experiments:

[1] Goal Question Metric / MEtric DEfinition Approach

Goal: To assess the capability of the proposed source-based metrics to predict the testing effort.

Next, we describe our perspective on the goal, and relevant factors of the environment, the context of the experiments.

Perspective: We evaluate the source-based metrics at the class level, and limit the testing effort to the unit testing of classes. Thus, we are assessing whether or not the values of the source-based metrics can predict the required amount of effort needed for unit testing a class.

Environment: The experiments are targeted at Java systems, which are unit tested at the class level using the JUnit testing framework. Further relevant factors of the study systems will be described in Chapter 5.

To help us translate the goal into measurements, we pose questions that pertain to the goal:

Question 1: Are the values of the source-based metrics for a class associated with the required testing effort for that class?

Answering this question directly relates to reaching the goal of the experiments. However, to answer it, we must first quantify 'testing effort'. To indicate the testing effort required for a class we use the size of the corresponding test suite.

The (projected) size of software has been used as an indicator for many attributes, including defect ratios, cost, maintenance effort, development effort and development time. Well-known cost models such as Boehm's COCOMO[4] and Putnam's SLIM model[18] relate development cost and effort to software size. Test suites are software in their own right; they have to be developed and maintained just like 'normal' software.

Now we refine our original question, and obtain the following new question:

Question 2: Are the values of the source-based metrics for a class associated with the size of corresponding the test suite?

Subsequently, we must decide on how to quantify the size of a test suite. For our experiments we use the dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases) metrics to indicate the size of a test suite. The 'd' prepended to the names of these metrics denotes that they are the *dependent* variables of our experiment. The dLOCC metric is defined like the LOCC metric. The dLOCC metric is applicable because typical use of JUnit would be to test a class using a single test class. The dNOTC metric provides a different perspective on the size of a test suite, and will be defined in Section 4.3.

Having quantified test suite size, we arrive at the final question:

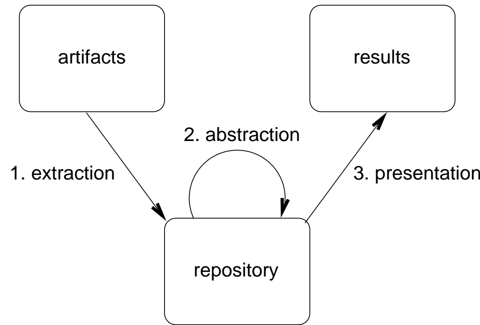


Figure 4.1: Methods overview.

Question 3: Are the values of the source-based metrics for a class associated with the dLOCC and dNOTC metrics of the corresponding test suite?

From Question 3 we derive the hypotheses that our experiments will test:

H_0 (m, n)**:** There is *no* association between source-based metric m and test suite metric n,

H_1 (m, n)**:** There is an association between source-based metric m and test suite metric n,

where m ranges over our set of source-based metrics, and n is either the dLOCC or dNOTC of associated test suites. In the next section we will show how we evaluate these hypotheses.

4.3 Methods

Figure 4.1 gives an overview of the methods used for the experiments. The approach we have taken is based on the process of reverse engineering, which is defined in [16]. We will discuss each of the steps depicted in Figure 4.1 involved, and the tools that we used to support them.

1. Facts about the subject system are gathered from system *artifacts* during the *extraction* phase. An artifact is an entity that is either part of the system or closely related to the system. Example artifacts are the system's source code or documentation, but also developers or business procedures. The extracted facts are subsequently stored in a *repository*. A repository may consist of a database, a collection of plain files or other storage mechanisms.

2. Next, the facts contained in the repository are used during the *abstraction* phase, to derive additional (meta-)facts about the system and store them in the repository.

3. Finally, during the *presentation* phase a subset of the (meta-)facts contained in the repository is selected and presented to the client of the reverse engineering process.

Although the goal of our experiments is not the analysis of a certain system – but the evaluation of metrics – our approach is largely the same. The extraction phase consists of the calculation of the value of each metric for every class of the subject system, and the storage of these values in a repository. The Eclipse tool platform [17] is used to calculate the source-based metrics. An existing plug-in for Eclipse, the "Eclipse metrics plug-in (version 1.08)" by Frank Sauer[2], was extended to calculate our set of metrics for a given system, and store the results in a relational database[3].

The systems that are subject to our case studies (see Chapter 5) both are unit tested at the class level using the JUnit testing framework[4]. The JUnit framework allows the user to create (and run) classes that are capable of unit testing a part of the system. A typical practice would be to create a test class for every class of the system. In the case of our study systems, there is approximately one test class responsible for the unit testing of each class. The sizes of the test classes, i.e. their dLOCC and dNOTC values, are determined using the same tool that we used to measure the system classes.

The dNOTC metric is calculated by counting the number of invocations of JUnit 'assert' methods that occur in the code of a test class. JUnit provides the tester with a number of different 'assert' methods, for example 'assertTrue', 'assertFalse' or 'assertEqual'. The operation of these methods is the same: the parameters passed to the method are tested for compliance to some condition, depending on the specific variant. For example, 'assertTrue' tests whether or not its parameter evaluates to 'true'. If the parameters do not satisfy the condition, the framework generates an exception that indicates a test has failed. Thus, the tester uses the set of JUnit 'assert' methods to compare the expected behavior of the class-under-test to its current behavior. As a result, by counting the number of invocations of 'assert' methods, we count the number of comparisons of expected and current behavior. We consider the latter to be an appropriate definition of a test case.

To summarize the extraction phase; both system classes and test classes are measured using an Eclipse plug-in, which stores the resulting values in a relational database.

Next, during the abstraction phase we calculate Spearman's rank-order correlation coefficient, r_s, for each source-based metric of the system classes and both the dLOCC and dNOTC metrics of the corresponding test classes. Spearman's rank-order correlation coefficient (r_s hereafter) is a measure of

[2]Information available at `http://sourceforge.net/projects/metrics`

[3]MySQL, information available at `http://www.mysql.com`.

[4]Information available at `http://www.junit.org`.

association between two variables that are measured in at least an ordinal scale [19]. The measurements are ranked according to both variables. Subsequently, the measure of association is derived from the level of agreement of the two rankings on the rank of each measurement. The value of r_s can range from -1, indicating perfect *negative* correlation, to 1, indicating perfect *positive* correlation. An r_s value of 0 indicates no correlation.

The r_s is a non-parametric statistic, which allows its application even if the distribution of the data is not known. This fact is the main motivation for our use of r_s, since we indeed lack knowledge about the distribution of the metric values. As a side note, recent work by Wheeldon and Counsell shows that several measures of object-oriented systems, including coupling and method or field counts, induce power law distributions [28]. A power law distribution implies that small values are extremely common, while very large values are extremely rare. Possibly, different statistics could be applicable to our experiments since some of our metrics are similar to those of Wheeldon and Counsell.

In order to test the hypotheses based on the observed value of r_s, we estimate the significance of the observed value of r_s by first calculating the t statistic. The t statistic is defined as follows:

$$t = r_s\sqrt{\frac{N-2}{1-{r_s}^2}}$$

where N is the number of measurements, i.e. the number of class – test class pairs. The statistical significance of the value of t indicates the probability that the observed value of r_s is a chance event. If the value of $r_s(m,n)$ is a chance event, we would be in error if we would use it to reject $\mathrm{H}_0(m,n)$, and accept the converse $\mathrm{H}_1(m,n)$). Thus, the statistical significance of the value of t will allow us to reject $\mathrm{H}_0(m,n)$,and accept $\mathrm{H}_1(m,n)$, with a certain level of confidence.

To calculate r_s, we need to find the corresponding test class for every system class. The JUnit documentation suggests that test classes should be named after the class they test, by appending "Test" to the class' name. Since this convention is used in both our study systems, we are able to associate a class and its test class in an automated way.

Both r_s and t are calculated for each source-based metric m and the dLOCC and dNOTC metrics of the test suite. First, the values for all classes for source-based metric m are fetched from the repository. Subsequently, each value for a class is paired with both the dLOCC and dNOTC values of the corresponding test class. The resulting pairs are then used to calculate r_s. Finally, t is derived from the value of r_s and the number of pairs involved, and the statistical significance (p) of t is obtained from a standard table [19]. This process is repeated for all the source-based metrics in our set, and

finally the results are presented in a table (see Chapter 5).[5]

4.4 Implementation

To support our experiments, we implemented a tool to calculate the metrics described in this chapter. The tool is based on the "Eclipse metrics plug-in (version 1.08)" by Frank Sauer, and hence is built on the technology provided by the Eclipse tool platform [17]. Functionality to calculate many of our metrics was already present in that version of the plug-in. We added support for the FOUT, RFC and dNOTC metrics, and adapted the existing implementations of the field and method counts to better reflect our NOF and NOM metrics. The Eclipse platform extension mechanism allowed us to quickly integrate the new metrics in the plug-in. Furthermore, the Java tools provided by Eclipse could be used well for the implementation of the metrics.

Secondly, we extended the export mechanism of the plug-in with the possibility to store the measurement results in a MySQL database. The original plug-in offered exporting of the results to an XML file, but this did not suffice for our purpose. The size of our case studies resulted in XML files that were very large and hard to process. The use of a relational database made it possible to efficiently store and access the data.

Finally, the original plug-in operates in an interactive modus. The user can view the metric values in a window while browsing a Java project in the Eclipse user-interface. For our purpose operation in a batch modus is more appropriate. We wanted to be able to calculate the metrics for all the classes of some Java system, without being required to do any browsing using the user-interface. The Eclipse platform allows plug-ins to run without the user-interface attached, effectively handing control to the plug-in itself. We implemented the necessary classes to run the plug-in in such a 'head-less' state, and allowing it to be invoked from the command line.

The calculation process itself is straightforward. Assuming the plug-in has been invoked from the command line, i.e. it is operating in batch modus, the following steps occur:

- The hierarchy of Java elements, i.e. methods, types, classes and packages, is traversed.
- For each Java element:
  - The appropriate metrics are calculated for the category of the Java element.
  - The metric values are stored in a data structure in memory.

[5] Thanks to Adam Booij for his help with the calculation of the statistics, and his help with statistics in general.

- The stored metric values are exported to the database.

The set of appropriate metrics for a Java element is determined by the configuration of the plug-in. Each metric is associated with a variable indicating the category of Java elements for which it is appropriate. For example, all of the metrics defined in this chapter are appropriate at the class level.

The interactive modus is simpler in operation. When the user selects a Java element in the Eclipse environment, all appropriate metrics are calculated and their values are displayed in a window. There is no export phase occurring in the interactive modus, unless the user requests it. However, we have not implemented the database export functionality for the interactive modus.

The calculation of our metrics uses the Eclipse Java parser to obtain an abstract syntax tree (AST) representation of the Java element. Subsequently, the AST is used to calculate the actual metric value. Many of our metrics traverse the AST using visitors, which originate from the visitor design pattern, defined by [9]. The support of the Eclipse platform for this kind of traversal allowed us to implement the new metrics with little effort.

We have implemented the tool mainly to support our experiments, and consider it to be in a prototypical phase. Only two weeks were spent on development of the tool. There are two main problems present in the current implementation:

- Memory usage is rather excessive. For example, the calculation of the complete set of metrics for the DocGen program (see Section 5.1) requires about 300 megabytes of memory. We are aware of two main causes for this problem. First, the Eclipse platform loads many unneeded components, even when the plug-in operates in batch modus and does not use the user interface. Second, the database access mechanism is sub-optimal. We have reused this functionality from the DocGen program itself, which in turn uses Apple's WebObjects object-relational mapping. Our current solution requires that many objects need to exist in memory at the same time.

- The reuse of DocGen's database access functionality also present us with another limitation. Because of licensing issues, the plug-in cannot be distributed in the public domain.

Chapter 5

Case Studies

In this chapter we describe the two software systems that were used for our experiments. The first is DocGen, a source code documentation tool in development at the Software Improvement Group. The second is Apache Ant, an automation tool for software development. Both systems are suitable candidates for study for a number of reasons. First, because of an internship at the Software Improvement Group, the DocGen source code was readily available to us. Apache Ant is an open source project, and hence we were also able to obtain its source code easily. Second, both systems are unit tested at the class level, which is the perspective we assume in this thesis. Third, both systems use the JUnit testing framework to implement their test suites. Finally, DocGen and Ant are both large Java systems, containing 138 and 111 test classes, respectively.

The results of the experiments are presented and discussed at the end of the chapter. The intermediate results of the measurements are included in Appendix A.

5.1 DocGen

DocGen is a documentation generator, developed by the Software Improvement Group (SIG)[1]. It processes source code of other programs, and generates technical documentation based on facts that are contained in the source code. The documentation consists of graphical representations of control-flow and data-flow dependencies, metrics, relevant pieces of source code, and other kinds of information. All the different kinds of the documentation are integrated into a web site, which allows the user to navigate the documentation by following hyperlinks. This section is based on a number of interviews done with the developers of DocGen. The interviews also served to obtain an intuitive notion of test difficulties as they occur in practice. During the discussion of the results, we will see that some of our metrics

[1] Information available at `http://www.software-improvers.com`.

are able to identify at least one of the classes that was mentioned as being badly testable. For further details on these interviews, see Appendix B.

DocGen is an instance of the reverse engineering concept, which we described in Section 4.3. The program first reads the source code, and extracts facts from it. For example, it records function call relationships between modules. The facts are stored in a repository. Subsequently, additional facts are inferred from the facts already in the repository. For example, some modules may not be called by any other module; a possible case of unreachable code. Based on the configuration of DocGen, derived facts such as unreachable code may be included in the documentation. Finally, the program generates a web site representing the entire documentation of the source code. The ideas and technologies that are the foundation of DocGen are described by van Deursen and Kuipers in [20].

The DocGen program is moderately large; its Java source code is about 3 megabytes in size, and contains roughly 90,000 lines of code. There are 66 packages, which contain 640 classes in total. Of these 640 classes, 138 classes have an associated test class. These are the classes that will be used for our experiment. DocGen is still in development, thus it evolves quickly. Furthermore, many different customized versions are both developed and maintained. Our experiment is based on the demonstration version of DocGen, dated May 19, 2003.

In addition to a description of the software itself, we provide information about the process that supports the development and maintenance of DocGen. The development of DocGen is based on Kent Beck's eXtreme Programming methodology [1]. We highlight some of the practices in use at the SIG. Refer to Appendix B for further details.

Testing Unit tests are continuously developed by the programmers themselves, if possible in a test-first fashion. Test-first development entails the creation of tests prior to the development of the actual code. Additionally, customers provide test cases of their own, which test the acceptance level of DocGen (see Section 1.2).

Refactoring The software is restructured to obtain higher quality code, while the behavior of the software is kept intact.

Pair programming Programmers working on production code work in pairs, on a single machine.

Collective ownership Each team member can access and modify any piece of code in the system.

Continuous integration The entire system is built and tested periodically. Every night, all the customized versions of DocGen are built and their suites of unit tests are run.

Coding standards The development of code is governed by simple rules on formatting, variable naming and structure. For example, a method should not have more than 12 statements, and should have at most one `return` statement.

As implied by the above practices, the testing of DocGen is not a separate process; it is an integral part of the development process. However, two issues are relevant to any kind of testing process. First, what testing criterion is used by the process, i.e. what needs to be tested before testing is complete? At the SIG, every method should be 'tested'. The test cases are created in an ad-hoc fashion, i.e. the programmer selects a test case based on personal preference. Commonly, bug reports are also converted into test cases that should capture the bug[2]. The level of compliance to the testing criterion – the code coverage – is not subject to measurement. Second, what kind of tool support is available to the testing process? Test suites for DocGen use the JUnit testing framework[3]. Additionally, these test suites are run automatically by means of a nightly build system.

5.2 Apache Ant

Ant is a build tool, and is being developed as a subproject of the Apache web server[4]. A build tool is used to automate many tasks related to the source code of a program, like compilation, execution and packaging. Many other tools exist that solve the same problem, including well-known Unix tools like Make. Ant aims at being portable, i.e. capable of running on multiple platforms, and at being easily extensible through the use of Java.

Ant's source code is large; about 6 megabytes, which contain roughly 170,000 lines of Java code. There are 887 classes contained in 87 packages. Again, our experiment uses the classes that have an associated test class; there are 111 such classes. The Ant source code is kept in a public CVS repository, which can be read by anyone. For our experiment we use source code from the 1.5.3 branch, dated April 16, 2003.

Since Ant is a subproject of the Apache Software Foundation[5], its development process is a derivative of the Apache project. In turn, the Apache project is a derivative of the popular open source model. Typically, an open source project consists of a number of contributors from around the world, who communicate and work together via the Internet. The open source model is not a full-fledged development methodology. Its main concerns are project management and adherence to a number of beliefs, including the free availability of source code. As such, there are few rules governing the

[2]'Bug' has approximately the same meaning as 'fault'.
[3]Information available at `http://www.junit.org`.
[4]Information available at `http://ant.apache.org`
[5]Information available at `http://www.apache.org`

actual development of Ant, save for a number of guidelines regarding the formatting of the code.

The testing process at the Ant project is similar to the testing process at the SIG. The programmers develop JUnit test cases during development, and run these tests nightly. Additionally, the functional correctness of the entire system is verified every night by running Ant in a typical production environment. There is no explicit testing criterion; test cases are created based on the preference of the programmers. Consequently, no measurement of the level of compliance to the testing criterion is done. Bug reports are again used as a source of test cases.

However, there is one major difference between the testing of Ant and the testing of DocGen. The developers of Ant did not start using the current testing procedure until late in the development process. [6] The DocGen development team applied their testing approach from the start of the development of DocGen. Hence, the classes of DocGen have been subject to a more homogeneous, while still ad-hoc, testing effort.

5.3 Results

In this section we present the results of the experiments described in Chapter 4. Tables 5.1 and 5.2 hold the results for DocGen and Ant, respectively. Both tables contain a left-hand sub-table, and a right-hand sub-table. Each row of the left-hand sub-table contains the values of Spearman's rank-order correlation coefficient (r_s) for source-based metric m and both test suite metrics dLOCC and dNOTC. Likewise, each row of the right-hand sub-table contains the statistical significance (p) of the t value derived from each r_s value (as described in Section 4.3).

The data sets used to compute the results are displayed by scatter plots in Appendix A.

$r_s(m,n)$	dLOCC	dNOTC
DIT	-.03664592	-.05901525
FOUT	.55386912	.45727837
LCOM	.16598053	.20712411
LOCC	.51296391	.51827393
NOC	-.02741486	.00241445
NOF	.24776041	.23295498
NOM	.35544872	.40061438
RFC	.53718219	.51978219
WMC	.42194549	.45878546

$p(m,n)$	dLOCC	dNOTC
DIT	.66958345	.49172625
FOUT	1.835e-12	1.718e-08
LCOM	.05170138	.01478843
LOCC	1.254e-10	7.479e-11
NOC	.74958827	.97757806
NOF	.00339098	.00596598
NOM	.00001884	1.123e-06
RFC	1.102e-11	6.447e-11
WMC	2.544e-07	1.521e-08

Table 5.1: Measurement results for DocGen.

[6] Personal communication.

$r_s(m,n)$	dLOCC	dNOTC
DIT	-.04563442	-.20101756
FOUT	.46451949	.30677602
LCOM	.43654512	.38164878
LOCC	.50017995	.32537468
NOC	.05374263	-.02614728
NOF	.45469619	.29441482
NOM	.53194369	.36896505
RFC	.52615509	.34120996
WMC	.53065411	.34769972

$p(m,n)$	dLOCC	dNOTC
DIT	.63436382	.03438547
FOUT	2.811e-07	.00105723
LCOM	1.670e-06	.00003581
LOCC	2.276e-08	.00049297
NOC	.57533668	.78530921
NOF	5.352e-07	.00170940
NOM	1.879e-09	.00006753
RFC	3.018e-09	.00024754
WMC	2.090e-09	.00018465

Table 5.2: Measurement results for Ant.

Based on these results, we evaluate hypotheses $\mathrm{H}_0(m,n)$ and $\mathrm{H}_1(m,n)$. These were defined in Chapter 4, but are repeated here for ease of reference:

$\mathbf{H}_0(m,n)$: There is *no* association between source-based metric m and test suite metric n,

$\mathbf{H}_1(m,n)$: There is an association between source-based metric m and test suite metric n,

where m ranges over our set of source-based metrics, and n is either the dLOCC or dNOTC of associated test suites. By definition of r_s, and correlation measures in general, if two variables are independent, i.e. there is no association between them, then $r_s = 0$. Thus if our results show that if $r_s(m,n) \neq 0$ for some m and n, then there is an association between m and n. In other words, if $r_s(m,n) \neq 0$, we can reject $\mathrm{H}_0(m,n)$ and accept the converse, $\mathrm{H}_1(m,n)$. The statistical significance $p(m,n)$ indicates the probability that the observed value of $r_s(m,n)$ is a chance event. Therefore, if the value of $p(m,n)$ is low, we can confidently reject $\mathrm{H}_0(m,n)$, and accept $\mathrm{H}_1(m,n)$. We can reject $\mathrm{H}_0(m,n)$ at a certain *confidence level* of x, if $1 - p(m,n) < x$.

For DocGen we can reject $\mathrm{H}_0(m,n)$ and accept $\mathrm{H}_1(m,n)$ at the 99% level of confidence for source-based metrics FOUT, LOCC, NOF, NOM, RFC and WMC, and both test suite metrics dLOCC and dNOTC. For Ant we can reject and accept the same hypotheses at the 99% confidence level, and additionally reject $\mathrm{H}_0(m,n)$ and accept $\mathrm{H}_1(m,n)$ for source-bases metric LCOM and both test suite metrics. The source-based metrics which are significantly correlated to the test suite metrics at the 99% confidence level, are set in boldface in Tables 5.1 and 5.2. Some of the source-based metrics are significantly correlated with one of the test suite metrics if the confidence level is lowered to 95%. For DocGen the LCOM metric is significantly correlated with dNOTC at the 95% level of confidence, while for Ant the correlation between DIT and dNOTC is significant at the 95% level of confidence.

5.4 Discussion

In addition to the measurement results of Tables 5.1 and 5.2, we calculated the correlations among the source-based metrics themselves. These correlations are contained in Table A in Appendix A. A first observation is that many of the source-based metrics are correlated among each other. Furthermore, both systems appear to have groups of metrics that are all strongly correlated to each other, and not to metrics outside their group. For DocGen, the metrics FOUT, LOCC, NOM, RFC and WMC form such a group. Likewise, Ant has a group consisting of the metrics FOUT, LOCC, NOM, NOF, RFC and WMC.

Second, we observe that none of the source-based metrics yields a much better correlation with the test suite metrics than the LOCC metric (see Tables 5.1 and 5.2). Indeed, according to Hotelling's t test for the difference between two correlation coefficients [11], none of our source-based metrics is a significantly better predictor of the test suite metrics than LOCC. It should be noted that the use of the term 'predictor' does not imply that the metrics are capable of predicting absolute values. Due to our use of Spearman's rank order correlation coefficient, a correlation between metrics indicates the ability of one metric to predict the *rank* of the value of the other metric. Hotelling's t is defined as:

$$t = (r_{yz} - r_{xz})\sqrt{\frac{(n-3)(1+r_{xy})}{2((((1-r_{xy}^2) - r_{xz}^2) - r_{yz}^2) + 2r_{xy}r_{xz}r_{yz})}}$$

where r_{xy} denotes Spearman's correlation coefficient between metrics x and y. For example, to calculate whether the correlations between FOUT and dNOTC, and RFC and dNOTC are significantly different, we calculate t for r_{yz}, r_{xz} and r_{xy} equal to $r_s(\mathsf{RFC}, \mathsf{dNOTC})$, $r_s(\mathsf{FOUT}, \mathsf{dNOTC})$ and $r_s(\mathsf{FOUT}, \mathsf{RFC})$, respectively. n represents number of samples, so in the case of DocGen $n = 138$. The difference between the correlations is significant if the absolute value of t is greater than the critical value for the number of samples, n, and the desired confidence level. For the confidence level of 99% and our number of samples, the critical value of t is 2.576. The sign of the value of t determines whether the metric x is a better or worse predictor than metric y. A positive value of t which is greater than the critical value makes x a significantly better predictor than y. Conversely, a negative value of t that is smaller than the critical value with a negative sign makes x a significantly worse predictor than y. Hotelling's t values for the source-based metrics versus LOCC are displayed in Table 5.3.

A number of our source-based metrics, while not significantly better predictors than LOCC, are not significantly worse either. For DocGen the most notable are the FOUT and RFC metrics (the WMC metric predicts only dNOTC equally well as LOCC). For Ant we find a larger set, consisting of FOUT, LCOM, NOF, NOM, RFC and WMC.

DocGen	dLOCC	dNOTC
DIT	-4.953	-5.219
FOUT	0.744	-1.069
LCOM	-4.218	-3.784
NOC	-5.231	-5.597
NOF	-3.405	-3.676
NOM	-2.896	-3.908
RFC	0.729	0.045
WMC	-3.362	-2.177

Ant	dLOCC	dNOTC
DIT	-5.046	-4.577
FOUT	-1.015	-0.485
LCOM	-0.692	0.554
NOC	-3.849	-2.774
NOF	-0.776	-0.480
NOM	0.653	0.810
RFC	0.949	0.524
WMC	1.670	1.104

Table 5.3: Hotelling's t values for the source-based metrics versus LOCC.

Third, we observe that some of the source-based metrics do a better job at predicting one of the test suite metrics over the other. Using Hotelling's t test, we determine which source-based metrics do a significantly better job at predicting dLOCC over dNOTC. For DocGen, the correlation between the test suite metrics dLOCC and dNOTC is 0.838, while for Ant that correlation is 0.767. Now let r_{yz}, r_{xz} and r_{xy} be equal to $r_s(\text{dLOCC}, z)$, $r_s(\text{dNOTC,z})$ and $r_s(\text{dLOCC, dNOTC})$, and z ranges over the source-based metrics. Then we find that, for DocGen, the FOUT metric is a significantly better predictor of dLOCC over dNOTC, but only at the 95% confidence level. The t value is 2.367, while the critical value for 138 samples at the 95% confidence level is 1.960. No other metrics are significantly better predictors of dLOCC over dNOTC for DocGen. For Ant, the source-based metrics that predict dLOCC significantly better over dNOTC are: FOUT, LOCC, RFC, NOF, NOM and WMC. Their t values are 2.722, 3.090, 3.332, 2.752, 2.937 and 3.306, respectively. Note that they are all significant at the 99% level, i.e. the t values are larger than 2.576.

Fourth, Figure A.2 shows that, for DocGen, the FOUT metric does a good job at identifying the class which has the largest dLOCC and dNOTC values for its associated test class. The class in question is the CobolModel class, which is identified in the same way by the LOCC, NOM, RFC and WMC metrics. In other words, the class with the highest fan out, number of lines of code, number of methods, response set size and weighted methods number, is associated with the test class which has both the highest number of lines of code and number of test cases. Note that the CobolModel class was mentioned as being hard to test during interviews at the SIG (see Appendix B), which were performed prior our measurements. On the other hand, the QueuedComponent class was also mentioned during the interviews as being hard to test. However, its values for the metrics are not much different from the averages, hence our metrics do not identify the QueuedComponent class. Apparently the DocGen developers consider the QueuedComponent class to be badly testable for reasons that are not captured by our set of metrics.

However, Figure A.11 does not show the same result for Ant. None of the metrics that identify the class with the largest test class for DocGen, are capable of identifying that class for Ant. Furthermore, these metrics appear to have many more outliers in the case of Ant compared to DocGen. The fact that the unit testing effort of Ant was started late in its development may explain this difference, because, on the one hand, starting from the moment unit testing became needed, developers would probably have started to develop classes with testing in mind. On the other hand, the classes that existed before unit testing was introduced have probably not been exposed to the same amount of testing as the classes that were developed afterwards. In other words, the implicit testing criterion is likely different for classes that were created before testing was started, and classes that were created afterwards. Furthermore, the nature of the project itself may have an influence. The Ant project is an open source project, consisting of people working at very different locations. In contrast, the SIG development team works together at one location, and hence is more likely to have better coordination and cooperation. Also, practices like test-first development and pair programming guarantee that test cases are developed more-or-less similarly for every class of DocGen.

Finally, we discuss the source-based metrics individually.

Depth Of Inheritance Tree (DIT)

For both case studies, the results show no significant correlation between the DIT metric and both test suite metrics dLOCC and dNOTC. Would a testing criterion have caused the DIT metric to measure a test case generation factor, and assuming that the test suites satisfied the testing criterion, we would have observed a correlation between the DIT metric and, at least, the dNOTC metric. We conclude that for our case studies the DIT metric does not measure a test case generation factor. Because there is also no correlation between DIT and the dLOCC metric, we conclude that DIT does not measure a test case construction factor either.

In Section 2.2.1 we discussed how inheritance can be a test case generation factor, i.e. influence the number of required test cases. For example, a testing criterion could require that all –inherited and newly defined– methods of a class are tested in the context of the class. As we concluded, no such criterion is in use for neither DocGen nor Ant.

Note that if we would allow our confidence level to go down to 95%, the observed correlation between the DIT and dNOTC metrics would become significant for the Ant system. However, the correlation is still a very weak one.

Fan Out (FOUT)

The FOUT metric has moderate and weak (but significant) correlation with the dNOTC (number of test cases) metric for DocGen and Ant, respectively. We conclude that the FOUT metric measures a test case generation factor of both systems. Since neither project has defined a testing criterion, we cannot confirm that the fan out of a class really is a test case generation factor. Our conclusion is thus limited to the observation that the implicit testing criteria of the Ant and DocGen projects cause the FOUT metric to measure a test generation factor.

We showed that FOUT is a significantly better predictor of the dLOCC metric than of the dNOTC metric (at the 95% confidence level for DocGen, 99% for Ant). Thus, the association between the fan out of a class and the size of its test suite is significantly stronger than the association between the fan out and the number of test cases. The fan out of a class measures the number of other classes that the class depends on. In the actual program, these classes will have been initialized before they are used. In other words, the fields of the classes will have been set to the appropriate values before they are used. When a class needs to be (unit) tested, however, the tester will need to take care of the initialization of the (objects of) other classes and the class-under-test itself. The amount of initialization required before testing can be done will thus influence the testing effort, and by assumption, the dLOCC metric.

We conjecture that the association between the FOUT and dLOCC metrics is a result of two factors. First, we saw before that for both systems the FOUT metric measures a test case generation factor, i.e. there is correlation between the FOUT and dNOTC metrics. The dNOTC and dLOCC metrics are also strongly correlated, which explains part of the association between FOUT and dLOCC. Second, the FOUT metric measures a test case construction factor, i.e. the amount of initialization required for testing.

Lack Of Cohesion Of Methods (LCOM)

The LCOM metric is associated to the test suite metrics in case of Ant, though not in the case of DocGen. Observe that Table A shows that for both systems the LCOM and NOF metrics are moderately correlated. In case of DocGen, the correlation is even fairly strong. Thus, it seems that for our case studies, classes that are not cohesive (high LCOM value) tend to have a high number of fields, and similarly, classes that are cohesive tend to have a low number of fields. Similar correlations exist between the LCOM and NOM metrics. Thus, in-cohesive classes tend to have a high number of fields and methods, and cohesive classes tend to have a low number of fields and methods. These effects are intuitively sound: it is harder to create a large cohesive class than it is to create a small one.

For DocGen, the correlations between the NOF and NOM metrics and the test suite metrics is significant but weak, while for Ant the correlations are moderate. We conjecture that for DocGen the lack of correlation between LCOM and the test suite metrics occurs because of the weak correlation between the NOF and NOM metrics and the test suite metrics. Conversely, because of a moderate correlation between the NOF and NOM metrics and the test suite metrics for Ant, we observe a moderate correlation between the LCOM metric and the test suite metrics.

Similar to the correlation between the DIT and dNOTC metric for Ant, the correlation between the LCOM and dNOTC metrics becomes significant for the DocGen system if the level of confidence is lowered to 95%. However, the correlation is still very weak.

Lines Of Code Per Class (LOCC)

For DocGen, the LOCC metric has moderately strong correlations with both the dLOCC and dNOTC metrics. For Ant, the correlation with dLOCC is also moderately strong, but the correlation with dNOTC is weak. Again, regarding the test case generation, we suffice by concluding that LOCC measures a test case generation factor of both DocGen and Ant. Because both projects lack an explicitly defined testing criterion, we are unable to confirm whether or not the lines of code of a class is the test case generation factor that we have measured.

The number of lines of code is a rather blunt aspect of the source code. As such, there is no clear direct link between the textual length of a class and the size of its test suite. However, the LOCC metric is correlated with many of the other source-based metrics (see Table A). For DocGen, strong correlations exist with FOUT, NOM, RFC and WMC, while for Ant the LOCC metric correlates strongly with FOUT, NOF, NOM, RFC and WMC. We conjecture that the correlations between LOCC and the test suite metrics are a result of the strong correlations between LOCC and the other metrics.

Number Of Children (NOC)

The NOC metric has no significant association with either test suite metric for both study systems. In the context of unit testing at the class level, the number of child classes of a class seems of little relevance to the testability. First, the child classes are tested by their own test classes. Second, any other effects of having child classes (polymorphism) are not of concern during testing of the parent class, but during the testing of classes that use the parent class. Objects of the child classes could be used instead of objects of the parent class, possibly requiring more testing. In any case, such effects lie outside of the scope of this thesis. We focus on the factors of a class that influence the required testing effort for that same class.

Number Of Fields (NOF)

The correlations between NOF and dNOTC are weak for both DocGen and Ant. By the same argument as for FOUT, we conclude that NOF measures a test case generation factor, but only weakly.

The fields of the class-under-test need to be initialized before testing can be done. We argued before that the amount of required initialization influences the testing effort and the dLOCC metric. Thus, we expect correlation between the NOF and dLOCC metrics. However, for DocGen the correlation we observe is only weak (but significant), while for Ant it is moderate. Neither is the correlation between NOF and dLOCC significantly better than the correlation between NOF and dNOTC for DocGen, though it is for Ant. A possible explanation is given by the definition of the NOF metric. In Chapter 4 $\mathsf{NOF}(c)$ is defined by $\mathsf{NOF}(c) = |F_D(c)|$. In words, $\mathsf{NOF}(c)$ is a count of the number of fields class c (newly) declares. The number of fields that class c inherits from its ancestors is therefore not included in the count. If classes tend to use fields they have inherited, the NOF metric may not be a sufficient predictor of the initialization required for testing. Whether or not this explains the difference between the observed correlations for DocGen and Ant remains subject of further research.

Additionally, for Ant there seems to be a fairly strong correlation between the NOF and FOUT metrics, while for DocGen that correlation is weak (see Table A). We showed how the FOUT metric could measure a test case construction factor; the amount of initialization required for testing. An alternate explanation for the moderate correlation between FOUT and dLOCC for Ant, and the weak correlation for DocGen, uses the strength of the correlation between the NOF and FOUT metrics. For Ant, the NOF metric is a fairly strong predictor of the FOUT metric, and vice versa. Furthermore, since we showed how the FOUT metric could predict the dLOCC metric, we could conclude that this link explains the strength of the correlation between NOF and dLOCC for Ant. Similarly, for DocGen the correlation between NOF and FOUT is weak, and as a result, the correlation between NOF and dLOCC is weak.

Number Of Methods (NOM)

The correlation between the NOM and dNOTC metrics is moderate for DocGen and weak for Ant. As before, we conclude that NOM measures a test case generation factor. The developers of DocGen intend to test each method a class implements, i.e. each method for which the class defines a method body (see Section 5.1). The extent to which each method should be tested is not defined. Because of this ‘guideline’ –it is not specific enough to qualify for a testing criterion– we would expect to see some correlation between NOM and dNOTC for DocGen.

We would not expect the number of methods of a class to measure a test case construction factor; the number of methods a class contains should not influence the effort needed to set up a test case. In the case of DocGen, the NOM metric does not have a significantly better correlation with dLOCC than with dNOTC. DocGen indeed lives up to the expectation. However, in the case of Ant, the difference between these correlations is significant. We offer two possible explanations. First, for Ant the correlation between the NOM and NOF metrics is strong (see Table A), i.e. the number of methods of a class is a strong predictor of the number of fields of a class. We saw before how the number of fields of a class can influence the effort needed to test, i.e. the dLOCC metric. Thus, the correlation between the NOM and dLOCC metrics for Ant could be explained indirectly via the NOF metric. The fact that the correlation between the NOM and NOF metrics is only moderate for DocGen confirms this explanation.

Second, for Ant we observe a similarly strong correlation between the NOM and FOUT metrics. By the same argument, the correlation between the NOM and dLOCC metrics for Ant could be explained indirectly via the FOUT metric. Again, the correlation between the NOM and FOUT metrics is only moderate for DocGen.

Response For Class (RFC)

The correlation between RFC and dNOTC is moderate for DocGen and weak for Ant. By the same argument as for FOUT, we conclude that RFC measures a test case generation factor.

In Chapter 4 we defined the RFC metric by $\mathsf{RFC}(c) = |M(c) \cup_{m \in M(c)} MI(m)|$, i.e. the RFC of c is a count of the number of methods of c and the number of methods of other classes that are potentially called by the methods of c. From the definition, it is clear that the RFC metric consists of two components. First, the number of methods of class c. The strong correlation between the RFC and NOM metrics for both systems is explained by this component. Second, the number of methods of other classes that are potentially invoked by the methods of c. The invocation of methods of other classes gives rise to fan out, hence the strong correlation between RFC and FOUT in both systems. Given the correlations between the RFC metric and both the NOM and FOUT metrics, the observed correlations between the RFC and dLOCC metrics for both DocGen and Ant are as expected.

Weighted Methods Per Class (WMC)

The correlation between the WMC and dNOTC metrics is moderate for DocGen and weak for Ant. By the same argument as for FOUT, we conclude that WMC measures a test case generation factor.

The WMC metric is defined in Chapter 4 by:

$$\text{WMC}(c) = \sum_{m \in M_{Im}(c)} \text{VG}(m),$$

where $\text{VG}(m)$ is McCabe's cyclomatic complexity number for method m. We observe that the WMC metric correlates strongly with the NOM metric for both DocGen and Ant (see Table A). Also, the relationships to the other metrics are very similar for both WMC and NOM. An explanation is offered by the fact that for both systems, the VG value of each method tends to be low, and close to the average. For DocGen, we have an average VG of 1.311, with standard deviation of 0.874 and maximum of 17. For Ant, we have an average VG of 2.141, with standard deviation of 2.910 and maximum of 61. Thus, for our systems the WMC metric will tend to measure the number of methods, i.e. the NOM metric. We conclude that the same effects explain the correlations with the test suite metrics for both WMC and NOM.

As a side note, the low average, standard deviation and maximum values of the VG of the methods of DocGen are a result of a coding standard in use at the SIG. According to the coding standard, each method should not contain more than 12 lines of code. In practice, the VG of a method is strongly correlated to the number of lines of code (WMC is also strongly correlated to LOCC), thus a low number of lines of code will indicate a low VG. In addition, the use of object-oriented features like polymorphism and dynamic binding helps to reduce the VG values of methods. Constructions like switches and abundant use of if–else statements, which both contribute to the VG value, can be avoided and replaced by use of polymorphism.

5.5 Conclusion

We found significant associations between metrics and the test suite size for a class for both our case studies. For DocGen, the metrics FOUT, LOCC and RFC have been shown to have a moderately strong association with the size of the test suite for a class. For Ant, a larger set of metrics qualified, consisting of the metrics FOUT, LCOM, LOCC, NOF, NOM, RFC and WMC. Conversely, the NOC and DIT metrics did not have significant associations with the test suite size, for neither case study.

We also found that none of the metrics is significantly more correlated to the test suite size than the LOCC metric. However, a number of metrics for both DocGen and Ant are not significantly less correlated, either. For DocGen the most notable are the FOUT and RFC metrics, while for Ant we have the metrics FOUT, LCOM, NOF, NOM, RFC and WMC. Apparently our metrics do not measure very specific factors of the source code, since a simple count of the lines of code is equally predictive. On the other hand, metrics like FOUT and RFC are still equally predictive of the test suite size, but also allow more insight into the cause of their correlations.

Chapter 6

Conclusions

In this chapter we summarize the contributions and limitations of the work presented in this thesis. Furthermore, we provide directions for future work.

6.1 Contributions

- In Chapter 2 we provided an overview of testability aspects of software development. These aspects were discussed and the topic of this thesis was put into context.
- Also in Chapter 2, we presented an initial model of testability with respect to source code factors. The model distinguishes between test case generation factors and test case construction factors. The former deals with source code factors that determine how many test cases need to be developed, while the latter deals with source code factors that influence the effort needed to develop individual test cases.
- Several approaches to testability assessment have been proposed in the literature. In Chapter 3 we discussed the fault sensitivity approach by Voas et. al., Freedman's controllability and observability notions, McGregor's visibility component and Jungmayr's approach to identifying test critical dependencies. At the end of that chapter, we discussed how these approaches relate to our own model.
- In Chapter 4 we turned towards testability metrics. Because of the match between the testability model described by Binder in [3] and our own model, Binder's testability metrics were taken as a starting point. We provided operational definitions of of the following metrics: Depth of Inheritance Tree (DIT), Fan Out (FOUT), Lack of Cohesion Of Methods (LCOM), Lines Of Code per Class (LOCC), Number Of Children (NOC), Number Of Fields (NOF), Number Of Methods (NOM), Response For Class (RFC) and Weighted Methods per Class (WMC).

- We evaluated the metrics by means of empirical study. Following the definition of the metrics, we used the GQM/MEDEA framework to set up our experiment. We have chosen to estimate the dependent variable in the experiment, i.e. the testing effort for a class, by measuring the size of the test suite associated with the class.

- At the end of Chapter 4, we provided details on the implementation of the tool we have used to perform the measurements. The tool is based on a plug-in for the Eclipse tool platform, and is capable of measuring a large set of metrics for Java code. The Eclipse metrics plug-in will be employed as part of a software analysis toolkit currently under development at the Software Improvement Group.

- In Chapter 5, we applied our experiment to two diverse case studies. The first is DocGen, a commercial source code documentation tool which is being developed by the Software Improvement Group. The second is Apache Ant, a widely-used automation tool for software development, which is being developed under the open source model as part of the Apache web-server project.

- The results of the case studies were used to perform a statistical evaluation of the metrics. The correlations between the metrics and the test suite size, measured by Spearman's rank-order correlation coefficient, were discussed at the end of Chapter 5. The discussion was based on the model of testability presented in Chapter 2, and took into account how external factors could have influenced the results.

- Finally, for both our case studies, we found significant associations between our metrics and the test suite size for a class. For the DocGen system, the metrics FOUT, LOCC and RFC have been shown to have a moderately strong association with the size of the test suite for a class. For the Ant system, a larger set of metrics qualified, consisting of the metrics FOUT, LCOM, LOCC, NOF, NOM, RFC and WMC. Conversely, the NOC and DIT metrics did not have significant associations with the test suite size, for neither case study.

In summary, we have seen that for two very different systems, a subset of our metrics is capable of assessing the testing effort required for a class. Furthermore, we have seen how our initial model of testability can be used to explain the results. On the one hand, the results have shown us that many of the metrics measure test case generation factors that are due to implicit testing criteria. On the other hand, some of the metrics also seem to measure test case construction factors. Most notably, the results allow for explanations of the FOUT, NOF and RFC metrics in terms of test case construction factors. Conversely, we have seen that the NOC and DIT

metrics do not measure factors of testability. We conclude that the results will allow us to improve both the set of metrics and the model of testability.

6.2 Limitations

- Our work considers testability from the perspective of unit testing at the class level. While this is a useful approach to testing, in practice many other approaches are used (see Section 1.2). For example, the Eclipse tool framework is tested from a more functional perspective, i.e. 'units of functionality' are the subjects of testing, instead of syntactical elements of the source code. As a side note, the Eclipse project employs the same testing framework (JUnit) as our case studies, yet the approach to testing is very different.

- In our discussion of the measurement results, we found that none of the metrics is significantly more correlated to the test suite size than the relatively simple Lines Of Code per Class (LOCC) metric. However, a number of metrics for both DocGen and Ant are not significantly less correlated, either. Apparently our metrics do not measure very specific factors of the source code. On the other hand, the fact that the LOCC metric correlates with the test suite size provides little insight in the source code factors that cause the correlation. Metrics like FOUT and RFC are still equally predictive of the test suite size, but also allow more insight into the cause of their correlations.

- For our statistical evaluation of the metrics we have used a non-parametric statistic, i.e. Spearman's rank-order correlation coefficient. Non-parametric statistics are applicable when one lacks knowledge of the distribution of the measurement data. As a result, the correlations we observed in the data indicate the predictive capability of the metrics with respect to the ranking of classes only.

- The systems we studied were both written in Java, and tested using the JUnit testing framework. As such, the validity of our results is further limited to systems using these technologies.

6.3 Future Directions

- To expand the perspective of testability used in this thesis, the Eclipse tool framework is a suitable candidate for a case study. The functional approach to testing employed by the Eclipse project is common in larger software development organizations. On the more practical side, the Eclipse project has a very open nature: source code, bug databases and communication channels are all available to the public.

- The results of our work can be used to guide the refinement of both our initial model of testability, and the metrics themselves. A possible approach is to analyze the source code of outliers in our measurement data. The FOUT and RFC metrics appear to be suitable candidates for such further investigation.

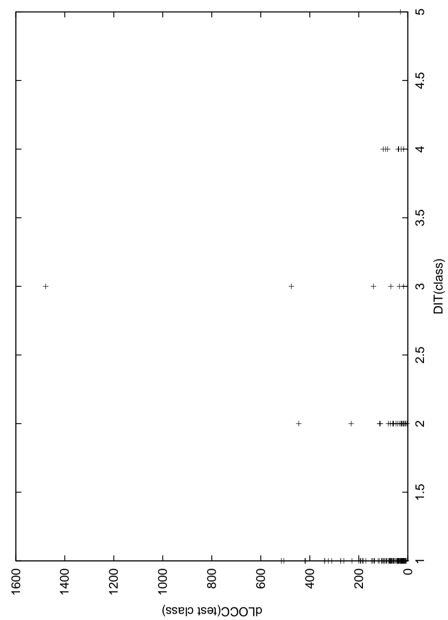
- The use of more powerful statistics than Spearman's rank-order correlation is needed to further assess the predictive capabilities of the metrics. In section 4.3 we mentioned work by Wheeldon and Counsell [28], which researches the distribution of measurement data of object-oriented systems. They show that many coupling metrics are distributed according to a power law distribution. Since metrics like FOUT and RFC are related to coupling metrics, they may be distributed similarly. Consequently, more powerful statistics could be used to assess them.

- Finally, the validity of our results could be explored for systems written in other object-oriented languages. The definitions of our metrics contain only a limited amount of dependency on the programming language of our case studies.

Appendix A

Results Of The Case Studies

This appendix contains the intermediate measurement results for both case studies. Each source-based metric is plotted against both test suite metrics dLOCC and dNOTC. The samples are those system classes that have an associated test class. A table of the Spearman correlation coefficients for each pair of metrics and both case studies is included at the end of this appendix. The results contained in this appendix are analyzed and discussed in Chapter 5.



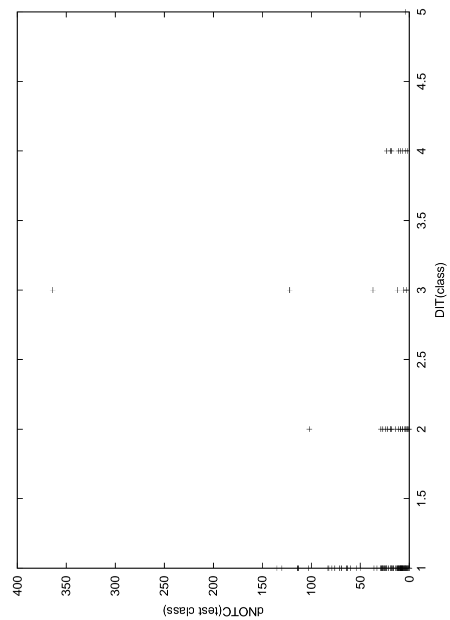
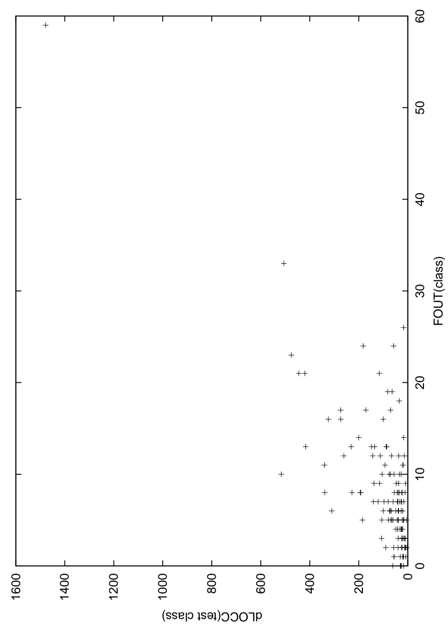


Figure A.1: DocGen: DIT



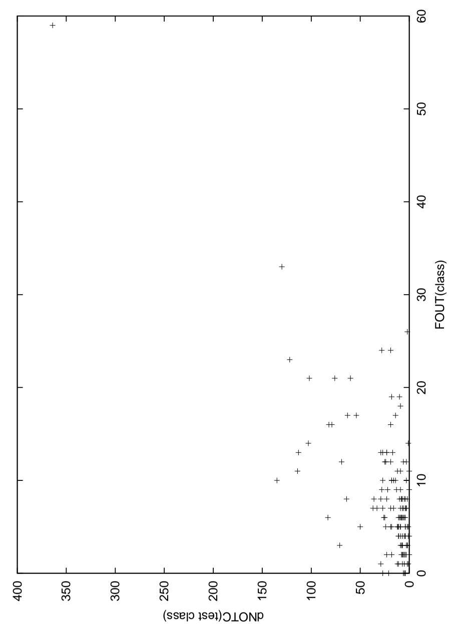
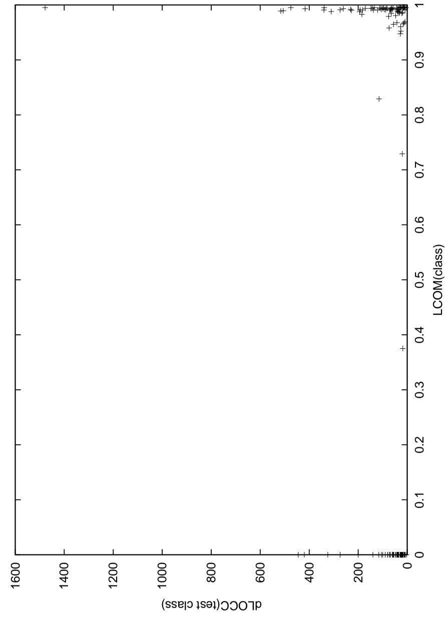


Figure A.2: DocGen: FOUT



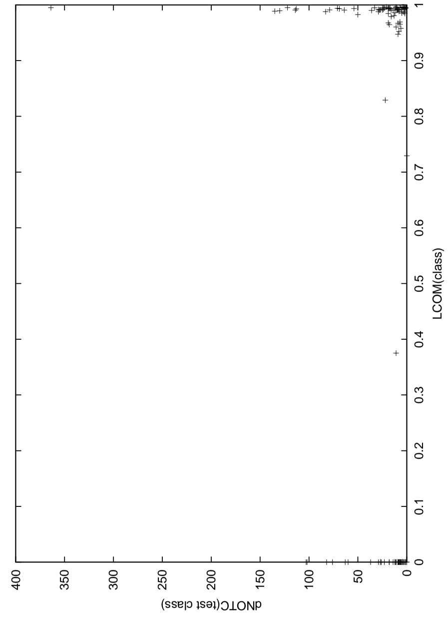
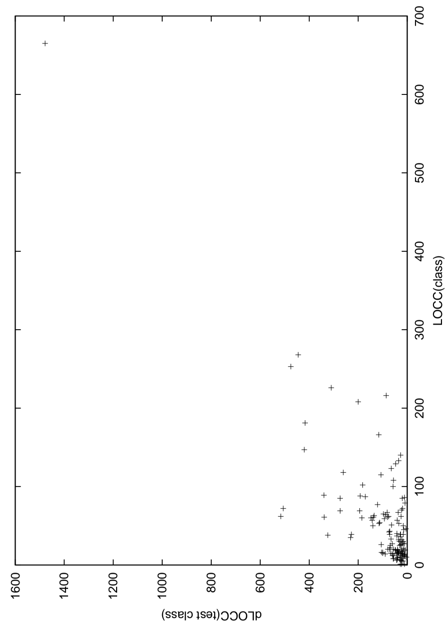


Figure A.3: DocGen: LCOM



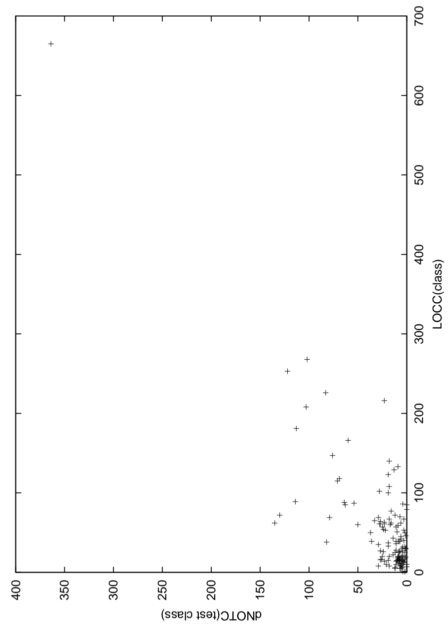
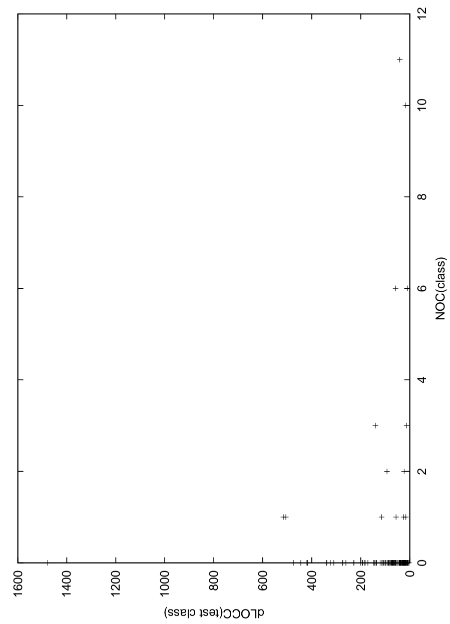


Figure A.4: DocGen: LOCC



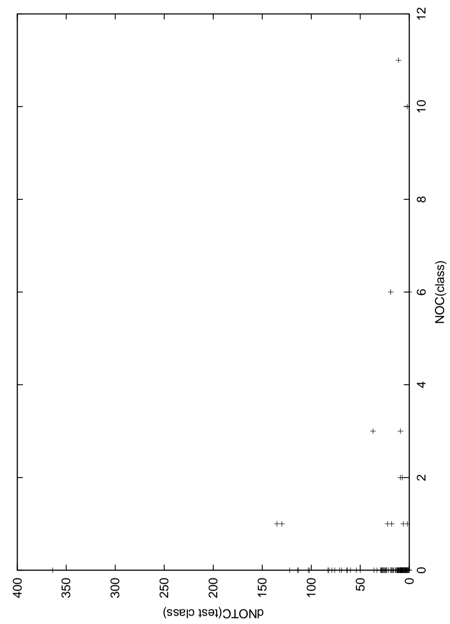
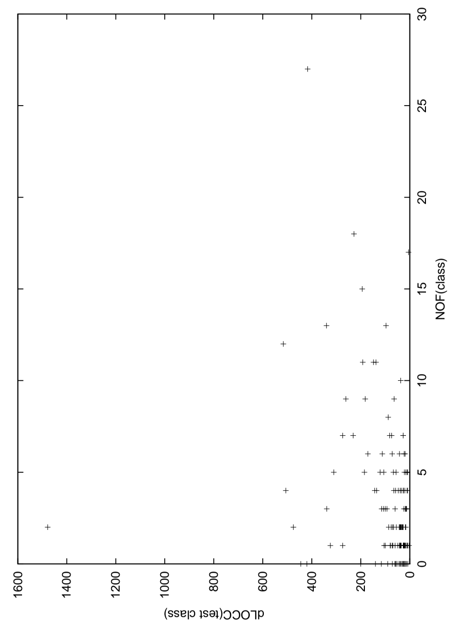


Figure A.5: DocGen: NOC



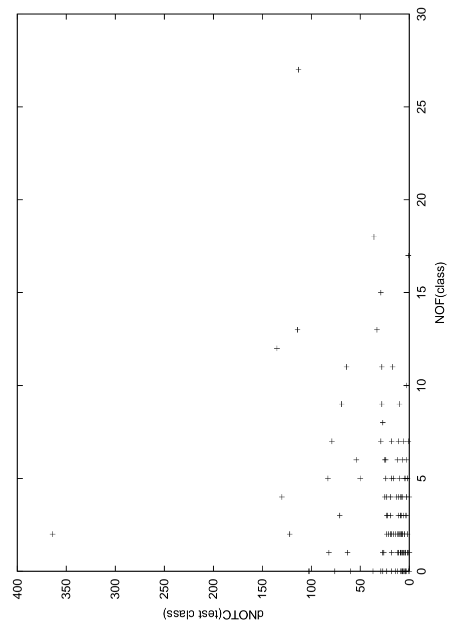


Figure A.6: DocGen: NOF

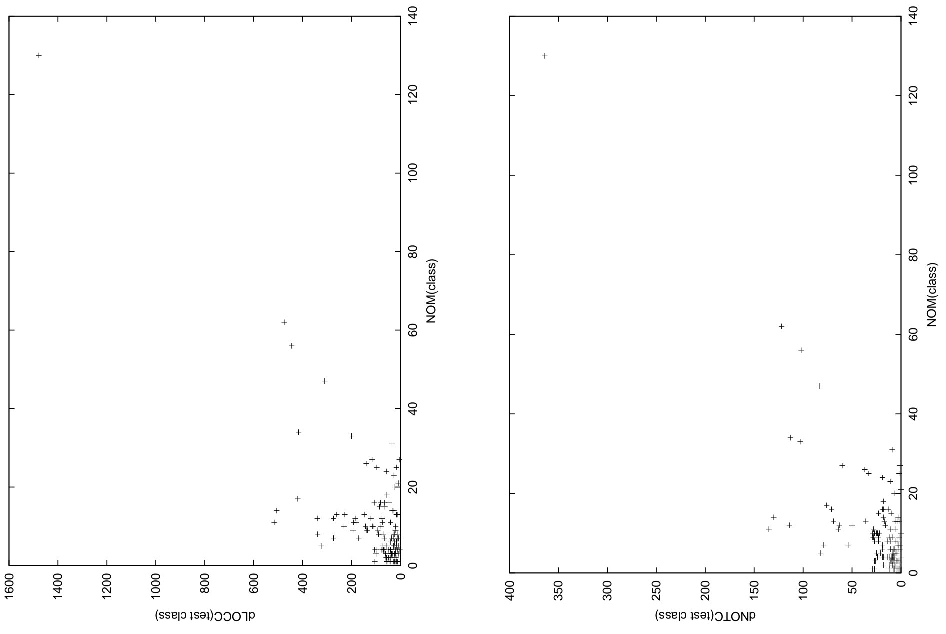


Figure A.7: DocGen: NOM

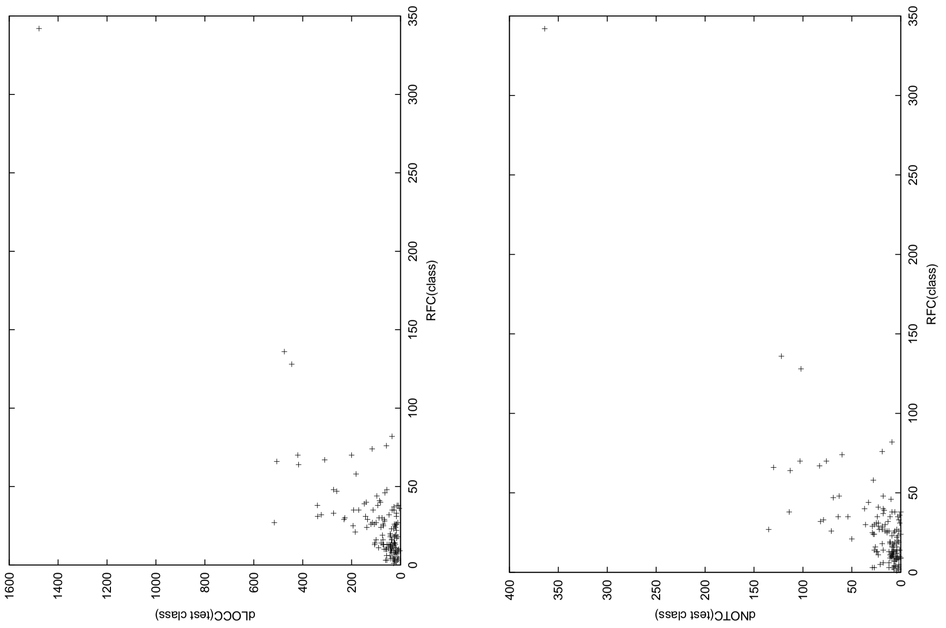
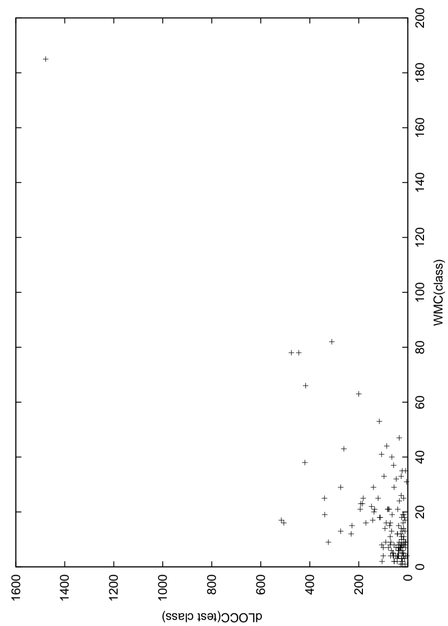


Figure A.8: DocGen: RFC



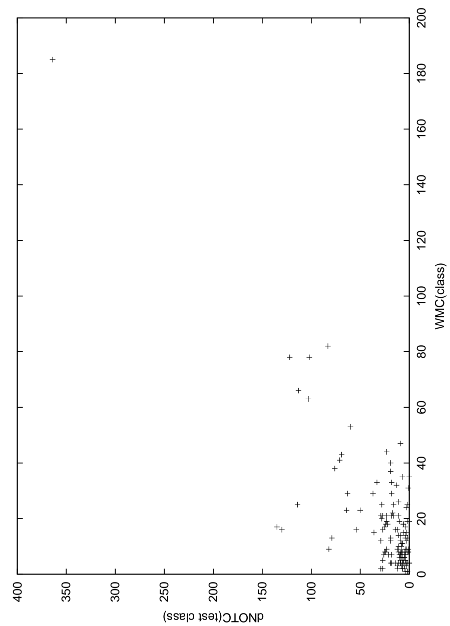
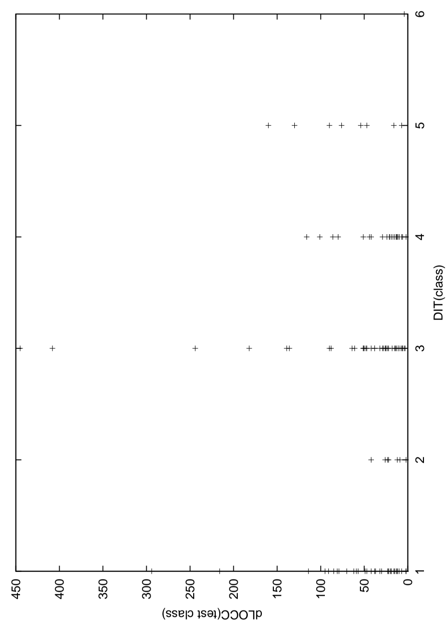


Figure A.9: DocGen: WMC



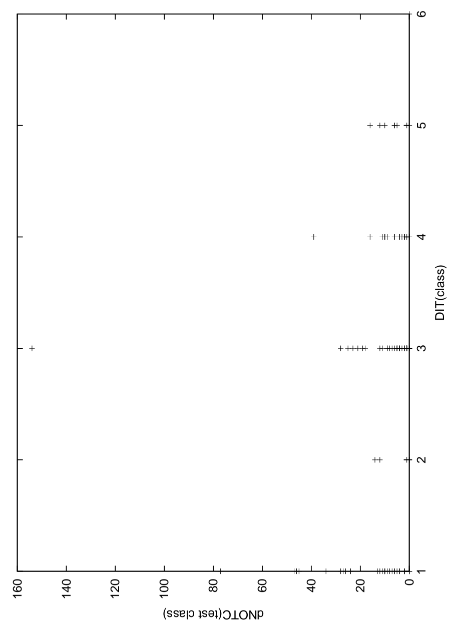
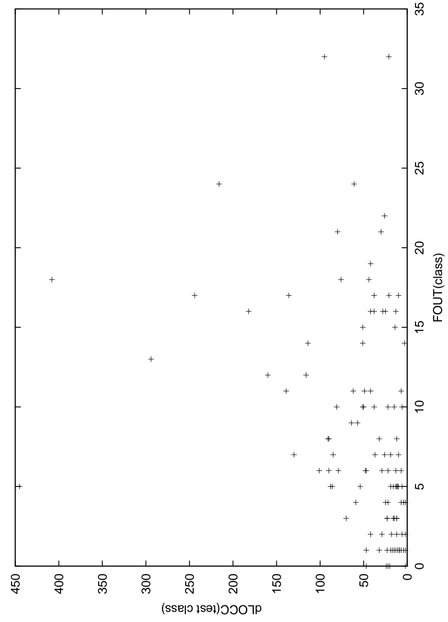


Figure A.10: Ant: DIT



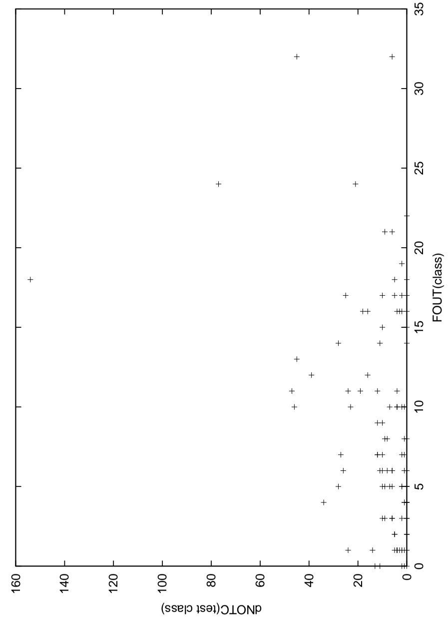
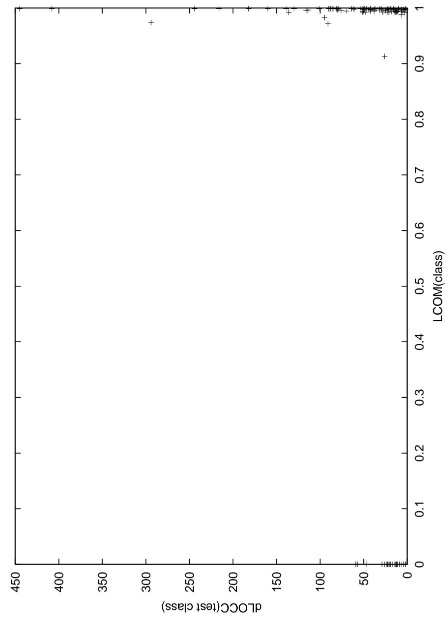


Figure A.11: Ant: FOUT



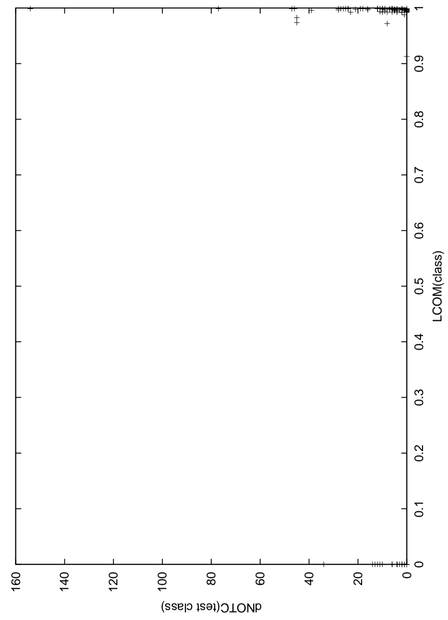
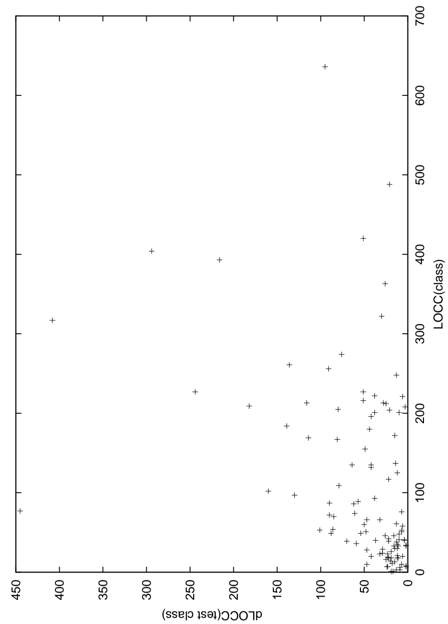


Figure A.12: Ant: LCOM



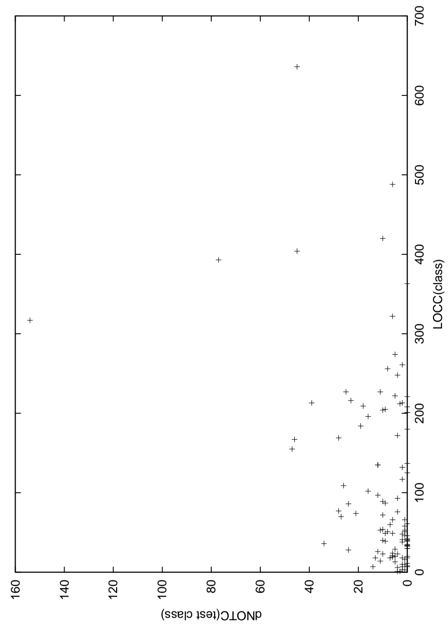
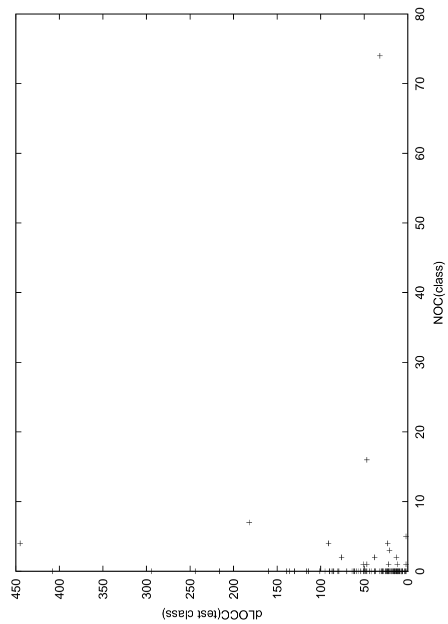


Figure A.13: Ant: LOCC



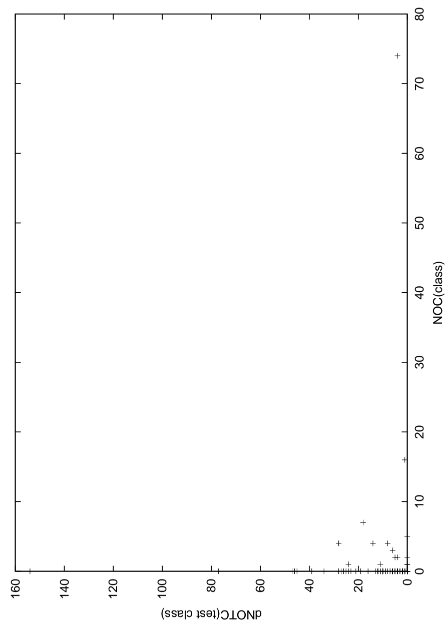


Figure A.14: Ant: NOC

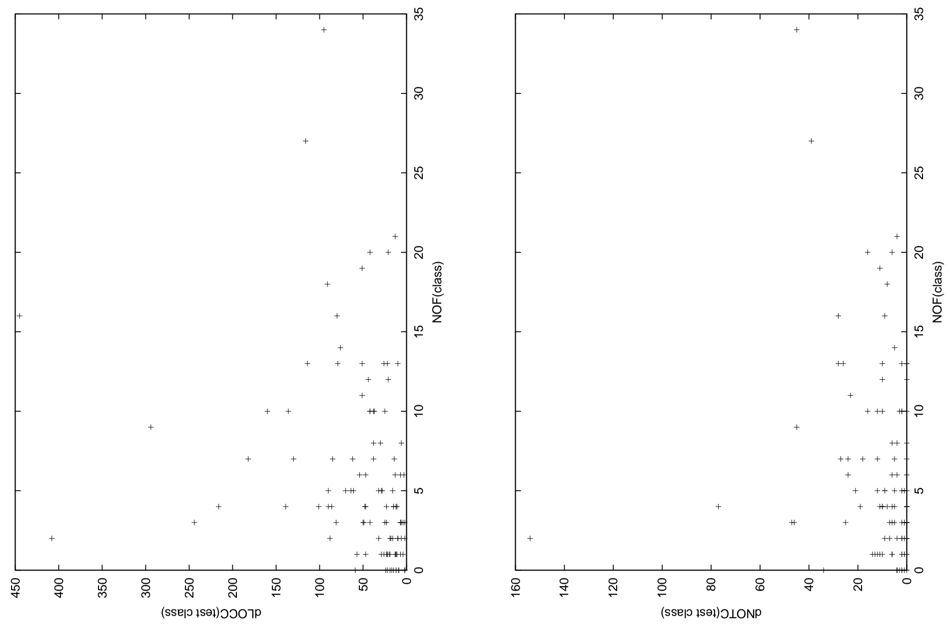


Figure A.15: Ant: NOF

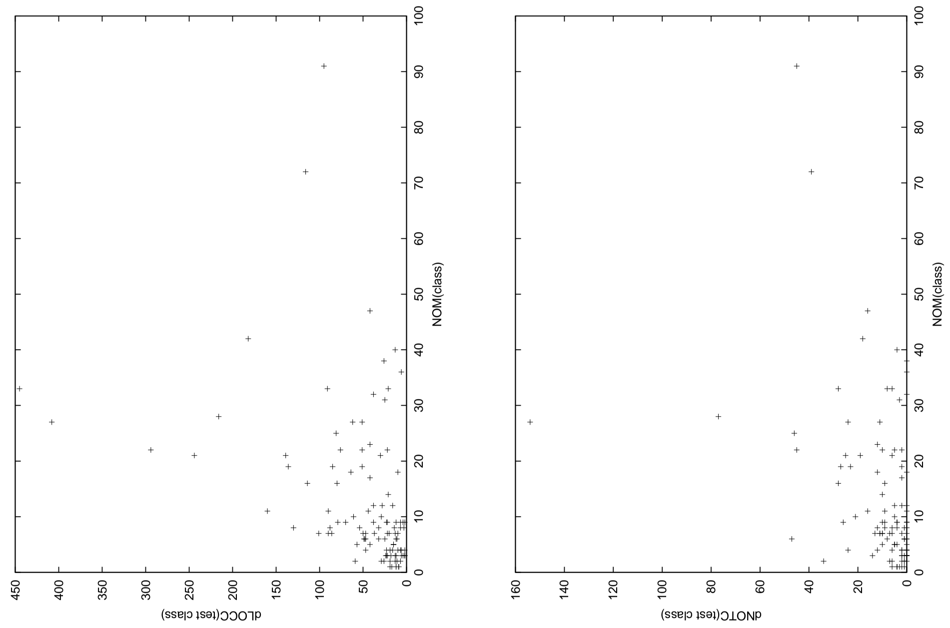
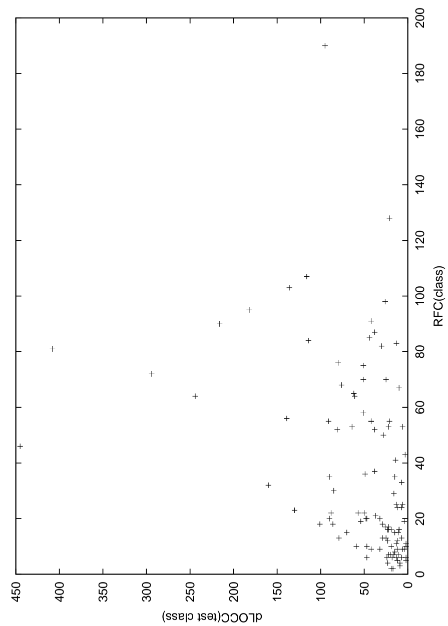


Figure A.16: Ant: NOM



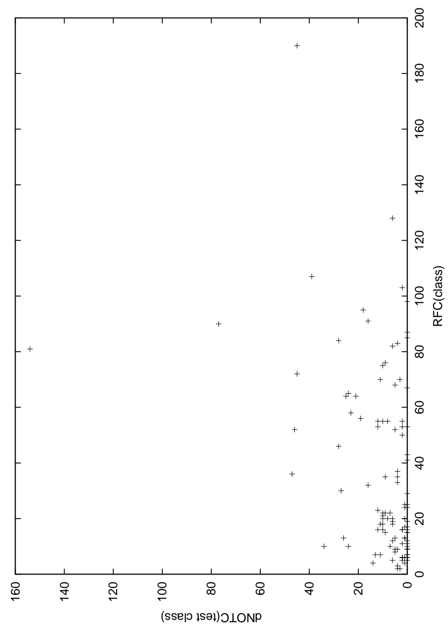
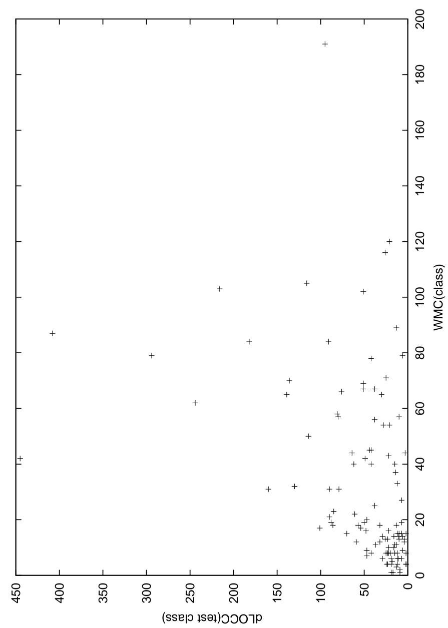


Figure A.17: Ant: RFC



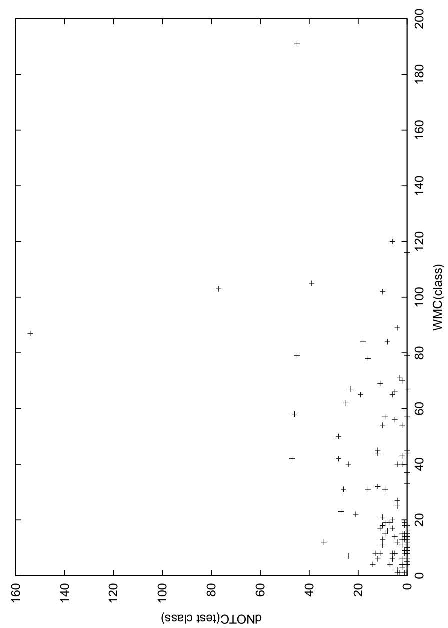


Figure A.18: Ant: WMC

DocGen	DIT	FOUT	LCOM	LOCC	NOC	NOF	NOM	RFC	WMC
DIT	1								
FOUT	-.0288628	1							
LCOM	-.0878565	.3170920	1						
LOCC	-.1291266	.6914187	.3793173	1					
NOC	.0511100	.0506100	.0134258	.2039079	1				
NOF	-.2518776	.3649002	.7655619	.4439602	.0520108	1			
NOM	.0749825	.5458229	.4806950	.8465885	.2263870	.4430824	1		
RFC	-.0067225	.8368887	.4198502	.8944888	.1839234	.4320281	.8669229	1	
WMC	-.0350667	.5790627	.4364944	.9306958	.2084994	.4213003	.9515929	.8786371	1

Ant	DIT	FOUT	LCOM	LOCC	NOC	NOF	NOM	RFC	WMC
DIT	1								
FOUT	.1197494	1							
LCOM	.0201313	.3063668	1						
LOCC	.1415034	.9110308	.3105703	1					
NOC	-.0761769	-.0793997	-.0722794	.0289468	1				
NOF	.1049955	.6856326	.4618926	.7473582	.1090735	1			
NOM	.0086915	.7040327	.4356426	.8185809	.2155809	.8247434	1		
RFC	.1504100	.9286402	.3443800	.9440796	.0229394	.7886552	.8606042	1	
WMC	.1256072	.8645581	.3541613	.9747772	.0963067	.7844764	.8987452	.9452133	1

Table A.1: Spearman correlations between the metrics.

Appendix B

Report On The Interviews

B.1 Introduction

The following reports on a series of 6 interviews done at the Software Improvement Group. All interviewees were involved with the development of the DocGen program. The interviews were performed with two goals in mind. First, we wanted to obtain an intuitive notion of test difficulties as they occur in practice. Second, to gather information about the DocGen program and its development process. The description of the DocGen case study in Section 5.1 is based on this report.

As a side note, we also performed interviews at several different companies. One interview was done at Tryllian, focusing on their Agent Development Kit (ADK). Three more interviews were done at Epictoid, which focused on their development of animation software. Finally, an interview was done to explore the testing efforts of the ABN-AMRO bank. Since no additional case studies resulted from these interviews, their results have not been included in this report.

B.2 Results

1. Which module is the most difficult to test? And which the easiest?

    - How would you describe the module?
    - Is the module modified frequently? Or infrequently?

    ▷ The modules that were mentioned as being the hardest to test (in order of frequency):

    - presentation package,
    - QueuedComponent class,
    - command package,

- code dealing with delete rules.

▷ The modules that were mentioned as being the easiest to test (in order of frequency):

- extract package,
- util package,
- regular expressions,
- model package.

2. Which module has the most test cases? And which the least?

   - How would you describe the module?
   - Is the module modified frequently? Or infrequently?

   ▷ The modules that were mentioned as having the most test cases (in order of frequency):

   - extract package,
   - CobolFile class,
   - CobolModel class,
   - QueuedComponent class,
   - ExtractSQLParsers class,
   - code dealing with delete rules.

   ▷ The modules that were mentioned as having the least test cases (in order of frequency):

   - presentation package.

3. Which module has the most complex test cases? And which the least complex?

   - How would you describe the module?
   - Is the module modified frequently? Or infrequently?

   ▷ The modules that were mentioned as having the most complex test cases (in order of frequency):

   - command package,
   - CobolModel class,
   - QueuedComponent class,
   - Repository class.

▷ The modules that were mentioned as having the least complex test cases (in order of frequency):

- util package,
- model package,
- database package,
- regular expressions.

4. Which module reveals the most bugs during testing? And which the least?

   - How would you describe the module?
   - Is the module modified frequently? Or infrequently?

   ▷ The modules that were mentioned as having the highest defect ratio (in order of frequency):

   - command package,
   - QueuedComponent class,
   - CobolModel class,
   - updateSourceCommand class.

   ▷ The modules that were mentioned as having the lowest defect ratio (in order of frequency):

   - extract package,
   - model package,
   - regular expressions.

5. Does your process prescribe a specific test strategy?

   - If yes: Could you describe the test strategy?
   - If no: Do you apply a personal test strategy?

   ▷ There is no prescribed test strategy, however in practice the following is done:

   - A test case is written for every method.
   - Test cases are generated in an ad-hoc manner.
   - Test cases are derived from bug reports.

6. Does your process measure the coverage of test cases?
   If yes:

   - Which module has the best coverage? And which the worst?

- How would you describe the module?
- Is the module modified frequently? Or infrequently?

▷ No.

7. Does your process prescribe a coding standard?

   - If yes: Could you provide a brief description of the coding standard?
   - If yes: Does your process verify the application of the coding standard?
   - If no: Do you apply a personal coding standard?

   ▷ Yes, there is a coding standard in use.

   - The Sun Java conventions are used for variable naming, among other things.
   - A method has at most 12 statements.
   - A method has a single return statement.
   - Every class must be accompanied by a test class.
   - Blocks of statements are always surrounded by curly braces ('{' and '}').

   Violations of the coding standard are not automatically detected, but practices like code reviews and pair programming impose some control.

8. Does your team use test automation tools? (JUnit, for example)

   - If yes: Are the tests being run periodically?

   ▷ The JUnit testing framework is used. All tests are run automatically following the nightly build.

9. Are test results for every module available to every team member?

   ▷ Every team member is capable of running all the tests, and view their results. The results of the nightly build are publicly available on the local network.

10. Have the following cases caused you trouble during testing? Have you ever modified the code to alleviate the problems? If yes: Could you describe your modifications briefly?

    - a method containing many local variables:
      ▷ Yes: 0, No: 6.

- a method containing many conditional statements (IF-ELSE, for example):
  ▷ Yes: 2, No: 4. Those answering yes said they split up their methods.
- a method causing many side-effects:
  ▷ Yes: 1, No: 5.
- a method has complex objects for its arguments and return value:
  ▷ Yes: 1, No: 5.
- a class that depends on many other classes (through inheritance or composition, for example):
  ▷ Yes: 1, No: 5.
- a class containing many private fields.
  ▷ Yes: 4, No: 2. Private fields and methods are often changed to a less restrictive access level.

11. Do you test exception handling code?

    - If no: Could you indicate why?

    ▷ In general, exception handling is not tested. Several reasons have been given:

    - The exception that was caught is simply thrown on.
    - The occurrence of an exception is recorded in a log file and subsequently ignored by the code.
    - The exception handling code is very simple.
    - Some exceptions are not expected to occur at all.

12. Do you (occasionally) define new types of exceptions?

    - If yes: Do you test the handling of exceptions of this kind?
      - If no: Could you indicate why?

    ▷ Custom exceptions are in use, but their handling is not different from the predefined exceptions.

13. Could you indicate your level of (professional) experience with using object oriented programming languages?
    –

14. Could you indicate your level of (professional) experience with using other programming languages?
    –

15. Compared to testing of code written in other languages, do you think object orientation influences the difficulty of testing?

- If yes: Could you explain the difference in some detail?

▷ Both answers have been given, supported by different arguments. Arguments in favor of object orientation making testing harder:

- Encapsulated data is harder to inspect.
- Dynamic bind makes it harder to figure out what piece of code is executed.
- Object oriented languages rely heavily upon side effects.
- Object equality is an added concern which the tester needs cope with.
- Object orientation allows for high levels of abstraction, which possibly makes the interpretation of test results a difficult task.

In favor of easier testing using object oriented languages:

- Better use of abstraction allows for hiding details.
- Object orientation generally results in modular software, consisting of modules that can be tested separately.
- Maintenance of the test cases can also benefit from OO-features like inheritance.

16. Could you describe a change to your current project that would improve the testing effort?

    ▷ The following suggestions were made to improve the testing effort:

    - The installation of tools capable of supporting the testing of the user interface.
    - Having the application generate an intermediate output format, which contains a minimum amount of formatting information.
    - Refactor the code to obtain a better modular structure.
    - To reduce the effort of setting up test cases, general test cases should be written that take care of repetitive tasks. New test cases can then be derived using inheritance.
    - The use of a bug tracking system.
    - Increasing the runtime performance of the test cases will make running the complete set of tests frequently a viable practice.
    - Code responsible for processing and code doing database access should be be separated in a better way.

B.3 Discussion

The results for questions 2 to 5 should point out those modules that are hard (and easy) to test. First, the *presentation* package was named during each interview as both being hard to test, and lacking any automated test cases. This package is responsible for the user interface, which consists of a generated web site. Judging from the answers to question 17, the reason for these testing problems is the lack of proper tools.

Second, the modules that were mentioned as being the hardest to test, having the most complex test cases and the highest defect ratio are the *QueuedComponent* class and the *command* package. The *QueuedComponent* class is responsible for the representation of a file that is to be processed by the DocGen program. The *command* package contains classes that represent commands which are used internally by the DocGen program to control its components. These classes are the result of an application of the command design pattern.

Third, if we instead look for modules that were mentioned as having the most test cases, the most complex test cases and the highest defect ratio, we find the *CobolModel* class and again the *QueuedComponent* class. The *CobolModel* class is a layer of convenience methods on top of those provided by the framework used for object-relational mapping (ORM). [1]

On the other hand, we try to identify modules that are thought to be well testable. First, modules that were mentioned as being the easiest to test, having the least complex test cases and having the lowest defect ratio are the *model* package and code dealing with regular expressions. The *model* package contains many classes that are generated by the ORM framework, which take care of data persistence and data access. Curiously, the *CobolModel* class is part of this package, yet it was also mentioned as being hard to test. Furthermore, the package contains a number of other classes like *CobolModel*, i.e. containing methods dealing with object persistence in a more convenient way. Typically, there is a class like *CobolModel* for every language that DocGen supports.

Second, a module that is said to be both easiest to test, and having the lowest defect ratio is the *extract* package. The *extract* package contains sub-packages for every supported language, which all deal with fact extraction from the parse tree representation of a piece of code. Parse tree traversal is done using so-called visitor classes, which are the result of an application of the visitor design pattern.

Finally, the *util* package is mentioned as being both easy to test, and having the least complex test cases. Classes in the *util* package are often static, and have a simple, well-defined responsibility like data conversion, file operations and logging.

[1] Apple's WebObjects.

B.4 Conclusion

The three modules we have identified above, the *CobolModel* and *Queued-Component* classes and the *command* package, have every sign of being badly testable. Conversely, the *model*, *util* and *extract* packages appear to cause little testing problems. Our further studies were focused on these modules. The remaining results, which have not been discussed here, were also taken into account.

Bibliography

[1] K. Beck. *eXtreme Programming eXplained.* Addison-Wesley, Reading, Massachusetts, 1999.

[2] A. Bertolino. Software testing. In *The Guide to the Software Engineering Body of Knowledge*, chapter 5. IEEE Computer Society, 2001. Public draft version 1.00, available at `http://www.swebok.org`.

[3] R. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[4] B.W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.

[6] L. C. Briand, S. Morasca, and V. Basili. An operational process for goal-driven definition of measures. *IEEE Transactions on Software Engineering*, 28(12):1106–1125, December 2002.

[7] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[8] R. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[10] B. Henderson-Sellers. *Object-Oriented Metrics.* Prentice Hall, New Jersey, 1996.

[11] H. Hotelling. The selection of variates for use in prediction, with some comments on the general problem of nuisance parameters. *Annals of Mathematical Statistics*, (11):271–283, 1940.

[12] IEEE. *IEEE Software Engineering Standards*. IEEE Society Press, New York, 1987.

[13] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[14] S. Jungmayr. Identifying test-critical dependencies. In *Proceedings of the International Conference on Software Maintenance*, pages 404–413. IEEE Computer Society, October 2002.

[15] J. McGregor and S. Srinivas. A measure of testing effort. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 129–142. USENIX Association, June 1996.

[16] Leon Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, December 2002.

[17] Object Technology International, Inc. *Eclipse Platform Technical Overview*, February 2003. Available at `http://www.eclipse.org`.

[18] L.H. Putnam. A general empirical solution to the macrosoftware sizing and estimating problem. *IEEE Transactions on Software Engineering*, 4(4):345–61, 1978.

[19] S. Siegel and N. J. Castellan Jr. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill Book Company, New York, 1988.

[20] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 40–49. IEEE Computer Society, 1999.

[21] B. van Zeist, P. Hendriks, R. Paulussen, and J. Trienekens. *Quality of Software Products*. Software Engineering Research Center, Utrecht, the Netherlands, 1996.

[22] J. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[23] J. Voas and K. Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20:207–216, March 1993.

[24] J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, 12:17–28, May 1995.

[25] J. Voas, K. Miller, and J. Payne. PISCES: A tool for predicting software testability. In *Proceedings of the Symposium on Assessment of Quality*

Software Development Tools, pages 297–309. IEEE Computer Society, May 1992.

[26] J. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8:41–48, March 1991.

[27] A. Watson and T. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. T. NIST Special Publication 500-235, U.S. Department of Commerce/National Institute of Standards and Technology, Washington, D.C., 1996.

[28] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, September 2003. To appear.