



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

An Empirical Study of the Comparative Effect of Aspect-
and Object-Oriented Programming on Testability

THESIS
9441

An Empirical Study of the Comparative Effect of Aspect- and Object-Oriented Programming on Testability

A thesis submitted to the University of Dublin, Trinity College
in fulfillment of the requirements for the degree of
Doctor of Philosophy

June 2010

Andrew Jackson

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

I agree that Trinity College Library may lend or copy this thesis upon request.

Andrew Jackson

Andrew Jackson


Dated: June 2, 2010



9441

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.



Andrew Jackson

Dated: June 2, 2010

Acknowledgements

I would like to thank my supervisor Dr. Siobhán Clarke for her guidance and encouragement. She is an amazing supervisor and is the reason that I was able to complete this thesis.

Thanks to all my friends in the Distributed Systems Group for making it a great place to work. A special thanks for all of those who helped me along the way and a very special thanks to my wife Jennifer, my daughter Kaia and the rest of my family for their constant support.

Andrew Jackson

University of Dublin, Trinity College

June 2010

Summary

Proponents of Aspect-Oriented Programming (AOP) claim that it improves maintainability over Object-Oriented Programming (OOP) by enhancing the degree to which concerns are separated in software. Maintainability is measured indirectly through a set of key indicators: analysability, changeability, stability and testability. To confirm that AOP improves maintainability, evidence of the comparative effect of AOP and OOP on each indicator is needed. Such evidence is required to objectively consider the adoption of AOP.

Empirical studies have contributed evidence of the comparative effect of AOP and OOP on analysability, changeability and stability. When analysed together this evidence indicates that AOP does improve this subset of maintainability indicators. However, these studies do not confirm that AOP improves maintainability as there is no comparative study of the effect of AOP and OOP on testability.

This thesis addresses this gap through an empirical study to quantify the comparative effect of AOP and OOP on testability. In the study, a series of maintenance activities are cumulatively applied to equivalent AOP and OOP implementations of a program. The effects of AOP and OOP on testability are measured by applying Mutation Analysis (MA) to both implementations after each maintenance activity. In MA, a set of tests are executed against mutants of the implementation. A mutant is a copy of an implementation that contains a fault. Testability is measured as the rate at which faults are exposed through test failure. The comparative effect is quantified by applying binomial regression (BR) to these measures.

Any comparative study of this kind faces challenges of maximising the degree to which

the results can be generalised and ensuring validity. Maximising the generalisability of the results is achieved in this thesis by selecting study inputs that are representative of the general case. The measure of the comparative effect of AOP and OOP on testability is valid if it is unbiased. Validity is ensured by designing the study such that sources of bias are controlled and inputs that are unbiased towards AOP or OOP are selected.

The contribution of this study is evidence of the comparative effect of AOP and OOP on testability. The evidence suggests that compared to OOP, AOP may increase testability. Although the study is based on inputs that are representative of the general case, the evidence is hard to generalise outside of the context from which it is derived. Although more studies are required to provide a more generally acceptable evidence, this evidence of the comparative effect of AOP and OOP on testability provides the first step toward filling the gap in the existing evidence of the effect of AOP on maintainability and enabling the adoption of AOP to be more objectively considered.

Contents

Acknowledgements	v
1 Introduction	9
1.1 Background	10
1.1.1 Maintainability	10
1.1.2 Crosscutting Concerns	11
1.1.3 Aspect-Oriented Programming	12
1.2 Motivation	13
1.2.1 Analysability	13
1.2.2 Changeability	13
1.2.3 Stability	13
1.2.4 Testability	14
1.2.5 Evidential Gap	14
1.3 Study	14
1.3.1 Mutation Analysis	15
1.3.2 Measurement Methodology	16
1.3.3 Analysis Approach	17
1.4 Challenge	18
1.4.1 Implementations	18
1.4.2 Tests and Mutants	18
1.5 Contributions	19
1.6 Thesis Outline	20
2 Related Studies	21
2.1 Empirical Evidence	22
2.1.1 Object Oriented Metrics	23
2.1.2 Comparing Object Oriented Metrics	23
2.1.3 Focus on Empirical Evidence	23
2.2 Walker et al. and Murphy et al.	23
2.2.1 Study	23

2.2.2	Empirical Evidence	25
2.3	Bartsch and Harrison	25
2.3.1	Study	26
2.3.2	Empirical Evidence	27
2.4	Lopes and Bajracharya	27
2.4.1	Study	28
2.4.2	Empirical Evidence	28
2.5	Li et al.	29
2.5.1	Study	29
2.5.2	Empirical Evidence	30
2.6	Kulesza et al.	30
2.6.1	Study	30
2.6.2	Empirical Evidence	31
2.7	Greenwood et al.	32
2.7.1	Study	32
2.7.2	Empirical Evidence	33
2.8	Figueiredo et al.	33
2.8.1	Study	34
2.8.2	Empirical Evidence	35
2.9	Evidential Gap	35
2.9.1	Existing Evidence	36
2.9.2	Testability Gap	36
2.10	Common Evidence Gathering Approach	36
2.10.1	Equivalence	36
2.10.2	Effects	37
2.10.3	Applicability	38
2.11	Chapter Summary	38
3	Testability	39
3.1	Fault Exposure Model	40
3.1.1	Faults	40
3.1.2	Tests	41
3.1.3	Model	44
3.2	Factors of Testability	45
3.2.1	Fault Type	45
3.2.2	Tests	46
3.2.3	Implementation	47
3.2.4	Factors	49
3.3	Comparative Studies of Testability	51

CONTENTS

3.3.1	Fault Type	51
3.3.2	Test	52
3.3.3	Implementation	52
3.3.4	Evidential Gap	53
3.4	Chapter Summary	53
4	Study Methodology	55
4.1	Testability Measurement	55
4.1.1	Measurement Approach Selection	56
4.1.2	Mutation Analysis	59
4.1.3	Summary	65
4.2	Gathering Measures of Testability	66
4.2.1	Following the Common Approach	66
4.2.2	Integrating Mutation Analysis into the Approach	69
4.2.3	Summary	75
4.3	Analysing Measures of Testability	76
4.3.1	Graphical Analysis	76
4.3.2	Binomial Regression Analysis	80
4.3.3	Threats to Analysis Validity	83
4.3.4	Summary	83
4.4	Chapter Summary	83
5	Study Inputs	85
5.1	Implementations	86
5.1.1	Selection of Implementations	87
5.1.2	Health Watcher - Use Cases and Maintenance Activities	88
5.1.3	Health Watcher - Java and AspectJ Implementations	90
5.1.4	Summary	92
5.2	Mutants	92
5.2.1	Fault Model	93
5.2.2	Mutant Operators	95
5.2.3	Mutant Generation Tool	96
5.2.4	Summary	99
5.3	Tests	99
5.3.1	Choosing a Test Selection Approach	99
5.3.2	Black Box Test Selection	101
5.3.3	Test Execution Automation	103
5.3.4	Summary	107
5.4	Chapter Summary	107

6	Study Results and Analysis	109
6.1	Comparison of Generated Mutants	111
6.1.1	Results of Mutant Generation	111
6.1.2	Mutant Equivalence	112
6.1.3	Summary	115
6.2	Analysis of Outcomes and Rates	115
6.2.1	Outcomes	115
6.2.2	Fault Execution	116
6.2.3	Infection and Propagation	119
6.2.4	Fault Exposure	122
6.2.5	Summary	124
6.3	Quantifying the Comparative Effects	124
6.3.1	Binomial Regression	124
6.3.2	Comparative Effects	125
6.3.3	Summary	127
6.4	Threats to Validity	128
6.4.1	Testability Measurement	129
6.4.2	Program Selection	130
6.4.3	Test Selection	132
6.4.4	Summary	135
6.5	Chapter Summary	135
7	Conclusions and Future Work	137
7.1	Conclusions	138
7.1.1	Comparative Effect of AOP and OOP on Testability	138
7.1.2	Causes of Comparative Effect	139
7.1.3	Advice for Adoption of AOP	140
7.1.4	Issues to Consider when Adopting AOP	140
7.2	Future Work	141
7.2.1	Pointcut Issues	141
7.2.2	Causation of Lower Infection and Propagation Odds for AOP	141
7.2.3	Testing	142
7.2.4	Program	142
7.2.5	Mutant Generation	142
A	Additional Results	143

List of Tables

2.1	Li et al. Change Impacts	30
2.2	Indicator Coverage	35
3.1	Requirements for Fault Exposure	45
3.2	Effects of Differences in Fault Factor	46
3.3	Effects of Differences in Test Factor	47
3.4	Effects of Differences in Implementation Factor	49
3.5	Factor Levels	50
3.6	Overview	50
4.1	Test Path Execution	59
4.2	Mutation Analysis Outcomes	75
4.3	Mutation Analysis Outcomes	76
4.4	Models	81
5.1	Candidates for Selection	87
5.2	Health Watcher Version 1: Use Cases [87]	89
5.3	Health Watcher Version 9: Use Cases [87]	89
5.4	Health Watcher Versions 2 - 10: Maintenance Activities	89
5.5	Java Fault Model	97
5.6	AspectJ Fault Model	98
5.7	Employee Login Use Case	102
5.8	Examples of Test Selection	102
5.9	Test Data	102
5.10	Tests Selected for Use Cases	103
6.1	Models	125
6.2	Max and Min results for rates for randomly selected test sizes	133

List of Figures

1.1	Fault Exposure: Outcomes and Rates	15
1.2	Measurement Methodology	16
1.3	Contributions	19
1.4	Causation	19
2.1	Evidential Gap	22
2.2	Walker et al. and Murphy et al. Experiments	24
2.3	Bartsch and Harrison Experiment	26
2.4	Lopes and Bajracharya Study	27
2.5	Li et al. Study	29
2.6	Kulesza et al. Study	30
2.7	Study of Greenwood et al.	32
2.8	Study of Figueiredo et al.	34
2.9	Approach	37
2.10	Measures	37
2.11	Result	37
3.1	Testability	39
3.2	Control-flow graph for <code>setSize</code> method	41
3.3	Control-flow paths through <code>setSize</code> method for tests 1 - 4	42
3.4	Fault Factors	46
3.5	Test Factor	47
3.6	Implementation Factors	49
3.7	Designs of Testability Studies	51
4.1	Measurement	56
4.2	Factors	56
4.3	Analysis	56
4.4	Stack Result	56
4.5	Measuring testability as the rate of fault exposure	57

4.6	Measuring Testability as the Number of Tests Needed for Fault Exposure	58
4.7	Mutation Analysis	61
4.8	Mutation Operators Applied at Locations	61
4.9	Locations Executed by <i>Test 1</i>	63
4.10	Faults Execution	64
4.11	Fault Exposure	64
4.12	Fault Exposure: Model, Rates and Effects of Sub-Rates	64
4.13	Approach	67
4.14	Factors	67
4.15	Stack Example	69
4.16	Size	70
4.17	Mutation Analysis	70
4.18	Mutant Gen	71
4.19	Location Exe	71
4.20	Exposure	71
4.21	Mutant Gen	71
4.22	Location Exe	71
4.23	Exposure	71
4.24	Java	75
4.25	AspectJ	75
4.26	Visualising Outcomes	77
4.27	Fault Exposure	77
4.28	Fault Execution	77
4.29	Infection and Propagation	77
4.30	Conclusions of Analysis for Stack	80
4.31	Correlations between each Rates and versions and implementation factors	82
4.32	FE	82
4.33	FX	82
4.34	IP	82
4.35	Causation for Comparative Effects	82
5.1	Program	85
5.2	Mutants	85
5.3	Tests	85
5.4	Object Oriented Health Watcher [63]	90
5.5	Aspect Oriented Health Watcher [87]	91
5.6	MuJava	98
5.7	AspectJ Extension	98
5.8	Automated Employee Login Test	104

LIST OF FIGURES

5.9	Location Execution	105
5.10	Fault Exposure	105
5.11	Distribution of Test Executions	106
6.1	Fault Exposure: Mutant and Rates	109
6.2	Mutant Generation	113
6.3	Measure Visualised	113
6.4	Types of Faults	113
6.5	Size for Correlation	113
6.6	Outcomes	116
6.7	Outcomes Visualised	116
6.8	Rates of Fault Execution	117
6.9	Data Layer Initialisation	117
6.10	Example of the Impact Of Data Layer Initialisation On Client	118
6.11	Fail and Pass Outcomes	119
6.12	Rates of Infection and Prop.	119
6.13	Fault Types	121
6.14	Rates of Fault Exposure	122
6.15	Comparative Effects	123
6.16	Measures of the relative effects of implementation and version levels on rates	126
6.17	FE	126
6.18	FX	126
6.19	IP	126
6.20	Comparative Effects Quantified	126
6.21	Causation	127
6.22	Measuring Testability In A Comparative Context	130
6.23	Convergence as test set size increases	134
6.24	Intervals for the difference between AspectJ and Java	135
7.1	Gap	137
7.2	Comparative Effects	138
7.3	Reason for Comparative Effects	140
A.1	Distribution of Fault Types	144

Chapter 1

Introduction

Proponents of Aspect-Oriented Programming (AOP) claim that it improves maintainability over Object-Oriented Programming (OOP) by enhancing the degree to which concerns are separated in software [84, 83, 50, 68, 125, 89, 40]. The key indicators of maintainability are analysability, changeability, stability and testability [73]. This chapter provides the background from which this claim has emerged by describing why AOP is expected to improve them.

Studies consistently show that maintenance accounts for the largest proportion of a programs total cost [154, 96, 22, 51]. Making programs easier to change reduces this cost. Improving modularity has been shown to make programs more maintainable [42]. OOP was a major improvement on modularity and is currently the defacto approach for implementing programs [136].

The claim that AOP improves maintainability and consequently reduces costs over OOP, has led organisations using OOP to consider adopting AOP [39, 2]. However, for the adoption of AOP to be objectively considered, confirmation of this claim is required [39, 25]. To confirm that AOP improves maintainability, evidence of the comparative effect of AOP and OOP on each indicator of maintainability is needed [129].

This chapter presents the existing evidence of the comparative effect of AOP and OOP on some of the key indicators of maintainability [149, 14, 88, 66, 92, 97, 54, 87, 63]. It shows a gap in this evidence, as there is no empirical evidence of the testability of AOP. Testability is a key component of maintainability [33, 22, 141]. Without evidence of the comparative effect of AOP and OOP on testability the confirmation of any claim related to maintainability is superficial [129, 25]. The adoption of AOP, therefore, cannot be objectively considered [39].

This thesis addresses this gap through an empirical study to quantify the comparative effect of AOP and OOP on testability. In the study, a series of maintenance activities are cumulatively applied to equivalent AOP and OOP implementations of a program. The testability of both implementations is measured after each maintenance activity and these

measures are then analysed to quantify the comparative effect. This chapter introduces how the testability is measured using mutation analysis [45]; how measures of testability are gathered to ensure that the testability measurements for the AOP and OOP implementations are directly comparable; and how binomial regression [52] is used to measure the comparative effect of AOP and OOP on testability.

This study, like similar studies [24], faces a fundamental challenge of maximising the degree to which the results can be generalised. To maximise the degree to which evidence gathered from a single study can be generalised, the inputs on which the study is based must be representative of the general case. This chapter describes the inputs selected for this study and shows that they are representative of the general case.

The chapter concludes by presenting the contributions of the study and outlining the remaining chapters of this thesis.

1.1 Background

This section outlines the background from which the claim that AOP improves maintainability over OOP has emerged and describes why AOP is expected to improve the key indicators of maintainability.

1.1.1 Maintainability

Maintainability is a measure of the ease with which a program's implementation can be changed [73]. Applying a change to a program is made in four steps [129]. The maintainability of an implementation is based on the ease with which each step is taken [153, 129].

The first step is to understand and identify what parts of the implementation need to be changed. An understanding of the implementation is needed before the code can be analysed to identify the parts the change is applicable to. The easier the code is to understand and analyse, the easier it is to change.

The second step is to implement the change. The ease with a change can be implemented is a measure of the size of the impact it will make [8]. The impact is measured as the effort needed to implement the change. The smaller the impact is, the easier the change is to make.

The third step is to address the ripple effect of the change. A change to a module can propagate through its efferent dependencies causing the effect of the change to be amplified. Minimising these dependencies makes the system more stable and resistant to ripple effects. The more stable an implementation is, the easier a change is to apply [129].

The fourth and final step is to expose faults that are introduced into the implementation during the change. Faults in an implementation are exposed through test failure

1.1. BACKGROUND

[110]. The implementation can hide faults by allowing tests to pass when faults are present [148]. An implementation that exposes more faults introduced through change is more testable and easier to change.

Measures of the ease with which each step can be made are key indicators of a programs maintainability. These key indicators are analysability, changeability, stability and testability [73]. Analysability is the ease with which the program's code can be understood and analysed. Changeability and stability are indicators of the ease with which a programs implementation can be changed. Testability is the ease with which faults can be exposed through testing [148].

1.1.2 Crosscutting Concerns

Since the inception of software engineering, increasing modularity has been recognised [9, 46, 138, 120, 27, 28, 13, 32] as a way to improve the separation of concerns in programs and improve maintainability. Concerns are the behaviours or features that make up a program [82, 84, 48].

In a well modularised object-oriented program, each concern is implemented in a module. The module encapsulates the concerns implementation. Some concerns can be separated into modules in well modularised object-oriented programs. There are however other concerns that cannot be cleanly encapsulated as modules. When the implementation of a concern cannot be encapsulated within one module, it becomes scattered across other program modules. Within these modules, this implementation becomes entangled with the implementation of the primary concern. Scattered and tangled concerns are said to crosscut the program and are called crosscutting concerns [82, 84, 48].

Claims that AOP improves maintainability over OOP are based on the improved separation of crosscutting concerns that AOP supports [82, 84]. It is expected that this improvement will result in improvements in the key indicators of maintainability. The negative effects of crosscutting concerns on analysability, changeability, stability and testability are the basis for the claims. The negative effects on each are outlined in this subsection.

Analysability

Decomposing a problem into its constituent parts makes each part and the problem itself easier to understand [123]. In OOP, the program is made easier to understand by decomposing each concern into a module. Crosscutting concerns have a scattering and tangling effect that make the program harder to understand [48]. Scattering increases the number of modules that need to be examined to understand a concern. Tangling makes it difficult to understand one concern in isolation from other entangled concerns. These issues conspire to make it harder to identify the parts of a crosscutting concern's implementation

to which a change is applicable.

Changeability

Change is easier to implement when its impact is small [8]. In OOP the goal is to minimise a change's impact by localising the change to one module. However, when a concern is crosscutting the impact of the change can increase [92], as it can have an impact on all of the modules across which it is scattered. The potential impact is compounded by tangling because a change applied to one concern, can have an impact the others with which it is entangled.

Stability

When a program is resistant to the ripple effect of change it is more stable and easier to change [101]. In OOP, this effect is reduced by minimising the ratio of outgoing to incoming dependencies for each module [101]. Crosscutting concerns can reduce stability by introducing outgoing dependencies into the modules they crosscut. A higher number of outgoing dependencies per module increases potential of ripple effects of change. If a program is not resistant to ripple effects it is harder to change.

Testability

A program that exposes more faults is more testable and easier to change. Faults are exposed through testing. A fault in behaviour can only be exposed if it is executed. When a behaviour is scattered and tangled it is more difficult to select tests that will guarantee execution of the behaviour. Also, more faults are found to occur at these scattered and tangled behaviours [48]. Together, these issues make the faults at scattered and tangled behaviours difficult to expose [48].

1.1.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) improves the degree to which crosscutting concerns are separated in software over Object-Oriented Programming (OOP) [82, 63]. AOP introduces a new aspect module that can be used to encapsulate crosscutting concerns. By modularising crosscutting concerns the amount of scattering and tangling caused by crosscutting is reduced. Crosscutting concerns have a negative effect on analysability, changeability, stability and testability. The expectation that AOP will lead to improvements in the key indicators of maintainability is based on the improved modularity of crosscutting concerns facilitated by AOP [84, 83, 50, 68, 125, 89, 40].

1.2 Motivation

This section outlines the evidence of the comparative effect of AOP and OOP on analysability, changeability and stability contributed by existing empirical studies. A clear gap is identified in the empirical evidence of the comparative effect of AOP and OOP on testability.

1.2.1 Analysability

Contributions to the evidence of the comparative effect of AOP and OOP on analysability have been made through empirical studies carried out by Murphy et al. [109], Walker et al. [149] and Bartsch and Harrison [14]. Analysability is measured as the time taken to identify the parts of each implementation that will be affected for a specific change by Murphy et al., Walker et al. and Bartsch and Harrison. The findings of Murphy et al. and Walker et al. suggest that AOP results in slightly higher analysability while Bartsch and Harrison find no significant difference between AOP and OOP.

1.2.2 Changeability

Evidence of the comparative effect of AOP and OOP on changeability have been contributed by a number of studies including Walker et al. [149], Bartsch and Harrison [14], Li et al. [92] and Lopes and Bajracharya [97]. In these studies, changeability is measured by applying the same change(s) to AOP and OOP implementations and measuring the difference between the impacts the change has on each. Walker et al. and Bartsch and Harrison measure the size of the impact as the time taken to implement the change. Li et al. measure the size of the impact by counting the number of modules that are affected by a change. Lopes and Bajracharya measure the impact in terms of its effect on design options. The findings of Lopes and Bajracharya, Li et al. and Walker et al. consistently suggest that AOP can lead to improved changeability over OOP. Bartsch and Harrison find no significant difference in the effects of AOP and OOP on changeability.

1.2.3 Stability

Empirical studies of the comparative effect of AOP and OOP on stability have been carried out by Figueiredo et al. [54], Kulesza et al. [87] and Greenwood et al. [63]. In these studies, stability is measured as the resistance of AOP and OOP implementations to ripple effects when the same change(s) are applied to each. This resistance is measured using a suite of metrics that measure coupling, size and other quality indicators and analysing how much these indicators are affected over maintenance activities [49]. The results of these studies indicate that AOP can lead to improved stability over OOP.

1.2.4 Testability

There are no empirical studies of the comparative effect of AOP and OOP on testability. Testability can have a significant effect on maintenance costs [33, 22, 141]. One cause of this is that new faults are introduced into the implementation when changes are applied. It is estimated that 40% of changes introduce new faults [124]. This indicates that testability is the most significant indicator of maintainability.

It has been shown that crosscutting concerns are more likely to contain faults [48]. AOP reduces the crosscutting of concerns over OOP. This indicates that there is potential for AOP to improve testability. The reduction in crosscutting is based on the introduction of a composition mechanism that can introduce a range of new types of faults. It has been observed that these new types of faults can be more difficult to expose through testing [2], deflating the potential of AOP to improve testability somewhat.

1.2.5 Evidential Gap

The existing studies provide a significant amount of evidence to suggest that AOP can improve analysability, changeability and stability. Although more studies are needed to fully validate the benefits of AOP, the existing evidence is encouraging for those considering the adoption of AOP to reduce maintenance costs. However, due to the high proportion of the maintenance costs attributed to testing, the confirmation of this claim, without evidence of the comparative effect of AOP and OOP on testability, is superficial. Considering the adoption of AOP based on a superficial confirmation cannot be objective [39, 25].

1.3 Study

This thesis addresses the evidential gap through a study to gather empirical evidence of the comparative effect of AOP and OOP on testability. In the study, a series of maintenance activities are cumulatively applied to equivalent AOP and OOP implementations of a program. The testability of both implementations is measured after each maintenance activity. These measures are then analysed to quantify the comparative effect. This quantification is empirical evidence of the comparative effect of AOP and OOP on testability.

This section introduces how the testability is measured using mutation analysis [45], how measures of testability are gathered, and how these measures are analysed to quantify the comparative effect of AOP and OOP on testability.

1.3. STUDY

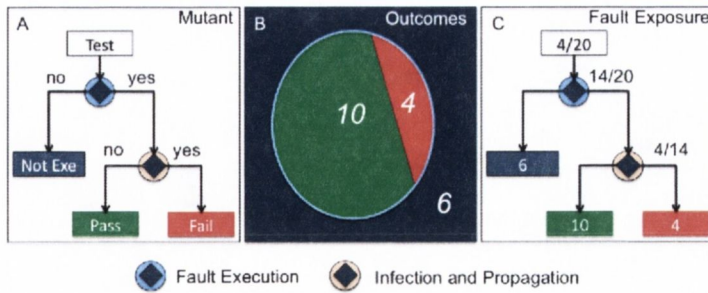


Figure 1.1: Fault Exposure: Outcomes and Rates

1.3.1 Mutation Analysis

Faults in a program’s implementation are exposed through testing. The implementation can hide faults by allowing tests to pass when faults are present [128, 148, 60]. A program that does not hide faults has high testability.

Mutation Analysis (MA) [45] measures the testability of an implementation as its rate of *fault exposure* under testing. In MA, tests are executed against mutants of the implementation. A mutant is a version of the implementation that contains a fault. Examples of mutants and details of how they are automatically generated are presented in Chapter 4.

The rate of *fault exposure* is based on the outcomes of executing tests against mutants. Part A of Figure 1.1 shows that there are three possible outcomes for each test-mutant execution. The test executes a path through the mutant implementation. The fault contained in the mutant may or may not be executed on this path. Chapter 3 demonstrates that if the fault is executed, then the state directly after that can become infected and this state infection can be propagated [128, 148, 146, 147, 76, 1], which in turn results in a *fail* outcome. A *fail* outcome indicates fault exposure. Infection occurs when the execution of the fault results in a state that differs from the state that would occur if the fault was not present [128, 148]. The infected state is propagated if it causes the output of the implementations execution to differ from the output of the normal execution [128, 148]. If the state directly after the faults execution does not become infected or propagated, then the resulting outcome is a *pass*.

Part B of Figure 1.1 illustrates a simple example in which one test is executed against 20 mutants of the implementation. Of the 20 mutants executed, the fault contained is *not* executed in 6 of them. This means that there are 14 mutants in which the fault is executed. Of these 14 fault executions, 10 do not cause state *infection and propagation* and result in *pass* outcomes. There are 4 out of the 14 fault executions that do cause state *infection and propagation* and result in *fail* outcomes.

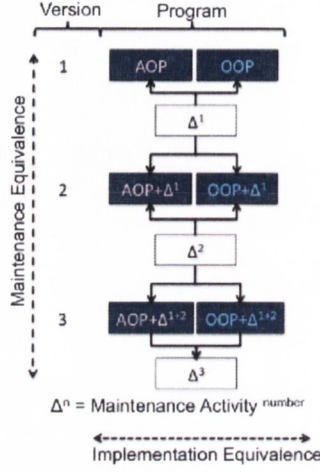


Figure 1.2: Measurement Methodology

Part C of Figure 1.1 demonstrates how these outcomes are used to calculate a rate of *fault exposure* for an implementation. The rate of fault exposure in this simple example is $\frac{4}{20} = \frac{\textit{fail}}{\textit{fail}+\textit{pass}+\textit{notexe}}$. This rate is based directly on the rates of *fault execution* and *infection and propagation*, $\frac{4}{20} = \frac{14}{20} \times \frac{4}{14}$. The rate of *fault execution* is $\frac{14}{20} = \frac{\textit{fail}+\textit{pass}}{\textit{fail}+\textit{pass}+\textit{notexe}}$. The rate of *infection and propagation* is $\frac{4}{14} = \frac{\textit{fail}}{\textit{fail}+\textit{pass}}$.

1.3.2 Measurement Methodology

The goal of this study is to compare the effects of AOP and OOP on testability, as an indicator of maintainability. To ensure that this goal was achieved, the study followed a measurement methodology widely used in existing studies [149, 14, 92, 54, 87, 63] that compare the effects of AOP and OOP on analysability, changeability and stability. The basis for this measurement methodology is illustrated in Figure 1.2 and is detailed in Chapters 2 and 4.

In the methodology, maintenance activities are cumulatively applied to AOP and OOP implementations of a program. The initial AOP and OOP implementations of a program are equivalent in that they differ only in the approach used for their development. Equivalence is assured by fixing all other factors that can cause the implementations to differ. Examples of these factors are the expertise used in developed of both AOP and OOP implementations and the requirements they satisfy. They are fixed by ensuring that these factors are equivalent for each pair of AOP and OOP implementations. As will be detailed in Chapters 2, 4 and 5, the implementations were developed to the same level of expertise, satisfy the same requirements, expose the same interface and produce the same outputs for a given input [63].

1.3. STUDY

The same maintenance activities are cumulatively applied to each pair of equivalent AOP and OOP implementations. After each maintenance activity is applied to both implementations, new versions of these implementations result. The new versions of these implementations are equivalent because the same maintenance activity is applied to both implementations [63]. This means that the respective versions of the AOP and OOP implementations are also equivalent. When all maintenance activities are applied, the only difference between each respective version of the AOP and OOP implementations is the maintenance activity.

Following this methodology ensures that the measures gathered represent the effects of the implementation approach (AOP or OOP) and maintenance factors on the measure. The use of MA within this methodology does however, require some additional factor fixing. This is because MA introduces new test and mutant factors that can affect each measure. The application of MA to the AOP and OOP implementations of each version of the program requires the execution of tests against mutants generated from the implementation. To preserved equivalence, these factors are fixed. They are fixed by using the same set of tests and mutant generation approach in the application of MA to each implementation. Further details of this methodology are presented in Chapter 4.

1.3.3 Analysis Approach

The result of applying mutation analysis to each AOP and OOP implementation over versions of the program are numbers of *not exe*, *pass* and *fail* outcomes for each implementation. These outcomes are used to derive rates of *fault exposure*, *fault execution* and *infection and propagation* for each implementation. As demonstrated in Chapter 4, these rates are informally analysed by interpreting graphs to identify if these rates are higher or lower for AOP compared to OOP.

Binomial Regression Analysis (BRA) [52] is also applied to the outcomes for each implementation to quantify precisely how much higher or lower the effects AOP on rates are compared to OOP. BRA is a formal statistical technique for analysing the causal relationship between a binomial response and explanatory factors. In this study, there are three applications of BRA, one for each rate. In these applications of BRA, the rate is the response and the explanatory factors are the implementation approach and maintenance version. The causal relationship between the binomial response and the explanatory factors is defined in a regression model [52]. To measure the effects of these factors on a rate, the model is fitted to the outcomes. In the fitting process the effects are measured based on the correlation between each factor and the rate [52]. The comparative effect of AOP and OOP on each rate is measured as the difference between the effects of AOP and OOP on the rate.

1.4 Challenge

This study, like similar studies [24, 79], faces the fundamental challenge of maximising the degree to which the results can be generalised. To maximise the degree to which evidence gathered from a single study can be generalised, the inputs on which the study is based must be representative of the general case. This section shows that implementation, tests and mutants selected for this study are representative of the general case.

1.4.1 Implementations

The implementations of the health watcher [87, 63, 55] program were selected from a pool of candidates, listed in Chapter 5, that fit the methodology presented in Figure 1.2. These implementations were selected because the health watcher program and the maintenance activities associated with it were the most representative of the general case.

The health watcher is a public health system that supports the registration, tracking and resolution of health complaints. This program is a relatively large, database-driven, distributed system with a web front-end and is made up of a set of concerns generally found in a wide range of contemporary programs [87, 63].

The program was deployed for use in 2001 [63] and since its deployment, a number of adaptive, corrective and perfective changes have been applied to it [63]. The maintenance activities selected for use in this study are based on these and have been selected because they are representative of the typical distribution of maintenance activity types [124].

The AOP and OOP languages used to develop the implementations are AspectJ and Java. AspectJ is currently the most widely used AOP language [106]. AspectJ is an extension of Java, which is currently the most widely used OOP language [136].

1.4.2 Tests and Mutants

The test set used in this study is the product of a use case driven test selection process [74, 75]. In this process, use cases are used as the basis for test case selection [5, 23]. As is detailed in Chapter 5, this approach was selected over others because it is representative of the type of approach used to select tests for an implementation in practice. Testing professionals applied the approach to the health watcher use cases to ensure that the application of the approach and the resulting tests were highly representative of practice.

The types of faults generated in mutants for the study are representative of those observed in practice. These mutants are generated using MuJava [98], a tool that generates mutants for Java implementations. MuJava has been widely used to generate mutants that contain realistic faults in testing related research [151, 122, 133, 104, 103, 102, 105, 130, 134]. As part of this work, the tool was extended to generate AspectJ specific mutants.

1.5. CONTRIBUTIONS

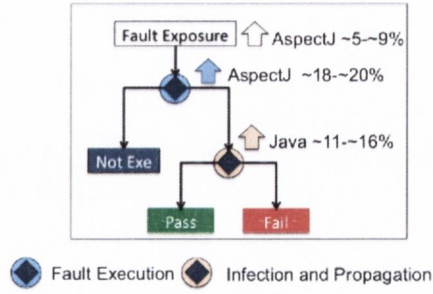


Figure 1.3: Contributions

The extension introduces the types of faults observed to occur in AspectJ implementations [53] and allows mutants containing these types of faults to be generated.

1.5 Contributions

The primary contribution of the study presented in this thesis is evidence to indicate that the effect of AOP is to increase testability over OOP. The results of the study are illustrated in Figure 1.3. This shows that the odds of *fault exposure* are between 5 and 9% higher for the AspectJ implementations of the health watcher program. This means that, for the health watcher program, faults are easier to expose in AspectJ compared to Java implementations. This is evidence to indicate that the effect of AOP is to increase testability over OOP. Testability can have a significant effect on maintenance costs [33, 22, 141] and for those considering the adoption of AOP to reduce maintenance costs [39], this evidence is encouraging.

A secondary contribution of the study, also presented in Figure 1.3, is to identify the causes of the 5-9% difference in the effects of AspectJ and Java on the odds of *fault exposure*. *Fault exposure* is a direct consequence of *fault execution* and state *infection and propagation*. If more faults are executed, then there are more chances for state

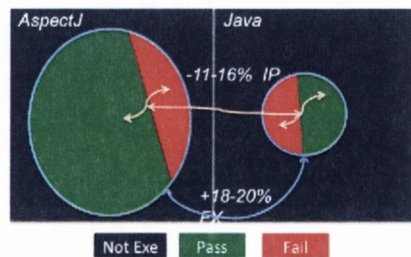


Figure 1.4: Causation

and propagation, resulting in fault exposure. The more executed faults that cause state infection and propagation, the more faults that are exposed. If more faults are exposed, then the odds of *fault exposure* increase.

Figure 1.3 shows that the odds of *fault execution* are between 18-20% higher in the AspectJ implementations and that the odds of state *infection and propagation* are between 11-16% lower in the AspectJ implementations. This means that in the AspectJ implementations there are more faults executed. However, it also means that compared to Java implementations, proportionally less of the executed faults cause state *infection and propagation*, resulting in lower odds of fault exposure.

This is explained further through the illustration in Figure 1.4. The boxes marked AspectJ and Java represent the total number of test-mutant executions for AspectJ and Java implementations, respectively. The circles in these boxes represent the number of faults executed by tests in AspectJ and Java mutants. This representation shows that there are more faults executed in AspectJ compared to Java mutants. This difference is the cause of the 18-20% higher odds of *fault execution* for AspectJ. The number of executed faults that result in pass and fails are represented inside the circle. This representation shows that there are proportionally less fails for AspectJ, indicating that less of the faults executed in AspectJ mutants result in infection and propagation. This difference is the cause of the 11-16% lower odds of *infection and propagation* for AspectJ.

Figure 1.4, indicates that even though there is proportionally less fail to pass outcomes from AspectJ test-mutant executions, the odds of fault exposure is 5-9% higher because the volume of fail outcomes is higher for AspectJ. The volume is higher because the number of faults executed in mutants (or pass and fail outcomes) is higher for AspectJ compared to Java.

1.6 Thesis Outline

The remainder of this thesis is organised as follows. Chapter 2 presents a review of the studies that compare the effects of AOP and OOP on indicators of maintainability to demonstrate the evidential gap addressed by this thesis. Chapter 3 describes the factors that affect fault exposure. Chapter 4 details the methodology followed in the study. Chapter 5 describes the implementations, tests and mutants on which the study is based. Chapter 6 presents the results of the study and Chapter 7 draws conclusions from these results.

Chapter 2

Related Studies

Studies consistently show that maintenance accounts for the largest proportion of a program's total cost [154, 96, 22, 51]. OOP is currently the defacto implementation approach with more new projects using it than any other approach [136]. The claim that AOP improves maintainability and consequently reduces costs over OOP has lead organisations using OOP, to consider adopting AOP [39, 2]. However, for the adoption of AOP to be objectively considered, empirical evidence of the comparative effects of AOP and OOP on maintainability is required [39, 25, 2].

Maintainability is a measure of the ease with which a program can be changed [73]. Applying a change to a program is made in four steps [129, 153]. The first step is to understand and analyse the program and identify what parts of the implementation need to be changed (analysability). The second step is to implement the change (changeability). The third step is to address the ripple effects of the change (stability) and the fourth step is to expose faults that are introduced into the program during the change (testability).

Measures of the ease with which each step can be made are key indicators of a program's maintainability. These key indicators are analysability, changeability, stability and testability [73]. To confirm that AOP improves maintainability, empirical evidence of the comparative effect of AOP and OOP on each indicator of maintainability is needed [129]. Studies have contributed empirical evidence of this comparative effect for some of these indicators [149, 14, 92, 97, 54, 87, 63].

The primary goal of this chapter is to review these studies to demonstrate the evidential gap left by them, illustrated in Figure 2.1. This is achieved in two steps. The first step is to identify the empirical evidence of the comparative effect of AOP and OOP gathered by each study and the indicator of maintainability the evidence to which it relates. The second step is to show that although there is empirical evidence of the comparative effect of AOP and OOP on analysability, changeability and stability, there has been no empirical evidence of the comparative effect on testability published in English in a major academic journal or conference.

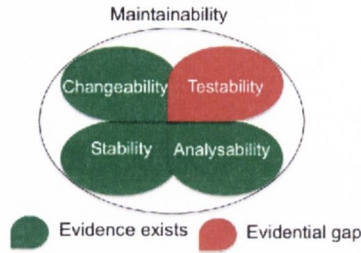


Figure 2.1: Evidential Gap

Testability can have a significant effect on maintenance costs [33, 22, 141]. This makes testability an important indicator of maintainability. Without evidence of the comparative effect of AOP and OOP on testability, the maintainability claim cannot be either rejected or confirmed [129, 25]. The adoption of AOP, therefore, cannot be objectively considered [39].

The secondary goal of this chapter is to review how these studies gather empirical evidence and **identify an approach to gathering evidence that can be used in the study presented in this thesis**. This is achieved in two steps. The first step is to identify how empirical evidence is gathered by each study. The second step is to identify an approach used in these studies that can be used to gather empirical evidence of comparative effect of AOP and OOP on testability.

The first section of this chapter justifies the focus on empirical studies. In the body of the chapter each of the studies that contributes empirical evidence is described. For each study, the way in which it gathers evidence and the empirical evidence it contributes are identified. The chapter is concluded by identifying an approach that is used consistently to gather evidence for of the comparative effect of AOP and OOP on the key indicators of maintainability and with a discussion of the applicability of this approach to this study. The empirical evidence contributed by each study is also collated to show that there is no empirical evidence of the comparative effect of AOP and OOP on testability.

2.1 Empirical Evidence

Only studies that have contributed empirical evidence of the comparative effect of AOP and OOP on the key indicators of maintainability are considered in this chapter. There are other studies that provide evidence of this comparative effect based on predictive metrics [144, 143, 108]. These studies typically use object-oriented metrics [144, 143, 108, 62] to predict, rather than observe, the ease with which an AOP and OOP implantation can be analysed, changed or tested.

2.1.1 Object Oriented Metrics

The predictions made by applying object-oriented metrics are based on assumed correlations between object-oriented features and the ease of analysis, change or testing. Although there is evidence to validate some of these correlations for OOP [38, 93, 4, 15, 81, 29, 30], there is no empirical evidence that these correlations are valid for AOP.

This means that some confidence can be associated with the accuracy of maintainability measures derived from applying object-oriented metrics to OOP implementations. It also means that less confidence can be associated with the accuracy of maintainability measures derived from applying object-oriented metrics to AOP implementations.

2.1.2 Comparing Object Oriented Metrics

If the object-oriented metrics do not provide accurate measures when applied to AOP, then this makes the comparison of measures that result from applying these metrics to AOP and OOP implementations inaccurate [34, 66, 88, 35, 144]. Object-oriented metrics are based on features of OOP and do not incorporate AOP specific features. Bias is introduced when comparing measures based on these metrics when they are not equally applicable to AOP and OOP. There are adaptations of these metrics that capture the effects of some, but not all AOP specific features [144, 108, 35, 34]. Although the adaptations reduce bias, they do not mitigate it because they do not capture the effects of all AOP features.

2.1.3 Focus on Empirical Evidence

Measuring the observed impacts of applying maintenance activities to AOP and OOP implementations of a program is the only way to gather accurate measures of maintainability. This is because these measures are direct observations rather than inaccurate predictions. Measuring the observed impact is an approach that is equally applicable to AOP and OOP implementations. This means that the measures of impact are directly comparable.

2.2 Walker et al. and Murphy et al.

Two experiments are carried out in a study by Walker et al. [149] and Murphy et al. [109]. In these experiments, empirical evidence of the comparative effect of AOP and OOP on analysability and changability is gathered.

2.2.1 Study

Figure 2.2 shows the three phases of the study. It begins with group assignment, in which participants are grouped for each experiment. In experiments 1 and 2 the participants are

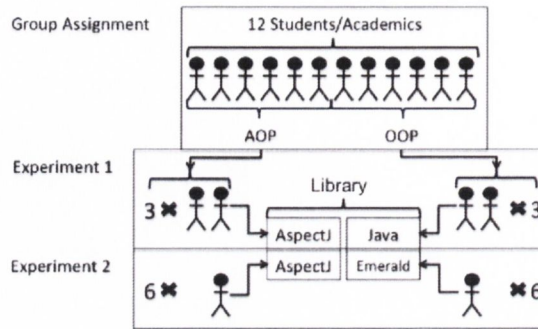


Figure 2.2: Walker et al. and Murphy et al. Experiments

asked to complete analytical tasks and perform tasks in which changes are applied to an implementation. During the tasks, the performance of the participants is measured and these measures are analysed.

Group Assignment

In this study, twelve computer science graduate students and academics were asked to carry out maintenance activities on OOP and AOP implementations of a trivial library program. Each participant was assigned to work on either the OOP or AOP implementation. At the end of the group assignment there were two groups of six, one group was assigned to the OOP implementation and the other to the AOP implementation. Figure 2.2 illustrates this process.

Experiment 1

In the first experiment, illustrated in Figure 2.2, three pairs of similar ability were formed out of each group. These pairs were then asked to analyse the Java and AspectJ implementations of the library program to which they were assigned. The goal of this analysis was to identify three cascading synchronisation faults. These faults were cascading in that symptoms of the first fault hid symptoms of the second, and so on. Each pair was videotaped and measures of the time taken to identify each fault, the amount of switching between files and the number of instances of semantic analysis were recorded.

Experiment 2

In the second experiment, also illustrated in Figure 2.2, the participants in each group were asked to cumulatively apply three changes to Emerald [21] (an OOP language) and AspectJ implementations to which they were assigned. Two changes were adaptive and

2.3. BARTSCH AND HARRISON

one change was perfective in nature. The time spent applying these changes and the proportion of time used for analysis and coding was recorded.

Analysis

The data gathered in both experiments is analysed graphically. The time taken, the number of switched between files less and number of semantic analyses recorded for each fault in the first experiment are analysed in separate graphs where the pairs assigned to the AOP and OOP implementations are directly compared.

The time each participant took to complete each change and the percentage of that time spent on coding and analysis identified in the second experiment are also presented in separate graphs. These graphs allow the individual participants assigned to the AOP and OOP implementations to be directly compared.

To identify causation for the interpreted differences between the AOP and OOP implementations, interviews with participants after the experiment were held. In these interviews the experience of the participants are recorded. These recorded interviews were then transcribed and used to identify causation of the graphed data.

2.2.2 Empirical Evidence

The results of the first experiment indicated that the AspectJ implementation was more analysable. The pairs who analysed the AspectJ implementation took less time to identify faults, switched between files less and performed more semantic analysis. These results suggest that the AspectJ implementation was easier to analyse than the Java implementation. Transcriptions of interviews with participants after the experiments identify that the localisation of synchronisation behaviour in the AspectJ implementation made the analysis easier.

The results of the second experiment indicated that the AspectJ implementation was less changeable. The graphical analysis suggests that overall, the changes took more time to implement using AspectJ. However, analysis of the proportions indicate that more time is spent on analysis in the Emerald implementation and more time was spent on implementation in the AspectJ. This again implies that AspectJ is more analysable but it also indicates that the AspectJ implementation is harder to change.

2.3 Bartsch and Harrison

Bartsch and Harrison [14] present a similar study to Walker et al [149]. Their study gathers empirical evidence of the comparative effect of AOP and OOP on analysability and changeability.

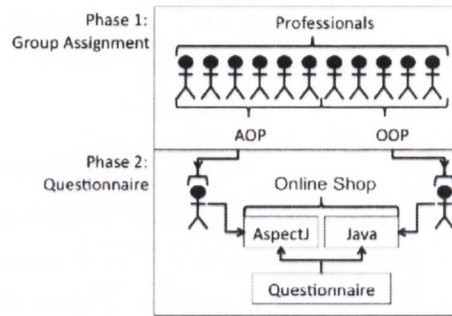


Figure 2.3: Bartsch and Harrison Experiment

2.3.1 Study

Figure 2.3 shows the two phases of the study. In the first phase, study participants are assigned into groups to work on AOP and OOP implementations of a simplified version of an online shop program. In the second phase, participants of each group were asked to answer a questionnaire. The questionnaire defined both analytical tasks and a task in which changes are applied to the implementation.

Group Assignment

Eleven professional software engineers with between two and five years experience took part in this study. None of the participants had any prior experience of AOP. To ensure that these professionals were equally able to understand and apply changes to both AspectJ and Java implementations of a program, a series of five introductory sessions in an online tutorial based on AspectJ was used. Each participant was randomly assigned to an AspectJ or Java implementation of a highly simplified version of an online shop program.

Questionnaire

Based on their assigned implementation, each participant was then asked to fill in a questionnaire which asked the participant to: identify all classes and aspects in the source code (Q1); identify the output of the software (Q2); implement a new requirement (Q3); and rate the understandability on a scale of 1 to 5 (Q4). This questionnaire was based on refinements of an initial questionnaire use in a pilot and pre-pilot tests.

Analysis

The answers to Q1, Q2 and Q3 were compared for both Java and AspectJ implementations based on measures of the time taken to answer each question and the accuracy of answer.

2.4. LOPES AND BAJRACHARYA

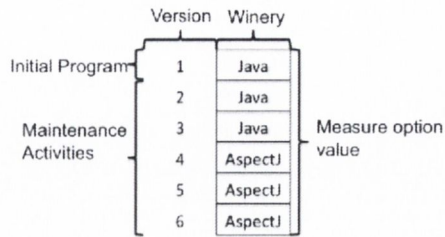


Figure 2.4: Lopes and Bajracharya Study

The comparison of the answers for Q 1-3 indicated that there were very few differences in the accuracy of the answers but showed there was a large amount of variation in the amount of time taken to answer these questions. For this reason the accuracy was dropped as a point of comparison. For Q3 an additional point of comparison is the number of lines of code that needed to be changed to implement a new requirement. The ratings in response to Q4 were directly compared for both Java and AspectJ implementations.

Statistical summaries represented as numeric tables and boxplots of these measures for Q1-4 are presented to facilitate ease of comparison. The significance of observed differences between measures for AspectJ and Java is tested using the MannWhitney and T tests [121, 41]. The MannWhitney and T tests indicate whether the two median or mean values for these measures are significantly different for AspectJ and Java implementations.

2.3.2 Empirical Evidence

For Q1, the median and mean time taken to identify all classes and aspects is the same for both AspectJ and Java implementations. For Q2, there is no significant difference between the median and mean time taken identify the output of the software for the AspectJ and Java implementations. Similarly for Q3, the median and mean time taken and lines of code changed to implement a new requirement were not significantly higher for the AspectJ implementation. For Q4, the medians are the same but there is a higher level of variation in the ratings for AspectJ. These comparisons suggest that there is no significant difference between the effects of AOP and OOP on analysability and changeability.

2.4 Lopes and Bajracharya

Lopes and Bajracharya [97] present a study that focuses on comparing the changeability of AOP and OOP implementations in terms of their value. The value of an implementation is measured in terms of the options it provides for extension and refactoring. The less dependent the modules that comprise an implementation are on one another the more extension and refactoring options there are and more valuable the implementation is.

2.4.1 Study

In this study five maintenance activities are applied to a web-based winery locator implementation. Figure 2.4 illustrates the six versions of the implementation that result. The maintenance activities differ in their goals but also in the programming language used to apply each maintenance activity.

Program and Maintenance Activities

The initial implementation provides very basic features of the winery locator program and is written in Java. The first two maintenance activities extend the set of features provided by the initial implementation. These extensions are applied using Java. The first maintenance activity adds new features. The second maintenance activity introduces a logging service.

The next three maintenance activities refactor the extended version of the winery locator. These extensions are applied using AspectJ. In the first refactoring, aspects are introduced to decouple core modules in the implementation. In the second refactoring, logging and an authentication feature are refactored using aspects. In the third and final refactoring, the web front for the program is refactored to introduce aspects to decouple the web front from the core application.

Measurement and Analysis

For each version of the implementation, a design structure matrix is constructed. This matrix represents the dependencies between the modules and interfaces that make up the implementation. The value of the implementation is calculated based on the options available for replacing modules and extending. These options are identified by analysing the design structure matrix and their value is measured using a model typically used in financial context. The impact of a maintenance activity is measured by calculating the difference in the value of the implementations before and after the maintenance activity is applied.

The analysis approach in this study is rather simple. The impacts of each maintenance activity is deemed positive if the value of the implementation increases and negative if it decreases. The effects of the Java and AspectJ based maintenance activities are compared based on whether they result in positive or negative effects.

2.4.2 Empirical Evidence

In this study, the extensions to the Java implementation both increase its value. The first two of the three AspectJ based refactorings increase the value over the Java based extensions to the implementation. This comparison provides evidence to indicate that

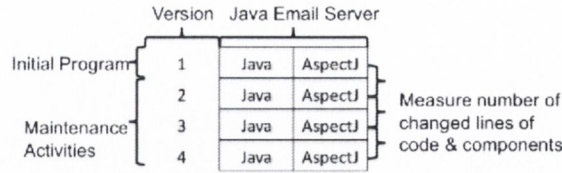


Figure 2.5: Li et al. Study

AOP has the potential to improve the value of the options to facilitate change. This evidence suggests that changeability is improved using AOP. The final AspectJ based refactoring decreases the implementations value. This refactoring illustrates the misuse of AOP constructs and provides heuristics to aid developers using AOP to avoid similar misuse.

2.5 Li et al.

Li et al. [92] present a study that gathers empirical evidence of the comparative effect of AOP and OOP on changeability.

2.5.1 Study

The study is based on measuring the size of the impact of applying three maintenance activities to AOP and OOP implementations of a program. Figure 2.5 illustrates the three maintenance activities cumulatively applied to initial AspectJ and Java implementations of Java Email Server.

Program and Maintenance Activities

This is an open source email server written in Java which has 21 classes and 1400 Lines Of Code (LOC). The first of the three maintenance activities adds a spam filtering feature to the email server. The second change refactors the logging system and the third change replaces the implementation of the spam filtering feature.

Measurement and Analysis

The impact of each maintenance activity is measured by counting the number of modules and lines of code that are changed when a maintenance activity is applied. Changeability is indicated by these measures. The lower the measure the easier it is to apply the maintenance activity. The measures for each maintenance activity are simply compared to assess the differing effects of AOP and OOP on changeability.

	Modules		LOC	
	AspectJ	Java	AspectJ	Java
Extension	1	2	44	36
Change	1	12	162	184
Change	1	1	15	15

Table 2.1: Li et al. Change Impacts

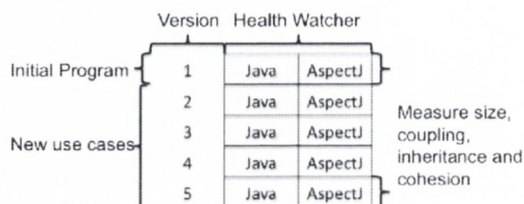


Figure 2.6: Kulesza et al. Study

2.5.2 Empirical Evidence

Table 2.1 illustrates the number of modules and LOC changed when applying each maintenance activity for the AspectJ and Java approach. This table shows that fewer modules and LOC need to be changed in the AspectJ implementation. Based on this table, Li et al. conclude that using AOP to apply changes can improve changeability over OOP.

2.6 Kulesza et al.

Kulesza et al. [87] present a study based on comparing the effects of AOP and OOP on stability.

2.6.1 Study

As illustrated in Figure 2.6, in this study five maintenance activities are applied to AspectJ- and Java-based implementations of an online public health care system, called the Health Watcher (HW) ¹.

Program and Maintenance Activities

The initial HW program is based on thirteen use cases and is comprised of a rich set of general concerns including, distribution, persistence and concurrency and is 5K LOC in size. In each of the five maintenance activities, an additional use case is implemented, extending the features of the HW.

¹The Health Watcher is also the program on which the study presented in this thesis is based

The same level of expertise in designing the implementation is used to show that the only differences between these implementations is the approach used to develop them. The HW implementations were developed independently of this study and were selected to mitigate any bias towards either implementation.

Measurement and Analysis

The metrics were applied to the initial and final versions of the AspectJ and Java implementations of the HW. The levels of coupling, inheritance, cohesion, concern diffusion and the size of each implementation is measured. The measures taken from the AspectJ and Java implementations of the initial and final versions of the HW are comparatively analysed by calculating the percentage difference between the measures gathered from implementations of both versions. These percentage differences are graphed for both initial and final versions of the HW. Those graphs are compared manually by identifying changes in the percentage difference for both initial and final versions of the HW.

2.6.2 Empirical Evidence

The resulting measures from the initial implementations indicate the AspectJ implementation is smaller in size but is dispersed across more modules, its modules are 4% less coupled, has a 2% smaller inheritance hierarchy and is 8% less cohesive. The measures from the implementations after the maintenance indicate the only notable change in size is a decrease in the relative number of attributes by 3% in the AspectJ implementation. The only other changes are that coupling in the AspectJ implementation decreases by a further 2% to 6% and cohesion is further decreased by 7% to 15%.

Before the maintenance activities are applied, the AspectJ implementation is smaller, less coupled and has a smaller inheritance hierarchy than the Java implementation. These are interpreted as indications that the AspectJ implementation is of higher quality. After the maintenance activities are applied to the AspectJ implementation, each of these indicators are improved indicating that over these maintenance activities the quality of the AspectJ implementation improves relative to the Java implementation. Based on these observations, Kulesza et al. conclude that the effect of using AOP is to increase changeability and stability compared to OOP.

Cohesion is the only measure that is worse for AspectJ both before and after the maintenance activities. Kulesza et al. claim that the measure of cohesion they use is not representative of how functionally cohesive modules are in an implementation. They argue that this measure of cohesion, which is based on the density of the relationships between methods and attributes in a module, does not capture functional cohesion.

To fully capture cohesion they use concern diffusion measurements that identify the extent to which concerns are diffused over modules, methods and lines of code. These mea-

		Version	Health Watcher		
Initial Program	1	Java	AspectJ	CaesarJ	Measure size, coupling, inheritance, cohesion and diffusion
	2	Java	AspectJ	CaesarJ	
	3	Java	AspectJ	CaesarJ	
	4	Java	AspectJ	CaesarJ	
Maintenance Activities	5	Java	AspectJ	CaesarJ	
	6	Java	AspectJ	CaesarJ	
	7	Java	AspectJ	CaesarJ	
	8	Java	AspectJ	CaesarJ	
	9	Java	AspectJ	CaesarJ	
	10	Java	AspectJ	CaesarJ	

Figure 2.7: Study of Greenwood et al.

asures are calculated by manually inspecting the source code and identifying or shadowing the modules, methods and lines of code that implement each concern. These measures are gathered for distribution, persistence and concurrency crosscutting concerns in both implementations to identify how cohesively these functions are implemented in the AspectJ and Java implementation. The resulting concern diffusion measures show that these concerns are implemented more cohesively in the AspectJ implementation.

2.7 Greenwood et al.

Greenwood et al. [63] present a study similar study to Kulesza et al. [87], which is based on comparing the effects of AOP and OOP on stability.

2.7.1 Study

As illustrated in Figure 2.7, nine maintenance activities are applied to CaesarJ-, AspectJ- and Java-based implementations of the same Health Watcher (HW) program.

Program and Maintenance Activities

In this study, Greenwood et al. indicate that the reason that they also focus on the Health Watcher is because it was developed using a high level of expertise to ensure stability. It was also used by Kulesza et al. and others allowing the results of the study to be correlated with the results of these previous studies. Greenwood et al. also reveal that the Health Watcher had been deployed in March 2001 and since then a number of incremental corrective, adaptive and perfective maintenance activities have been applied to the HW. The maintenance activities in this study were based on the real maintenance activities applied to the deployed Health Watcher implementation.

Measurement and Analysis

The level of coupling, cohesion and concern diffusion for AspectJ, CaesarJ and Java implementations over each versions of the HW are measured. For each measure, a graph plotting each measure is provided. These graphs enable the changes in the measure for the AspectJ, CaesarJ and Java implementations to be directly compared over the nine maintenance activities. Based on these graphs, the impacts the maintenance activities have on the implementation can be determined and compared. Higher levels of impact imply a less stable implementation.

2.7.2 Empirical Evidence

The graphs show that the stability of the AspectJ and CaesarJ implementations was similar to the Java implementations in terms of size. Although all three implementations consistently increase in size over the maintenance activities, the increases in the size of AspectJ and CaesarJ implementations were consistently smaller in terms of their lines of code. Greenwood et al. interpret this difference as not being significant.

The stability of the AspectJ and CaesarJ implementations was very different to the Java implementations in terms of coupling and cohesion. The overall trend for all three implementations is an increase in coupling and cohesion. Over the maintenance activities, the AspectJ and CaesarJ implementations are more stable because the levels of coupling and cohesion do not change as much as they do in the Java implementation. This indicates that the impact in terms of coupling and cohesion on the AspectJ and CaesarJ implementations is lower than the Java implementation, which suggests that the AspectJ and CaesarJ implementations are more stable.

Stability is also compared in terms of concern diffusion. To compare the impacts on concern diffusion over the maintenance activities the changes in the level of diffusion for crosscutting and non-crosscutting concerns are compared. The level of change in the diffusion of both types of concerns varied. Some crosscutting concerns were more stable in the AspectJ and CaesarJ implementations, but no stability gains were observed for other crosscutting concerns.

Based on comparing each measure taken from AspectJ, CaesarJ and Java implementations over maintenance activities, Greenwood et al. conclude that when compared to OOP the effect of AOP is to increase stability.

2.8 Figueiredo et al.

Figueiredo et al. [54] compare the effects of AOP and OOP on stability in a software product line context.

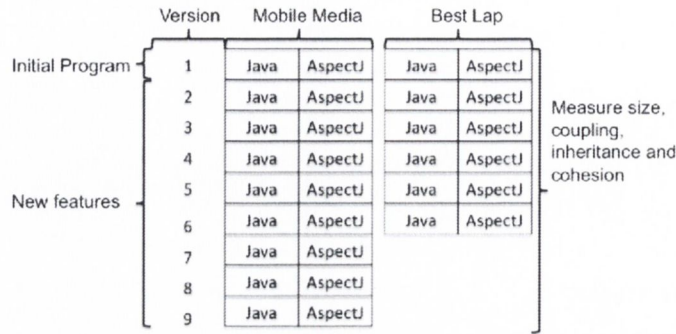


Figure 2.8: Study of Figueiredo et al.

2.8.1 Study

In this study, the stability of AspectJ and Java implementations of two heterogeneous product lines are compared.

Program and Maintenance Activities

The first product line, called Mobile Media, is for applications with 3K LOC that manipulate photo, music, and video on mobile devices, such as mobile phones. Mobile Media was independently developed and used in other research studies. The second, called Best Lap, is for mobile game applications with 10K LOC which can be deployed on a number of different mobile devices. Best Lap was developed in industry. Both AspectJ and Java implementations of the Mobile Media and Best Lap product lines were developed following best practice to ensure a high level of stability.

As illustrated in Figure 2.8, there were a number of maintenance activities defined for both product lines. These involved the introduction of new features into the product line. For the Mobile Media product line, there were eight maintenance activities and there were five maintenance activities for Best Lap.

Measurement and Analysis

The level of coupling, cohesion, concern diffusion and the size of each both implementations of each version of the two programs are measured. Similar to Greenwood et al., graphs are provided in which these measures are plotted. These graphs enable the changes in the measure for the AspectJ and Java implementations to be directly compared over the nine maintenance activities.

The impact is analysed from two perspectives. The first perspective compares the impacts of maintenance activities applied to both AspectJ and Java implementations. The number of modules, methods, lines of code and feature composition specifications

2.9. EVIDENTIAL GAP

	Analysability	Changeability	Stability	Testability
Walker and Murphy et al.	△	▽		
Bartsch and Harrison	○	○		
Lopes and Bajracharya		△		
Li et al.		△		
Kulesza et al.		△	△	
Greenwood et al.			△	
Figueiredo et al.			△	

Table 2.2: Indicator Coverage

that are changed for each maintenance activity are counted and compared. The second is based on measuring the impact on the diffusion of features over modules, methods and lines of code in both implementations for both implementations.

2.8.2 Empirical Evidence

The AspectJ and Java implementations of the Mobile Media product line are analysed from the first perspective. Of the maintenance activities defined for this product line, two are applied to mandatory features, three are applied to optional features and two are applied to alternative features. For each maintenance activity, the number of modules, methods, lines of code and feature composition specifications that are changed are counted and compared. The results show that the impact on the AspectJ implementation is lower when changes are applied to optional and alternative features, but higher when applied to mandatory features. This is because the separation of concerns in the AspectJ implementation makes it easier to localises the effects of change.

Analysis from the second perspective focuses first on analysing features in both the Mobile Media and Best Lap product lines. Specific features are analysed in terms of their diffusion in the AspectJ and Java implementations. Diffusion is measured over modules, methods and lines of code. This shows AspectJ provides superior stability for features with no shared implementation.

Based on the measure of impact taken from the AspectJ and Java implementations over maintenance activities, Figueiredo et al. conclude that when compered to OOP the effect of AOP is to increase stability.

2.9 Evidential Gap

The primary goal of this chapter is to demonstrate the evidential gap left by identifying the evidence contributed by existing studies. In this section, the contributed evidence identified for each study is summarised. This summary is used to demonstrate the evidential gap.

2.9.1 Existing Evidence

Table 2.2 lists the studies that provide empirical evidence of the comparative effects of AOP and OOP on maintainability indicators. These studies provide evidence to suggest that AOP can improve analysability, changeability and stability over OOP. The arrows used in the table indicate whether the evidence indicated that AOP lead to improvements in a maintainability indicator and the circle indicates that no (significant) difference between the effects of AOP and OOP was identified.

Although more studies are needed to fully validate this existing evidence, it is encouraging for those considering the adopting AOP to improve maintainability and reduce maintenance costs.

2.9.2 Testability Gap

However, Table 2.2 clearly shows that these studies do not provide any evidence of the comparative effect of AOP and OOP on testability. Testability is the most important indicator of maintainability, as it can have a significant effect on maintenance costs [33, 22, 141]. The claim that AOP improves maintainability cannot be tested without evidence of the comparative effect of AOP and OOP on testability. Considering the adoption of AOP based on an untested claim cannot be objective [39, 25]. The study presented in this thesis, provides evidence of testability enabling the adoption of AOP to be more objectively considered.

2.10 Common Evidence Gathering Approach

The secondary goal of this chapter is to identify an approach to gathering evidence that is applicable to the study presented in this thesis. In this section, an approach that is commonly used to gather evidence of the effects of AOP and OOP on the key indicators of maintainability is identified and its applicability to this study is discussed.

The goal of each study described in this chapter is to gather evidence of the effects of AOP and OOP on analysability, changeability and/or stability over maintenance activities. Figures 2.9 and 2.10 illustrate the commonly used approach to gather this evidence.

2.10.1 Equivalence

In this approach, maintenance activities are cumulatively applied to AOP and OOP implementations of a program. The initial AOP and OOP implementations of a program are equivalent in that differ only in the approach used for their development. Equivalence is assured by fixing all other factors that can cause the implementations to differ. For equivalence to be assured, the implementations are developed to the same level of expertise,

2.10. COMMON EVIDENCE GATHERING APPROACH

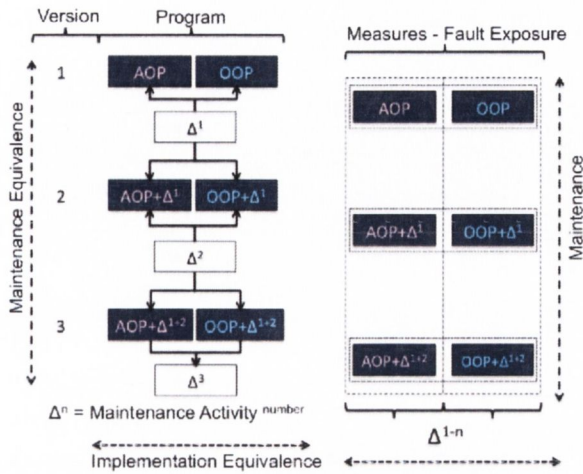


Figure 2.9: Approach

Figure 2.10: Measures

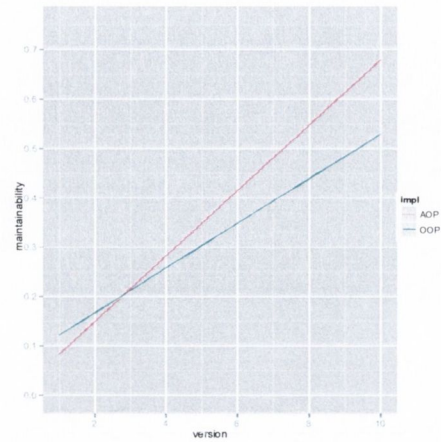


Figure 2.11: Result

satisfy the same requirements, expose the same interface and produce the same outputs for a given input [63].

The same maintenance activities are cumulatively applied to both AOP and OOP implementations. After each maintenance activity is applied to both implementations, a new version of these implementations results. The new version of these implementations are equivalent because the maintenance activity is applied to both implementations to the same level of expertise. This means that the versions of the AOP and OOP implementations are also equivalent. When all maintenance activities are applied, the only difference between each respective version of the AOP and OOP implementations is a maintenance activity.

2.10.2 Effects

Figure 2.10 illustrates the measures taken from the AOP and OOP implementations of each version of the program. Because each pair of AOP and OOP implementations are equivalent, the difference between each pair of AOP and OOP measures represents the difference in the effects of AOP and OOP of the implementation factor on the measure. Because each respective version of the program is equivalent, the difference between the AOP and OOP measures for each version represents the effects of the maintenance factor on these measures.

Figure 2.11 illustrates the typical result (as identified in Section 2.9) of analysing the measures gathered by the studies that follow this approach. This shows that AOP improves maintainability over OOP. In this analysis the effects of AOP and OOP on maintainability are identified by observing the difference in the measures for AOP and OOP over versions of the program.

Graphical and statistical analysis approaches are used to identify these effects in the studies presented in this chapter. Graphical analysis enables a more intuitive but inaccurate comparison while statistical analysis enables a more formal and accurate quantitative comparison. As will be detailed in Section 4.3, these analysis approaches are good at identifying a difference between combined effects. They do not separate combined effects and provide means to accurately quantify these effects.

2.10.3 Applicability

The goal of the study presented in this thesis is to gather evidence of the effects of AOP and OOP on testability over maintenance activities. The commonly used approach, outlined in this section, can be used to achieve this goal. The application of this approach enables the testability of equivalent AOP and OOP implementations of a program over equivalent maintenance activities to be gathered and analysed to identify and compare the effects of AOP and OOP on testability over maintenance activities.

2.11 Chapter Summary

The first section of this chapter justified the focus on empirical studies by identifying the inaccuracies of using predictive object oriented metrics. In the body of the chapter, each of the studies that contributes empirical evidence was described. For each study, the way in which it gathers evidence and the empirical evidence it contributes were identified. The chapter was concluded by collating the empirical evidence contributed by each study to show that there is no empirical evidence of the comparative effect of AOP and OOP on testability. This evidential gap motivates the study presented in this thesis.

This chapter also identified an approach that is commonly used to gather evidence for of the comparative effect of AOP and OOP on the key indicators of maintainability and discussing the applicability of this approach to this study. The approach is the basis for the approach used in this study, presented in this thesis, to gather evidence for of the comparative effect of AOP and OOP on testability.

Chapter 3

Testability

Testing is the process of executing the implementation of a program with the intent of exposing faults [110]. A fault is a deviation from the intended program, and is exposed through test failure [110]. A program can hide faults by not causing tests to fail when faults are present [148]. The testability of an implementation is the ease with which faults can be exposed through testing [148].

Figure 3.1 asks three questions the answers to which are the basis for methodology and motivation of the study presented in this thesis. This chapter answers these three questions.

To answer the first question, *how are faults exposed?*, a model to explain the conditions that cause fault exposure is presented. This model is used as a basis for determining the causation of observed rates of fault exposure in the next chapter.

To answer the second question, *what are the factors that influence fault exposure?*, each factor is identified and its influence on fault exposure is demonstrated. These factors are the basis of the measurement approach and design used in this study, also presented in Chapter 4.

To answer the third and final question, *what evidence has been contributed by existing studies of testability?*, an overview of the studies and the evidence they contribute is presented. This overview reaffirms the gap identified in the pervious chapter.

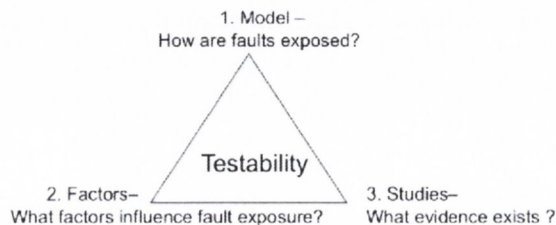


Figure 3.1: Testability


```

1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){
5         return this.size;
6     }
7     public void setSize(int size) {
8         if(size >= 0){
9             this.size=size; // fault size=size - "this" is deleted
10            if(size > Stack.max){
11                this.size = Stack.max;
12            }
13        } else {
14            throw new IllegalArgumentException();
15        }
16    }
17 }

```

Listing 3.1: Fault

3.1 Fault Exposure Model

A fault exposure model identifies the conditions in which tests expose faults in an implementation. This model is used to analyse and compare the causes of fault exposure for AOP and OOP implementations in this study.

Listing 3.1 illustrates the Java code based on a `Stack` class example used by Ma et al. [98]. Listing 3.2 illustrates tests to validate the correctness of the implementation of the `Stack` class. The comment on Line 9 of Listing 3.1 of the listing illustrates a fault. In this section, this example is used to demonstrate a model of fault exposure through testing.

3.1.1 Faults

Listing 3.1 shows the source code of the `Stack` class with two accessor methods. In the `setSize` method, at line 9, there is an assignment to the stack's `size` attribute. In the assignment, the `size` parameter to the `setSize` method is assigned to the `size` attribute. To differentiate with the parameter, the attribute is referenced using the `this` keyword.

In a new version of the `Stack`, a fault is deliberately created in the `setSize` method, for the purpose of analysing how easy the fault is to expose and therefore how testable the `Stack` class is. The fault is illustrated in the comment on line 9 of the listing. The fault occurs when the `this` keyword is deleted. Deleting the `this` keyword makes the target of assignment the `size` parameter rather than the `size` attribute. This fault is likely to lead to an incorrect value for the `size` attribute, declared at line 2 of the listing.

Figure 3.2 illustrates a control flow graph of the `setSize` method. The nodes in the control-flow graph represent locations, or lines of source code, of the `Stack` class presented in Listing 3.1. The location containing the fault is coloured red. The directed

3.1. FAULT EXPOSURE MODEL

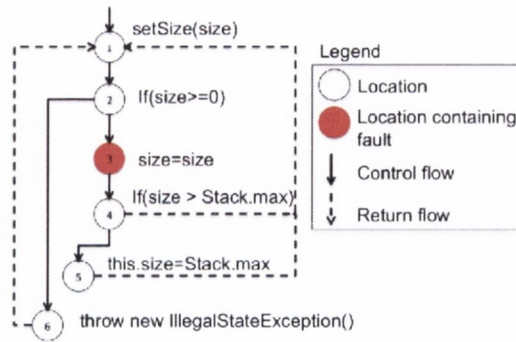


Figure 3.2: Control-flow graph for `setSize` method

arrows between nodes represent the flow of execution between nodes. Arrows with a solid line represent control-flow and arrows with a broken line represent a return of control. This type of control-flow graph is used and described in detail by Ammann and Offutt [5].

Location 2 in Figure 3.2 shows a precondition that needs to be met before the body of the method executes. This defines zero as the minimum size the stack can be. If the value of the `size` parameter is less than zero then this results in the execution of location 6, where an exception is thrown. If the value of the `size` parameter is greater than or equal to zero, then location 3 is executed. Location 3 is where the `this` deletion fault occurs. Location 4 is executed after location 3. Location 4 defines a condition defining the maximum size of the stack to be five. If the value of the `size` parameter is greater than five then then the `size` attribute is reset to a value of five by executing location 5 and returning. If the value is less than or equal to five the method returns.

3.1.2 Tests

Listing 3.2 shows four tests (1, 2, 3 and 4) to expose this fault. Each test calls the `setSize` method to set the size of the stack and the `getSize` method to ensure that the size has been properly set. The goal of the combined set of tests is to ensure the correctness of the code. If the program is testable, then the fault at location 3, illustrated in Figure 3.2, will be exposed by the tests.

Figure 3.3 illustrates the execution of each test against the `setSize` method on an instance of the `Stack` class. The top of Figure 3.3 illustrates the values for the `size` parameter ($p:size$) and attribute ($a:size$) before each test executes the `setSize` method on an instance of the `Stack` class. All tests execute an instance of the `Stack` class in which the `size` attribute is initialised to zero. The value of the `size` parameter differs for each test.

```

1 public class StackTest extends TestCase {
2   Stack stack = new Stack();
3   public void test1(){
4     try{
5       stack.setSize(-1);
6       fail();
7     }catch(Exception e){...}
8   }
9   public void test2(){
10    stack.setSize(0);
11    assertEquals(0,stack.getSize());
12  }
13  public void test3(){
14    stack.setSize(11);
15    assertEquals(5,stack.getSize());
16  }
17  public void test4(){
18    stack.setSize(1);
19    assertEquals(1,stack.getSize());
20  }
21 }

```

Listing 3.2: Tests 1 - 4

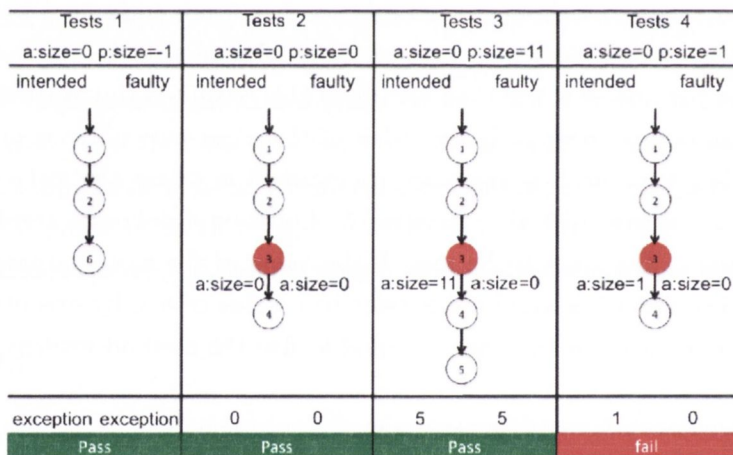


Figure 3.3: Control-flow paths through `setSize` method for tests 1 - 4

The middle of Figure 3.3 illustrates the control-flow path through the `setSize` method exercised by each test. This shows whether the location containing the fault, location 3 is executed by the test or not.

For the tests that do execute this location, two values of the `size` attribute, `a.size`, are shown directly after the execution of the location. On the left hand side of the control-flow path, the value of the `size` attribute is shown when the `setSize` method does not contain the fault at location 3. On the right hand side of the control-flow path, the value of the `size` attribute is shown when the `setSize` method contains the fault at location 3.

3.1. FAULT EXPOSURE MODEL

A comparison of these values shows whether the fault results in an incorrect value for the `size` attribute at this stage of execution for each test. If the value for the `size` attribute is correct at this stage, the fault will not cause the actual output of the test to differ from the expected output. If it is incorrect, then this may cause the actual output to differ from the expected output. The bottom of the figure presents the expected and actual outcomes for each test. If the actual outcome differs from the expected outcome, the test fails and the fault is exposed.

Test 1

Test 1 sets the value of the `size` parameter to minus one. This is below the minimum size which results in a control-flow path that does not execute the faulty location. This ensures that the state of the `Stack` class is correct. The outcome of test execution is that an exception is thrown, as expected. The test passes and the fault is not exposed.

Test 2

Test 2 sets the value of the `size` parameter to zero. This is not below the minimum size which results in a control-flow path that does execute the faulty location.

In the test, the `size` attribute is initialised to zero. Because the input to the `setSize` method is also zero, the fault does not cause the state of the `size` attribute to become incorrect. This is illustrated where the value of the `size` attribute is presented after the intended and faulty version of location 3 is executed. This shows that when the fault is absent or present the value of `size` attribute is zero after location 3 is executed.

The value of the `size` parameter does not exceed the maximum size and as such returns after location 4 is executed. The output of the test execution is zero which is as expected. The outcome is that the test passes and the fault is not exposed.

Test 3

Test 3 sets the value of the `size` parameter to eleven. This is above the minimum size which results in a control-flow path that executes the faulty location.

The fault causes the parameter value of eleven to not be assigned to the `size` attribute. This causes the state of the `size` attribute to become incorrect directly after the faulty location is executed. This is illustrated in the figure where the value of the `size` attribute is presented after the intended and faulty version of location 3 is executed. This shows that when the fault is absent value of the `size` attribute is eleven after location 3 is executed and when it is present the value of the `size` attribute is zero.

The value of the `size` parameter does exceed the maximum size and as such location 5 is executed. Location 5 resets the `size` attribute to a value of 5, masking the fault and

subsequent incorrect value of the `size` attribute. The output of the tests execution is 5 which is as expected, the outcome of which is test passing.

Test 4

Test 4 sets the value of the `size` parameter to one. This is above the minimum size which results in a control-flow path that executes the faulty location.

The fault causes the parameter value of one to not to be assigned to the `size` attribute. This causes the state of the `size` attribute to become incorrect directly after the faulty location is executed. This is illustrated in Figure 3.3 where the value of the `size` attribute is presented after the intended and faulty version of location 3 is executed. This shows that when the fault is absent, the value of the `size` attribute is one after location 3 is executed and when it is present, the value of the `size` attribute is zero.

The value of the `size` parameter does not exceed the maximum size and returns after location 4 is executed. The output value of the tests execution is zero, which differs from the expected value of one. This causes the test to fail.

3.1.3 Model

This simple example is used to illustrate the model for fault exposure which was previously identified in the Relay [128] and later in the PIE models [148, 146, 1] of testability. This model specifies that for a fault to be exposed through testing, a test input must cause the location at which the fault occurs to be executed. The execution of the fault must cause the state after the location's execution to become infected. State is infected when it deviates from what it would be at this point of execution in the intended implementation. This infected state must then be propagated into the output. If the test's output deviates from what it is expected to be the test fails exposing the fault.

Table 3.1 illustrates the model by showing whether tests 1 to 4 executed the location containing the fault; caused the value of the `size` attribute of the `Stack` class to become infected; caused the infected state to become propagated; and whether or not the test exposed the fault through test failure.

Test 1 does not execute the location at which the fault occurs and as such, cannot cause infection or propagation, resulting in a pass. Although Test 2 does cause execution, it does not cause the state of the `Stack` class to become infected. This is because value of the `size` attribute is the same as when the fault does not occur. As there is no state infection caused by Test 2, the infection cannot be propagated and the test does not expose the fault. In contrast, Test 3 does cause execution and infection, but not propagation and as such, does not expose the fault. Test 4 does expose the fault because it is the only test to cause execution of the fault, infection of the state directly after the fault is executed and propagation of this infected state to the tests output.

3.2. FACTORS OF TESTABILITY

Test	Execution	Infection	Propagation	Exposure
1	○	○	○	no
2	●	○	○	no
3	●	●	○	no
4	●	●	●	yes

Table 3.1: Requirements for Fault Exposure

This model is used to demonstrate the conditions that result in the exposure of a fault through testing. The model can be used to explain the reasons why faults are exposed. For example, a rate of execution of locations in an implementation containing faults can be used to explain a high rate of fault exposure for the implementation. This model is useful when analysing the causes of observed rates of fault exposure for an implementation.

3.2 Factors of Testability

There are three factors that have an effect on testability: fault type, tests and implementation [76]. In this section, an extension of the `Stack` class, used in the previous section, is used to illustrate each of these factors.

3.2.1 Fault Type

The accidental deletion of the `this` keyword is an instance of a type of fault observed in practice. In the example, the fault type is the accidental deletion of the `this` keyword. The instance of this type of fault is its occurrence in the `setSize` method of the `Stack` class.

Listing 3.3 extends the number of faults identified in the `setSize` method of the `Stack` class in Listing 3.1. In this listing, there are five faults identified in the `setSize` method in comments. These faults are of two different types: insertion and deletion of the `this` keyword. The same four tests (1 - 4) illustrated in Listing 3.2 are executed against implementations of the `Stack` class each containing one of the five identified faults.

The outcomes of applying each test to each faulty version of the `Stack` are illustrated in Figure 3.4. Table 3.2 summarises the outcomes illustrated in Figure 3.4 by identifying the instances of each fault type and the rate of exposure for these fault instances.

The first row of the table shows that the faults where the programmer inserts a `this` keyword in error are fully exposed. The second row shows that 0.67 of the faults where the programmer deletes a `this` keyword in error are exposed. This shows that in this trivial example, faults of the insertion type of fault is easier to expose than faults of the deletion type.

The third and final row shows that when the faults of both types are considered the rate of exposure is 0.8. These different rates of exposure for the different fault types show


```

1 public class Stack {
2   int size = 0;
3   static final int max = 5;
4   public int getSize(){
5     return this.size;
6   }
7   public void setSize(int size) {
8     if(size >= 0){ // if(this.size >= 0) fault 1 insertion
9       this.size=size; // size=size fault 2 deletion
10      // this.size=this.size fault 3 insertion
11      if(size>Stack.max){ // this.size> fault 4 insertion
12        this.size=Stack.max; size= // fault 5 deletion
13      }
14    } else {
15      throw new IllegalArgumentException();
16    }
17  }
18 }

```

Listing 3.3: This Insertion and Deletion Faults

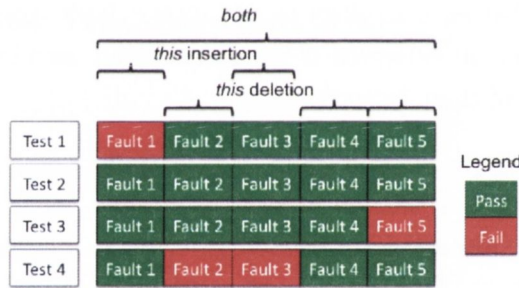


Figure 3.4: Fault Factors

Fault Type	Faults	Exposure
insertion	1,3	$\frac{2}{2}$
deletion	2,4,5	$\frac{2}{3}$
both	1 - 5	$\frac{4}{5}$

Table 3.2: Effects of Differences in Fault Factor

that fault type is a factor that influences the rate of exposure in an implementation.

3.2.2 Tests

The tests illustrated in Listing 3.2 and described in Section 3.1 each execute the `setSize` method of the `Stack` class with an input selected from the domain of all possible inputs to this method. The full set of all possible inputs is the set of all integer values. The inputs are selected from this set to exercise each control-flow path through the method.

Figure 3.3 shows the control-flow paths through the `setSize` method executed by each

3.2. FACTORS OF TESTABILITY



Figure 3.5: Test Factor

Tests	Tests	Exposure
A	2 - 4	0.6
B	1 - 3	0.4
C	1 - 4	0.8

Table 3.3: Effects of Differences in Test Factor

test. Figure 3.5 illustrates the outcome of applying these tests to each faulty version of the `Stack` class. To illustrate the effect of using different sets of tests on the rate of fault exposure, the rate of exposure of three sets of tests A, B and C are compared. Sets A and B are subsets of the set of all four tests illustrated in Listing 3.2, which is set C.

The outcomes illustrated in Figure 3.5 are summarised Table 3.3. This summary identifies the tests in each set and their associated rates of exposure. The Tests column of this table shows that set A is comprised of tests 2 - 4, set B is comprised of tests 1- 3 and set C is comprised of tests 1- 4. The Exposure column shows the rate of fault exposure for each set. It shows that set A exposes 0.6 of the faults, set B exposes .4 of faults and set C exposes 0.8 of faults. These results imply that set C, containing the full of four tests, expose more faults than sets B or C. This means that set C maximises the testability of the `Stack` class.

These different rates of exposure for each set show that the tests selected to assert correctness of the implementation influence the rate of exposure. This identifies the tests used to assert correctness as a factor of testability.

3.2.3 Implementation

The implementation of the `Stack` class illustrated in Listing 3.3 is one of many possible implementations of this class. For example, the pre- and post-conditions which identify maximum and minimum sizes for the stack, defined in the `setSize` method, could be removed as illustrated in Listing 3.5 or re-implemented using AspectJ as illustrated in Listing 3.4.

In the AOP re-implementation, a new implementation of the stack is created that provides precisely the same behaviour as the original. The only difference is that in the AspectJ implementation the pre- and post-conditions are implemented in `around` advice.

```

1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){
5         return this.size;
6     }
7     public void setSize(int size) {
8         this.size=size; // size=size fault 2 deletion
9         // this.size=this.size fault 3 insertion
10    }
11 }
12 public privileged aspect StackAspectPrePost {
13     void around(int size, impl2.Stack stack) :
14         execution(void impl2.Stack.setSize(int))
15         && args(size) && target(stack){
16         if(size >= 0){
17             proceed(size, stack);
18             if(size > Stack.max)
19                 {
20                 stack.size = Stack.max;
21             }
22         } else {
23             throw new IllegalArgumentException();
24         }
25     }
26 }

```

Listing 3.4: AspectJ Refactored

```

1 public class Stack {
2     int size = 0;
3     public int getSize(){
4         return this.size;
5     }
6     public void setSize(int size) {
7         if(size >= 0){ // if(this.size >= 0) fault 1 insertion
8             this.size=size; // size=size fault 2 deletion
9             // this.size=this.size fault 3 insertion
10        } else {
11            throw new IllegalArgumentException();
12        }
13    }
14 }

```

Listing 3.5: Condition Removal Refactored

The `around` advice is executed in the place of the `setSize` method. The pre- and post-conditions are executed relative to a call to `proceed`. This `proceed` call executes the `setSize` implementation.

Moving the implementation of these conditions into advice means that faults in which this is inserted or deleted are no longer relevant because the `this` keyword cannot be used in aspects. Of course moving the implementation of the conditions into advice means that new AspectJ specific faults can occur. These types of faults and their effects on the rate

3.2. FACTORS OF TESTABILITY

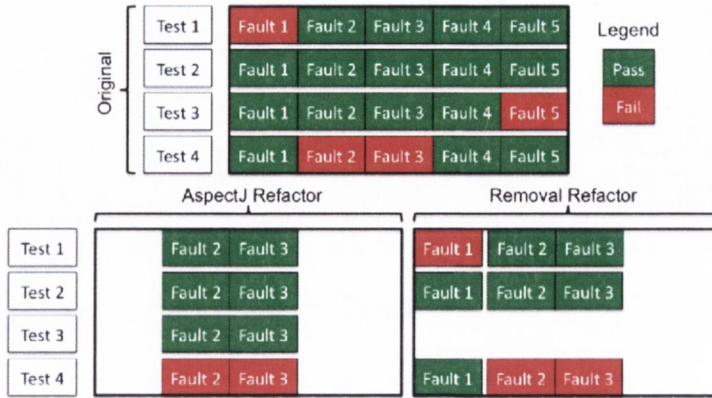


Figure 3.6: Implementation Factors

Implementation	Tests	Faults	Exposure
Original	1-5	1-5	$\frac{4}{5}$
AOP Refactor	1-4	2,3	$\frac{2}{3}$
Removal Refactor	1,2,4	1,2,3	$\frac{3}{3}$

Table 3.4: Effects of Differences in Implementation Factor

of fault exposure are presented later in Section 4.2.2. For the sake of illustration purposes only, these types of faults are not considered in this demonstrative example.

Figure 3.6 illustrates the outcome of executing the tests that are relevant for each implementation against the faults contained in each implementation. This figure and Table 3.4 show that the rate of exposure for the original implementation is 0.8, 1 for the AspectJ refactored implementation and 1 for the implementation in which the pre- and post-conditions are removed. The differences in these rates show that different implementations do have an effect on fault exposure.

3.2.4 Factors

Table 3.6 presents an overview of examples used to identify and illustrate the three factors of testability: fault type, test and implementation. In the left most column, named Factor, the factor being illustrated is identified. In the next three columns (Fault Type, Tests and Implementation) illustrate the levels of each factor used to derive the fault exposure rate, in the right most column.

The levels of each factor are identified in Table 3.5. The differences in each factor are used to demonstrate the influence fault type, test and implementation factors have on the rate of fault exposure. For example, three different sets or levels of tests (1-3, 2-4 and 1-4) are used to demonstrate the influence of the test factor on testability.

The effects of the different levels in one factor can be isolated by fixing the other two

Factor	Levels
fault Type	<i>this insertion</i>
	<i>this deletion</i>
	<i>both</i>
tests	1-3
	2-4
	1-4
Implementation	<i>Original</i>
	<i>AOP</i>
	<i>Removal</i>

Table 3.5: Factor Levels

Factor	Fault Type	Tests	Implementation	Exposed
fault Type	<i>this insertion</i>	1 - 4	<i>Original</i>	1
	<i>this deletion</i>			0.67
	<i>both</i>			0.8
test	<i>both</i>	2 - 4	<i>Original</i>	0.6
		1 - 3		0.4
		1 - 4		0.8
implementation	<i>both</i>	1 - 4	<i>Original</i>	0.8
			<i>AOP</i>	1
			<i>Removal</i>	1

Table 3.6: Overview

factors when deriving the rate of fault exposure. This pattern is important because it is the basis for the designs used in all empirical studies of testability. The pattern is also used in this thesis.

In this example, there are three instance of factor fixing. This first is where the different levels of the test factor are isolated by fixing the levels of the implementation and fault type factors. In this case, the *original* level of the implementation factor and the *both* level of fault type factor are fixed for the derivation of the fault exposure rate at each level of the test factor.

The second example of this pattern is illustrated by looking at the fault type factor in the Table 3.6. The different levels of the fault type factor are isolated by fixing the levels of the implementation and test factors. In this case, the *original* level of the implementation factor and the *1-4* level of the test factor are fixed for the derivation of the fault exposure rate at each level of the fault type factor.

The third and final example of this pattern is illustrated by looking at the implementation factor in Table 3.6. The different levels of the implementation factor are isolated by fixing the levels of the fault type and test factors. In this case, the *both* level of the fault type factor and the *1-4* level of the test factor are fixed for the derivation of the fault exposure rate at each level of the fault type factor.

The **Stack** class example shows that different levels of each factor have an effect on the rate of fault exposure and consequently testability. In each case, the effects of different levels in one factor are isolated by fixing the other two factors. This pattern

3.3. COMPARATIVE STUDIES OF TESTABILITY

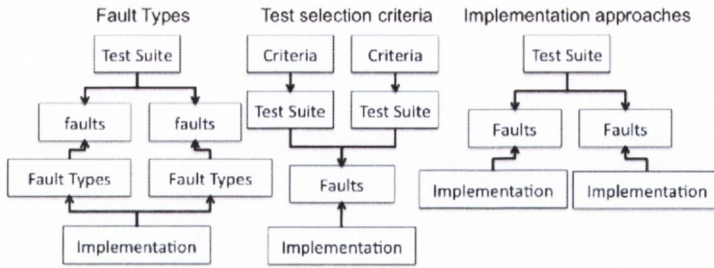


Figure 3.7: Designs of Testability Studies

is an important observation that is used as the foundation for the design of the study presented in Chapter 4.

3.3 Comparative Studies of Testability

There are a number of empirical studies that compare the effects of differences in the three factors of testability: fault type, test and implementation. This section shows that these studies do not provide evidence of the effects of AOP and OOP on testability and illustrates the use of the factor fixing pattern.

3.3.1 Fault Type

There are relatively few studies that compare the effects of different fault types on testability. However, the rates of exposure for faults of different types have been compared [112, 111]. In these studies, the rates of exposure for faults of different types are compared. The goal of these studies is to identify types of faults that have a significant effect on testability, so that the implementation or testing process can be streamlined. For example, the implementation or testing processes can be altered to ensure specific types of fault are easier to expose [60].

Although much larger in scale, these studies are similar in design to the trivial example presented in Section 3.2.1, where the rates of exposure for three different sets of fault types are compared. The studies ensure that only the effects of the different fault type sets are compared the same implementation and test set are used to calculate each rate. Using the same implementation and test set ensures that these factors are fixed.

The left hand side of figure 3.7 illustrates the design of studies that compare the effects of different fault types on testability. These studies are designed to isolate the effects of the fault type on testability. This is done by fixing the test and implementation factors. This ensures that the differences in the resulting rates are caused only by differences in test set.

3.3.2 Test

The majority of the comparative studies of testability are based on comparing different test selection criteria [24, 132, 19]. Test selection criteria are rules to guide the selection of test sets, to maximise the rate of fault exposure. The criteria compared in these studies are typically based on control- and data-flow analysis [59, 58, 72, 7].

Control-flow analysis, identifies the paths of execution through an implementation. Data-flow analysis is an extension of control flow analysis in which the flows of data along the paths of execution are identified. Test selection criteria based on these flow analyses identify the paths through the implementation that need to be exercised by a test set for it to be considered adequately tested.

In these studies, sets of test are selected to satisfy each criterion. The number of tests selected to satisfy each criterion is used as an indicator of the cost of testing with that criterion. The rate of faults exposed when the selected tests are executed against faulty implementations is used as a measure of test efficiency [59, 58, 72, 7]. These measures are then used together to quantify the cost-benefit effects of different criteria on testability.

The middle of Figure 3.7 illustrates the typical design of these studies. These studies are designed to isolate the effects of the test sets selected to satisfy each criteria. This is done by fixing the fault type and implementation factors. The use of this pattern ensures that the differences in the resulting rates are caused only by the different tests selected to satisfy each criterion.

3.3.3 Implementation

There is only one study that compares the effects of using different implementation approaches on testability. In this empirical study, Voas et al. [147] compare the effects of Object Oriented and Procedural Programming (OOP and PP) on testability.

This is done by developing OOP and PP implementations of a program. Tests, equally applicable to both implementations, are executed against both implementations to expose faults. The result of this study is a comparison of the rate of exposure for both implementations. It showed that the information hiding associated with OOP had the effect of reducing testability.

The right hand section of Figure 3.7 illustrates the design of this study. It is designed to isolate the effects of the implementation on the rate of fault exposure. This is done by fixing the test set and fault type factors. This ensures that the differences in the resulting rates are caused only by differences between OOP and PP approaches to implementation.

3.4. CHAPTER SUMMARY

3.3.4 Evidential Gap

The majority of comparative studies on testability are focused on identifying test selection criteria that yield the best cost-benefit balance. The fault type factor is of some interest and there is very little evidence of the effects of different implementation approaches on testability. The only study that compares the effects of using different implementation approaches on testability compares the effects of OOP and PP on testability and provides no evidence of comparative effect of AOP and OOP.

3.4 Chapter Summary

This chapter has described a model of fault exposure, it has identified the factors in that model that have an effect on testability and reviews the studies on the effects of differences in these factors on testability. The model and factors and factor fixing pattern provide a foundation for describing the measurement, design and analysis approaches used in the study, in the next chapter.

Chapter 4

Study Methodology

This thesis gathers empirical evidence of the comparative effect of AOP and OOP on testability through a study. This study is conducted in two phases. In the first phase, illustrated in Figure 4.1, the testability of equivalent AOP and OOP implementations is measured for each version of the program. AOP and OOP implementations are equivalent if they differ only in the approach used for their development. The results of this phase are pairs of AOP and OOP implementation testability measures, one pair for each version of the program, as illustrated in Figure 4.2. These measures are analysed in the second phase, illustrated in Figure 4.3. This analysis quantifies the comparative effects of AOP and OOP on testability.

The graph presented in Figure 4.4 illustrates the result of the measurement and analysis phases for the `Stack` class example. The lines in this graph represent the generalised effects of AOP and OOP on testability over maintenance activities. The difference between these lines is the comparative effect of AOP and OOP on testability for the trivial `Stack` class example.

The first and second sections of this chapter describe the methodology followed in the measurement phase and the third section describes methodology followed in the analysis phase. The first section describes the approach used to measure testability. The second section describes how this approach is applied to ensure the resulting pairs of measures isolate the combined effects the implementation and maintenance factors. The third section describes how the comparative effects of AOP and OOP are analysed and quantified.

4.1 Testability Measurement

In the measurement phase of the study, illustrated in Figure 4.1, the testability of the AOP and OOP implementations are measured. There are two approaches to measuring the testability of an implementation. The selection of the approach that best meets defined selection criteria is described and the selected approach is detailed.

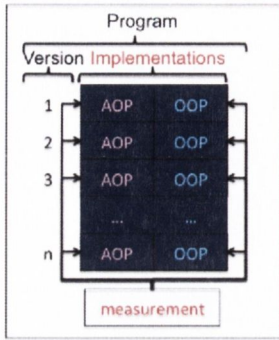


Figure 4.1: Measurement

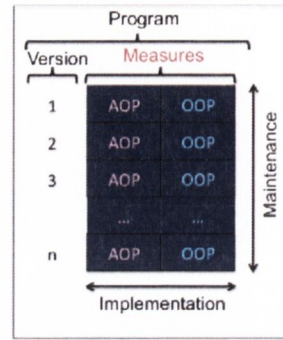


Figure 4.2: Factors

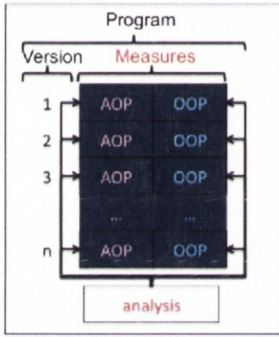


Figure 4.3: Analysis

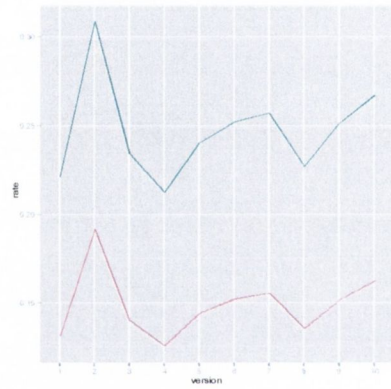


Figure 4.4: Stack Result

4.1.1 Measurement Approach Selection

The review of the empirical studies of testability is presented in Section 3.3 illustrate there are two main approaches to measuring testability. The first measurement approach uses mutation analysis to measure the rate of fault exposure in an implementation through testing. The second approach measures the number of tests that are needed to adequately test an implementation. Here, these approaches are compared against the testability selection criterion.

Selection Criterion

Testability is the ease with which faults can be exposed through testing [148]. Measures of testability are approximations of this ease [121, 24]. The goal is to select the approach that provides the most accurate approximation. This is because the more accurate the measures of testability are, the more accurate the comparison of the effects of AOP and OOP on testability will be.

The rate at which faults in an implementation are exposed through testing is used as a measure of testability [59, 58, 72, 7, 85, 150, 117, 147]. Mutation analysis [45, 31] derives

4.1. TESTABILITY MEASUREMENT

Test 1	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5
Test 2	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5
Test 3	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5
Test 4	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5

Legend

Pass
Fail

Figure 4.5: Measuring testability as the rate of fault exposure

this measure using faulty versions of the implementation. In each faulty implementation, a fault is introduced at a location. A location is a line of the implementation’s source code. A set of tests is executed against each of these faulty implementations. Test failure indicates fault exposure. Non-failure of a test indicates that fault contained in the implementation is not exposed.

The rate of fault exposure can be measured in two ways. The first is to identify the proportion of faults exposed. The second is to identify the proportion of fails over the total number of test executions. In both cases, the higher the proportion is, the easier it is to expose faults. The top part of Figure 4.5 illustrates this approach. It presents the outcomes of executing the four tests, shown in Listing 3.2, against five faulty implementations of the `setSize` method of the `Stack` class. Each of these faulty implementations contains one of the five faults illustrated in Listing 3.3.

In this example, the rate of fault exposure can be measured as the number of faults exposed taken as a proportion of the total number of faults in the implementation, which is $\frac{4}{5}$. The numerator of 4 is obtained by counting the number of faults exposed in Figure 4.5. The denominator of 5 is the number of faults in Figure 4.5. Fault exposure can also be measured as a proportion of the total number of faults in the implementation, which is $\frac{4}{20}$. The denominator of 20 is the total number of test-mutant executions illustrated in Figure 4.5.

Both measures reflect the ease with which faults are exposed through testing. Higher proportions suggest that faults in the implementation are easier to expose. In a context in which the testability of implementations is compared the latter measure is preferred because it provides a more detailed measure of fault exposure.

Number of Tests

The second approach to testability measurement is based on counting the number of tests needed to adequately test an implementation [59, 58, 72, 7, 85, 150, 117]. The key to deriving this measure is establishing when a test set is adequate.

Adequacy is determined by identifying the paths of execution through the implementation that must be exercised through testing. These paths are typically identified using

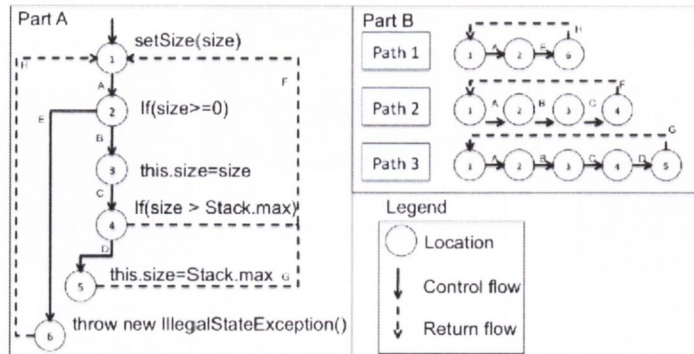


Figure 4.6: Measuring Testability as the Number of Tests Needed for Fault Exposure

control- or data-flow analysis. In these analyses, the source code of the application is analysed and a graph representing the paths through the implementation is created. A test set is considered adequate if it exercises all of the unique and complete paths through this graph.

Part A of Figure 4.6 illustrates a control-flow graph representation of the source code of the `setSize` method. Part B identifies all three of the unique and complete paths through this graph. For this method to be adequately tested, a set of tests that exercise all of these paths is required. Table 4.1 identifies which of the paths, identified in part B of Figure 4.6, are executed by the test set (illustrated in Listing 3.2). This test set is considered adequate because all of the paths are exercised by at least one test. In this case, the measure of testability is four.

This approach is based on the assumption that if a set of tests is adequate, then the faults in an implementation will be exposed. This assumption is based on the observation, demonstrated in Section 3.1, that execution is the primary requirement for fault exposure. However, Section 3.1 also demonstrates that although execution is the primary requirement, fault exposure also requires state infection and propagation of an infected state.

Figure 4.5 shows the outcome of executing the test set, identified in Table 4.1, against faulty versions of `setSize` method. Although Table 4.1 shows that these tests are considered adequate, this figure shows that only $\frac{4}{5}$ of the faults contained in these faulty implementations are exposed. This demonstrates the weakness of the correlation between the number of tests in a test set and fault exposure. The weakness of this correlation has been documented through empirical studies [72, 7]. This weakness suggests that the number of tests needed to adequately test an implementation is not an accurate measure of the ease with which faults in an implementation are exposed through testing.

4.1. TESTABILITY MEASUREMENT

Path	Tests			
	1	2	3	4
1	●	○	○	○
2	○	●	●	○
3	○	○	○	●

Table 4.1: Test Path Execution

Selection

The descriptions of the two approaches to measuring testability indicate that the rate of fault exposure, derived using mutation analysis, is a more accurate approximation of the ease with which faults are exposed through testing. The mutation analysis approach provides a measure of testability of $\frac{4}{5}$, which is accurate. The number of tests required to exposed is 4 based on the approach based on excessing control-flow paths. This is inaccurate because this number of tests does not result in full fault exposure. Mutation analysis is selected because analysis of the accurate measures of testability it provides result in an accurate comparison of the effects of AOP and OOP on testability.

4.1.2 Mutation Analysis

In Mutation Analysis (MA) [45], the rate of fault exposure of an implementation is derived by executing a set of tests against mutants of the implementation. In this subsection, mutants are described, the three phases of MA: *mutant generation*, *location execution* and *fault exposure* are outlined, and each phase is described in detail through simple examples.

Mutants

A mutant is version of an implementation that contains a fault. The fault contained in a mutant is a small deviation from the intended implementation. Mutants are created by generating a copy of the implementation and introducing a deviation into the copy.

Two examples of the deviations used to generate mutants are presented in Listing 4.1. This listing shows the implementation of the `Stack` class. Lines 9 and 11 of the source show two deviations from the intended program in which the `this` keyword is deleted. Mutants are created by making two copies of the `Stack` class implementation and introducing one of these deviations into each copy.

As will be illustrated in Chapter 5, there are different types of deviation. The type illustrated in Listing 4.1 is where the `this` keyword is deleted. This type of deviation is representative of the type of fault that occurs when a developer forgets to use the `this` keyword where its use is intended. Each deviation in which the `this` keyword is deleted is representative of a fault of this type.

```
1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){
5         return this.size;
6     }
7     public void setSize(int size) {
8         if(size >= 0){
9             this.size=size;//deviation size= "this" is deleted
10            if(size > Stack.max){
11                this.size = Stack.max;//deviation size= "this" is deleted
12            }
13        } else {
14            throw new IllegalArgumentException();
15        }
16    }
17 }
```

Listing 4.1: `this` keyword deletion deviants

Overview of MA process

An overview of the MA process is presented in Figure 4.7. This provides a high level view of the inputs and outputs of the three phases of MA.

The first phase of MA is *mutant generation*. In this phase, mutants of an implementation are generated. An implementation is taken as input. The implementation is analysed to identify locations at which deviations can be created and mutants generated. The outputs of this phase are the generated mutants and the locations at which they were generated.

The second phase of MA is *location execution*. The goal of this phase is to reduce the number of mutants that need to be executed to improve the efficiency of applying mutation analysis. The inputs to this phase are a test set and the locations identified in the *mutant generation* phase and the implementation. In this phase, each test is executed against the implementation to identify which of the locations are executed. This enables the identification of mutants that contain faults at locations that are not executed by each test. If the location at which a fault is contained is not executed then the fault cannot be exposed. The output of this phase is a list of mutants generated at locations that are executed by each test.

Fault exposure is the third and final phase of MA. The inputs to this phase are the test set used in the *location execution* phase and the list of mutants generated at locations that are executed by each test in that set. In this phase, tests are executed against the mutants generated at the executed locations. The output of this phase is the pass and fail outcomes from executing tests against mutants. A fail outcome is indicative of fault exposure.

4.1. TESTABILITY MEASUREMENT

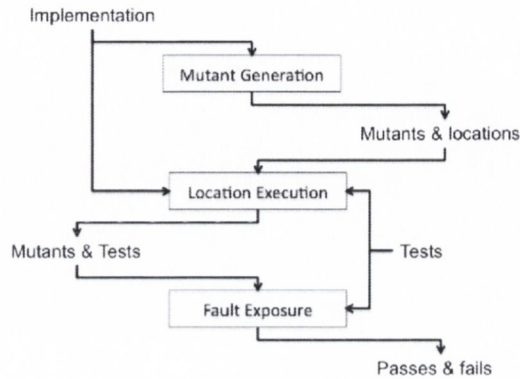


Figure 4.7: Mutation Analysis

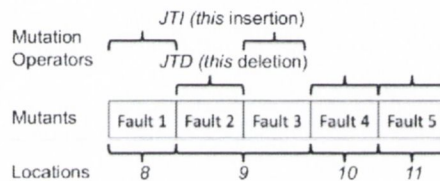


Figure 4.8: Mutation Operators Applied at Locations

Mutant Generation

Mutant generation is the first phase of MA. In this phase, mutants are generated by analysing the source code of an implementation to identify locations at which to apply mutation operators. Mutation operators create deviations at these locations and these deviations are used to generate mutants. Each mutation operator is used to create a specific type of deviation or fault. As will be demonstrated later in Section 5.2, there are mutation operators defined for different languages. The mutation operators defined for a specific language are used to generate the types of faults that occur in practice when that language is used. The output of this phase is a set of mutants of the implementation that contain faults at specific locations.

Mutant Generation Example

The process of *mutant generation* is demonstrated using the `Stack` class example. In this example, two Java mutation operators [98], the Java `this` Deletion (JTD) and Java `this` Insertion (JTI) operators are applied to the source code of the `Stack` class.

The JTD operator is applicable to locations which contain the `this` keyword. When applied to these locations, the operator creates deviations by deleting the `this` keyword. The result of applying the JTD operator to the `Stack` class is illustrated in the comments

```

1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){
5         return this.size;
6     }
7     public void setSize(int size) {
8         if(size >= 0){//deviation this.size >= "this" is inserted
9             this.size=size;//deviation =this.size "this" is inserted
10            if(size>Stack.max){//deviation this.size> "this" is inserted
11                this.size=Stack.max;
12            }
13        } else {
14            throw new IllegalArgumentException();
15        }
16    }
17 }

```

Listing 4.2: `this` keyword insertion deviants

at lines 9 and 11 in Listing 4.1. Each of these locations contain a reference to the `this` keyword. In each deviation the keyword is deleted.

The JTI operator is applicable to locations which contain references to method parameters that have the same name as a method attribute. An example of such a location occurs at line 8 of Listing 4.2. At this line, the `size` parameter of the `setSize` method is referenced. There is also an attribute in the `Stack` with the name `size`. By default, a reference to `size`, is a reference to the parameter. To reference the `size` attribute can only be made by prefixing the reference with the `this` keyword.

When applied to these locations, the operator creates deviations by inserting the `this` keyword as a prefix to parameter references. The result of applying the JTI operator to the `Stack` class is illustrated in the comments at lines 8, 9 and 10 in Listing 4.2. At each of these locations, the operator creates deviations in which the `this` keyword is inserted.

The output of the *mutant generation* phase, after being applied to the `Stack` class example, is illustrated in Figure 4.8. This shows that five mutants are generated at four locations at lines 8, 9, 10 and 11 by the JTD and JTI operators.

Location Execution

The second phase of MA is *location execution*. The goal of this phase is to reduce the number of mutants that need to be executed by each test in the *fault exposure* phase.

In the *fault exposure* phase, tests are executed against mutants. The *fault exposure* phase can be computationally expensive [31, 116, 114]. This is because every test needs to be executed against every mutant. If there are a large number of tests and/or mutants, then the number of test-mutant executions is large. A large number of test-mutant executions requires a large amount of computational resources, which may be infeasible.

4.1. TESTABILITY MEASUREMENT

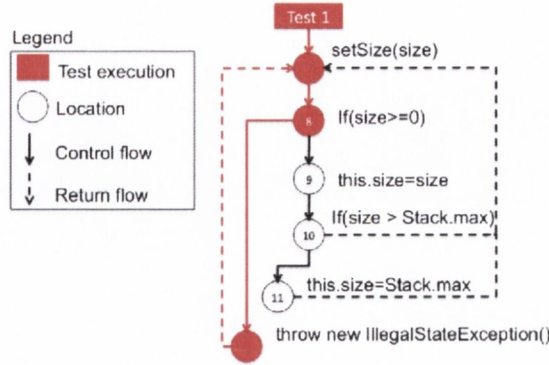


Figure 4.9: Locations Executed by *Test 1*

The need for a large amount of computational resources can be reduced by reducing the number of test-mutant executions in the *fault exposure* phase. The goal of the test-mutant executions in the *fault exposure* phase is to identify the outcome of each test-mutant execution. This reduction is made possible by identifying the mutants that contain faults at locations that are not executed by tests. Section 3.1 demonstrates that the execution of the location at which the fault occurs is the primary requirement for fault execution. If a test does not execute the location at which a fault occurs, then the test cannot expose the fault.

In this phase of MA, the locations executed by each test are identified by executing each test against the original version of the implementation. By identifying the locations executed by each test, the locations that are not executed are determined. The mutants at the locations that are not executed by a test, do not need to be executed to identify the outcome as it is known to be a pass. The output of this phase of MA is a reduction in the number of mutants that must be executed by each test. For each test, only the mutants that contain faults at locations that are executed by the test need to be executed.

Location Execution Example

The process of identifying the locations executed by each test is demonstrated using the `Stack` class example. This demonstration focuses on *Test 1* from Listing 3.2. In this phase, this test is executed against the original version of the `Stack` class to identify which of the locations, identified in Figure 4.8, are executed by the test.

In Figure 4.9, the path of execution exercised by *Test 1* through the `setSize` method of the `Stack` class is highlighted in red. This figure identifies the relevant locations¹ and shows that *Test 1* executes location 8. Because locations 9, 10 and 11 are *not* executed by *Test 1*, the outcome of executing *Test 1* against mutants that contain faults at these

¹These locations are identified in the *mutant generation phase*

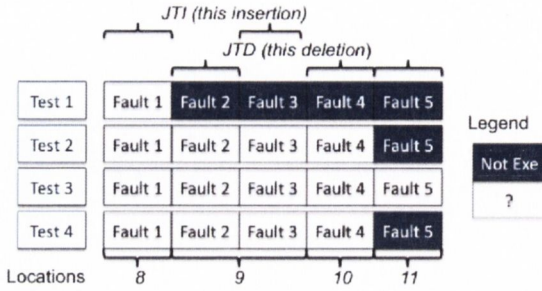


Figure 4.10: Faults Execution

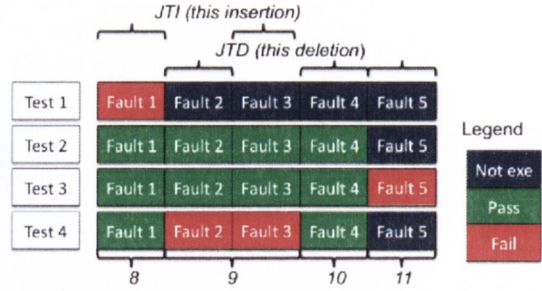


Figure 4.11: Fault Exposure

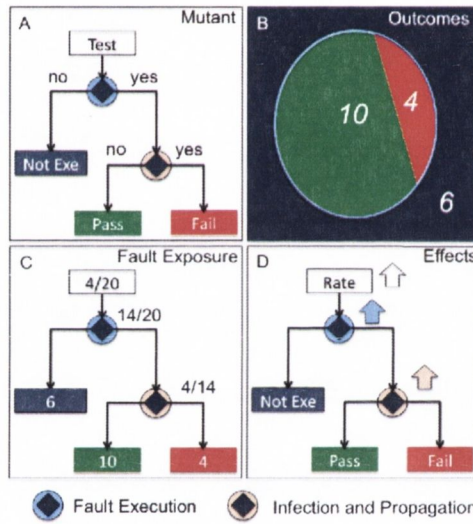


Figure 4.12: Fault Exposure: Model, Rates and Effects of Sub-Rates

locations is a *pass*, indicating that these faults are not exposed.

Based on this, the number of mutants that need to be executed in the *fault exposure* phase of MA is reduced from five to one. Figure 4.10 presents the output of this phase for the four tests presented in Listing 3.2. The figure identifies for each test, the mutants that contain faults at locations that will be exercised by the test.

Fault Exposure

The third and final phase of MA is *fault exposure*. In this phase, each test is executed against the mutants that contain faults, at locations that are exercised by the test. When the execution of a test against a mutant results in failure, **the fault is exposed**. When the execution of a test against a mutant results in a pass, **the fault is not exposed**.

This model of fault exposure is illustrated in Part A of Figure 4.12. This shows the causes of the observed outcomes. If a test does not execute the fault contained in the

4.1. TESTABILITY MEASUREMENT

mutant, then a *not exe* outcome results. If a test does execute the fault contained in the mutant, then the state directly after the execution may become infected and this infected state can be propagated into the output of the mutants execution. If infection and propagation occurs, then this results in a *fail* outcome. If not, then a *pass* outcome results.

Figure 4.11 illustrates the outcomes of executing each of the four tests, presented in Listing 3.2, against the mutants of the `Stack` class identified in Figure 4.10. These outcomes are the basis for calculating the rate of *fault exposure*. To calculate the rate, the number of *not exe*, *pass* and *fail* outcomes are counted. Based on these counts, the rate of fault exposure is calculated as $rate = \frac{fail}{fail+pass+notexe}$. The outcomes of test-mutant execution, illustrated in Figure 4.11 are summarised in Part B of Figure 4.12, which shows that there are 6 *not exe* outcomes for test-mutant execution. There are 10 *pass* and 4 *fail* outcomes. Based on these counts, the rate of fault exposure is calculated to be $\frac{4}{20} = \frac{4}{4+10+6}$, as illustrated in Part C of Figure 4.12.

Part C of Figure 4.12 indicates that the rate of *fault exposure* is caused by two sub-rates. The first is the rate of *fault execution*. As illustrated, in the figure this rate is calculated as $rate = \frac{fail+pass}{fail+pass+notexe}$ or $\frac{14}{20} = \frac{14}{4+10+6}$. The second is the rate of *infection and propagation*. As illustrated, this rate is calculated as $rate = \frac{fail}{fail+pass}$ or $\frac{4}{14} = \frac{4}{4+10}$.

The effects of these sub-rates on the rate of fault exposure is illustrated in Part D of Figure 4.12. The relationship between the rates of *Fault eXecution (FX)* and *Infection and Propagation (IP)* and the rate of *Fault Exposure (FE)* is multiplicative $FE = FX \times IP$. This is demonstrated by the rates derived from the `Stack` example, $\frac{4}{20} = \frac{14}{20} \times \frac{4}{14}$. The rate of *fault execution* represents the proportion of mutants that are executed. If this rate is high, then the number of potential *fail* outcomes is high. The rate of *infection and propagation* represents proportion of the executed mutants that result in a *fail* outcome. If this rate is high, then the number actual *fail* outcomes is high, which ensures a high rate fault exposure.

4.1.3 Summary

Figure 4.1 shows that in the first phase of the study, the testability of implementations are measured. In this section, mutation analysis was selected as the approach used to measure the testability of implementations. It is selected because it results in a measure that is more reflective of the ease of fault exposure than other approaches. The mutation analysis approach was also described in detail.

4.2 Gathering Measures of Testability

In the measurement phase of the study, illustrated in Figure 4.1, mutation analysis (MA) is applied to AOP and OOP implementations of different versions of a program. The goal of this phase is to gather testability measures from these implementations that can be analysed to identify the comparative effect of AOP and OOP on testability over maintenance activities, as illustrated in Figure 4.4.

To perform this analysis requires pairs of AOP and OOP measures, one pair for each version of the program. Each measure must represent the combined effects of two factors: implementation and maintenance. Section 2.10, identifies an approach used in studies, similar to this study, that compare the effects of AOP and OOP on the other indicators of maintainability that result in measures that meet these requirements.

One challenge to overcome in following this approach was the integration of mutation analysis. Typically, when this approach is applied result in measures that are solely representative of the combined effects of two factors: implementation and maintenance. The measure of testability, derived using mutation analysis, introduces two other factors, tests and mutants. To overcome this, an extension is introduced, as outlined in Section 3.2.4, to fix these factors over measurements. Fixing these factors for each measurement, ensures that the resulting measures are solely representative of the combined effects of two factors: implementation and maintenance.

4.2.1 Following the Common Approach

Section 2.10, outlines an approach commonly used in similar studies, that compare the effects of AOP and OOP on the other indicators of maintainability, that result in measures that are solely representative of the combined effects of two factors: implementation and maintenance. The foundation of this approach is **implementation and maintenance equivalence**, which are explained and illustrated through the `Stack` class example.

Implementation and Maintenance Equivalence

In the common approach, illustrated in Figure 4.13, maintenance activities are cumulatively applied to AOP and OOP implementations of a program. The initial AOP and OOP implementations of a program are equivalent in that they differ only in the approach used for their development. Equivalence is assured by fixing all other factors that can cause the implementations to differ. For example, the implementations are developed to the same level of expertise, are based on the same style of programming, satisfy the same requirements, expose the same interface and produce the same outputs for a given input etc.

The same maintenance activities are cumulatively applied to both AOP and OOP

4.2. GATHERING MEASURES OF TESTABILITY

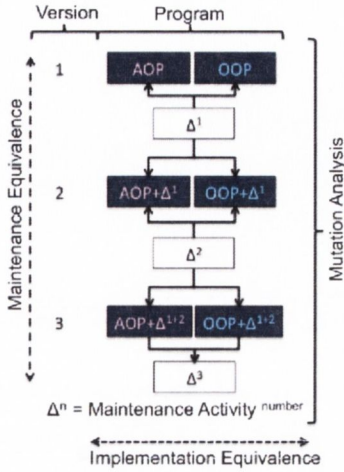


Figure 4.13: Approach

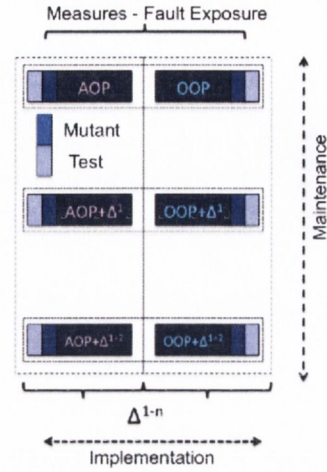


Figure 4.14: Factors

implementations. After each maintenance activity is applied to both implementations, a new version of these implementations result. The new versions of these implementations are equivalent because the maintenance activity is applied to both implementations. This means that the respective versions of the AOP and OOP implementations are also equivalent. When all maintenance activities are applied, the only difference between each respective version of the AOP and OOP implementations is a maintenance activity.

Stack Example

To demonstrate equivalence, Java and AspectJ implementations of the `Stack` class are illustrated in Listings 4.3 and 4.4. Both implementations are developed by a programmer proficient in both Java and AspectJ, are based on the same style of programming, satisfy the same requirements, expose the same interface and produce the same outputs for a given input. Specifically, both implement a `setSize` in which the pre- and post-conditions are defined that identify minimum and maximum sizes for the `Stack`. The only difference between these implementations is that Listing 4.3 is implemented in Java and Listing 4.4 is implemented in AspectJ. They are equivalent in that they differ only in the approach used in their development.

As illustrated in Figure 4.15, one maintenance activity is then applied to both implementations of the `Stack` class. In this maintenance activity, the post-condition that identifies a maximum and size for the `Stack` is removed. The results of applying this maintenance activity are illustrated in Listings 4.5 and 4.6. The only difference between Listings 4.3 and 4.4 and Listings 4.5 and 4.6 is that the post-condition is removed. This means that the Java implementations for the `Stack` in Listings 4.3 and 4.4 are maintenance equivalent, in that the only difference between these implementations is the effect

```

1 public class Stack {
2   int size = 0;
3   static final int max = 5;
4   public int getSize(){...}
5   public void setSize(int size) {
6     if(size >= 0){
7       this.size=size;
8       if(size>Stack.max){
9         this.size=Stack.max;
10      }
11     } else {
12       throw new IllegalArgumentException();
13     }
14   }
15 }

```

Listing 4.3: Java Stack - Initial Implementation

```

1 public class Stack {
2   int size = 0;
3   static final int max = 5;
4   public int getSize(){...}
5   public void setSize(int size) {
6     this.size=size;
7   }
8 }
9 public privileged aspect StackAspectPrePost {
10 void around(int size, impl2.Stack stack) :
11   execution(void impl2.Stack.setSize(int))
12   && args(size) && target(stack){
13   if(size >= 0){
14     proceed(size, stack);
15     if(size > Stack.max){
16       stack.size = Stack.max;
17     }
18   } else {
19     throw new IllegalArgumentException();
20   }
21 }
22 }

```

Listing 4.4: AspectJ Stack - Initial Implementation

```

1 public class Stack {
2   int size = 0;
3   public int getSize(){...}
4   public void setSize(int size) {
5     if(size >= 0){
6       this.size=size;
7     } else {
8       throw new IllegalArgumentException();
9     }
10  }
11 }

```

Listing 4.5: Java Stack - Maximum Condition Removed

4.2. GATHERING MEASURES OF TESTABILITY

```

1 public class Stack {
2     int size = 0;
3     public int getSize(){...}
4     public void setSize(int size) {
5         this.size=size;
6     }
7 }
8 public privileged aspect StackAspectPre {
9     void around(int size, impl2.Stack stack) :
10        execution(void impl2.Stack.setSize(int))
11        && args(size) && target(stack){
12        if(size >= 0){
13            proceed(size, stack);
14        } else {
15            throw new IllegalArgumentException();
16        }
17    }
18 }

```

Listing 4.6: AspectJ Stack - Maximum Condition Removed

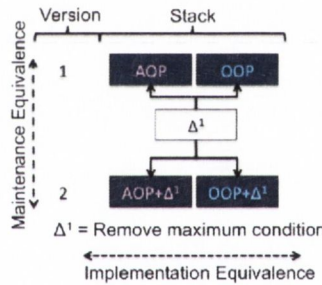


Figure 4.15: Stack Example

of applying the maintenance activity.

4.2.2 Integrating Mutation Analysis into the Approach

The measurement approaches used when this approach is applied result in measures that are solely representative of the combined effects of two factors: implementation and maintenance. An example of a typical measurement approach is to measure the size of each implementation. As illustrated in Figure 6.5, this measurement is based solely on the implementation from which it is taken. When this measurement approach is applied to implementations that are equivalent, the resulting measures are solely representative of the effects of the implementation and maintenance factors.

Figure 4.14 illustrates the result of applying MA to the implementations. There are measures for AOP and OOP implementations of each version of the program. Each measure is representative of the combined effects of four factors: test, mutant, implementation

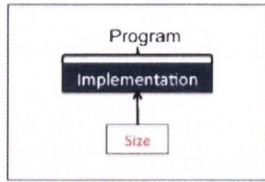


Figure 4.16: Size

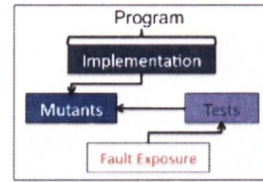


Figure 4.17: Mutation Analysis

and maintenance. As illustrated in Figure 4.17, the application of MA to each implementation requires the generation of mutants of the implementation and the execution of tests against those mutants, the outcomes of which are used to derive a measure of testability. The test and mutant factors are an issue because their uncontrolled presence introduces the effects of these factors into the measures and has the potential to confound the comparison of the effects of AOP and OOP on testability.

To address this problem, and ensure that each measure is representative of the combined effects of only implementation and maintenance, the effects of test and mutant factors must be neutralised. Section 3.2.4 outlines an approach to neutralise the effects of these factors, which is to fix the test and mutant factors for each application of mutation analysis.

Fixing the Mutant Factor

The mutant factor is fixed by ensuring that the mutants generated for each implementation are (implementation and maintenance) equivalent. Figure 4.18 illustrates the mutants generation phase of MA. In this phase, mutants are generated for the AOP and OOP implementations of each version of the program.

The types of faults that are generated in the mutants of the AOP and OOP implementations have to differ, as they are based on the types of faults that are observed in practice. Different types of faults occur in AOP and OOP implementations, and as such the types of faults generated in mutants of these implementation also differ [53, 26, 36, 156].

To fix the fault type factor, this study ensures that the mutants generated for each pair of equivalent AOP and OOP implementations are also equivalent. That is, the only difference between AOP and OOP mutants is based directly on the difference between the AOP and OOP implementations from which they are generated.

To ensure that the fault type factor is fixed, the same process is used to generate mutants in both implementations. Although mutants that contain different types of faults are generated for the AOP and OOP implementations, the resulting mutants are equivalent and the mutant factor is fixed.

4.2. GATHERING MEASURES OF TESTABILITY

Fixing the Test Factor

Figure 4.19 illustrates the application of the *location execution* phase to the AOP and OOP implementations. In this phase, tests are executed against the implementation that to identify the locations that are executed by each test. To fix the test factor in this phase, the same set of tests are executed against the AOP and OOP implementations of each version of the program. This ensures that the only difference in the results of this phase is the difference between the AOP and OOP implementations.

Figure 4.20 illustrates the application of the *fault exposure* phase. In this phase, tests are executed against mutants. To fix the test factor in this phase, the same set of tests are executed against the mutants of the AOP and OOP implementations of each version of the program. This ensures that the only difference between outcomes for each set of mutants is based on the difference between the AOP and OOP implementations.

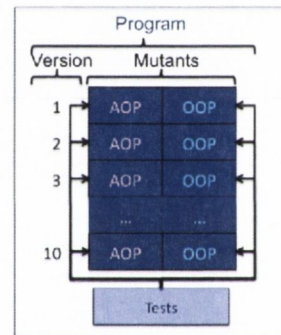
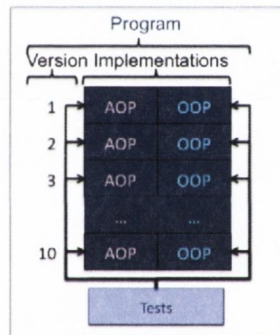
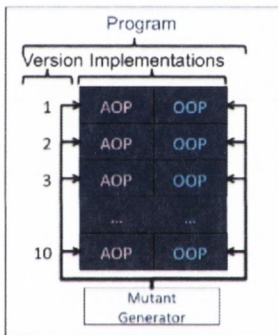


Figure 4.18: Mutant Gen Figure 4.19: Location Exe Figure 4.20: Exposure

Stack Example

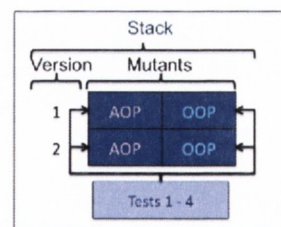
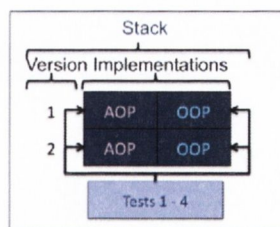
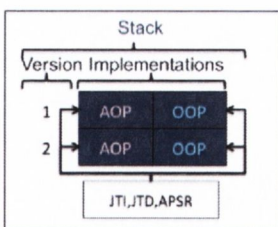


Figure 4.21: Mutant Gen Figure 4.22: Location Exe Figure 4.23: Exposure

The integration of the measurement gathering and mutation analysis approaches used in this study is demonstrated using the **Stack** class example. In this example, mutation analysis is applied to the implementation and maintenance equivalent Java and AspectJ implementations of the **Stack** class, illustrated in Listings 4.3, 4.4, 4.5 and 4.6.

Mutant Generation for Stack Implementations

Figure 4.21 illustrates application of a mutant generator to the Java and AspectJ implementations of versions 1 and 2 of the `Stack` class. The mutant generator applies different mutation operators to each implementation. The AspectJ mutation operators are applied to locations that contain AspectJ features and Java mutation operators are applied to locations that contain Java features. In this example, two Java mutation operators, the Java `this` Deletion (JTD) and Java `this` Insertion (JTI) operators are applied to the AspectJ and Java versions of the `Stack` class. The Around Proceed Statement Removal (APSR) operator is also applied to the AspectJ implementation. These operators represent a subset of the types of faults observed to occur in Java and AspectJ implementations [98, 53]. The complete set of operators are detailed in Section 5.2 of the next chapter.

The JTD operator is applicable to locations that contain the `this` keyword. When applied to such a location, the operator creates a deviation or fault by deleting the `this` keyword. The JTI operator is applicable to locations that contain references to method parameters that have the same name as a method attribute. When applied to such a location, the operator creates a fault by inserting the `this` keyword. The APSR operator is applicable to locations that contain a call to `proceed`. When applied to such a location, the operator creates a fault by deleting the `proceed` call.

In this application of mutant generation, the generator analyses the code to identify all of the locations that meet these criteria. The mutation operators are then applied to the appropriate locations. The results of this process are illustrated in Listings 4.7, 4.8, 4.9 and 4.10. In each listing, the comments identify faults generated in mutants and the mutation operators that created them.

The only difference in each set of generated mutants is based solely on the implementation for which they are generated. For example, Listings 4.7 and 4.8 contain equivalent Java and AspectJ implementations of the first version of the `Stack` class. The mutants generated for the Java and AspectJ implementations differ. Five mutants of the Java implementation are generated by the JTI and JTD operators. Three mutants of the AspectJ implementation are generated by the JTI, JTD and APSR operators. They are implementation equivalent because they represent the faults that can occur in both implementations.

Another example of mutant equivalence is observed in Listings 4.7 and 4.9. These contain the Java implementations of the first and second versions of the `Stack` class. These are equivalent in that the only difference between these implementations is the maintenance activity applied to the first Java implementation to create the second. Different mutants are generated for each implementation. In the implementation of the first version, there are five faults and in the second there are three. They are maintenance equivalent because the only difference between these mutants is directly based on the

4.2. GATHERING MEASURES OF TESTABILITY

```
1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){
5         return this.size;
6     }
7     public void setSize(int size) {
8         if(size >= 0){// if(this.size >= 0) fault 1-JTI
9             this.size=size;// size=size faults 2-JDT 3-JTI
10            if(size>Stack.max){// this.size> fault 4-JTI
11                this.size=Stack.max;// size= fault 5-JDT
12            }
13        } else {
14            throw new IllegalArgumentException();
15        }
16    }
17 }
```

Listing 4.7: Java Stack - Initial Implementation Mutants

```
1 public class Stack {
2     int size = 0;
3     static final int max = 5;
4     public int getSize(){...}
5     public void setSize(int size) {
6         this.size=size;// size=size fault 2-JTD 3-JTI
7     }
8 }
9 public privileged aspect StackAspectPrePost {
10 void around(int size, impl2.Stack stack) :
11     execution(void impl2.Stack.setSize(int))
12     && args(size) && target(stack){
13     if(size >= 0){
14         proceed(size, stack);// fault 6-APSR
15         if(size > Stack.max){
16             stack.size = Stack.max;
17         }
18     } else {
19         throw new IllegalArgumentException();
20     }
21 }
22 }
```

Listing 4.8: AspectJ Stack - Initial Implementation Mutants

difference between the Java implementations.

By ensuring that each set of mutants are implementation and maintenance equivalent, the mutant factor is fixed. The generated mutants do not impose any influence on the measures derived from executing tests against these to produce outcomes, from which fault exposure rate measures can be derived.

```

1 public class Stack {
2     int size = 0;
3     public int getSize(){...}
4     public void setSize(int size) {
5         if(size >= 0){ // if(this.size >= 0) fault 1-JTI
6             this.size=size;
7         } else {
8             throw new IllegalArgumentException();
9         }
10    }
11 }

```

Listing 4.9: Java Stack - Maximum Condition Removed Mutants

```

1 public class Stack {
2     int size = 0;
3     public int getSize(){...}
4     public void setSize(int size) {
5         this.size=size;// size=size fault 2-JTD 3-JTI
6     }
7 }
8 public privileged aspect StackAspectPrePost {
9     void around(int size, impl2.Stack stack) :
10    execution(void impl2.Stack.setSize(int))
11    && args(size) && target(stack){
12        if(size >= 0){
13            proceed(size, stack);// fault 6-APSR
14        } else {
15            throw new IllegalArgumentException();
16        }
17    }
18 }

```

Listing 4.10: AspectJ Stack - Maximum Condition Removed Mutants

Location Execution and Fault Exposure for Stack Implementations

Figures 4.22 and 4.23 illustrate the *location execution* and *fault exposure* phases of MA. Both of these phases use the four tests developed for the Stack, illustrated in Listing 3.2. In the *location execution* phase, they are executed against each implementation to identify the locations executed by each test. In the *fault exposure* phase, they are executed against each mutants of executed locations in each implementation. In both phases, the test factor is fixed. The only difference in the outcomes for each implementation, are due solely to the difference between each implementation.

The outcomes of both phases for the Java and AspectJ implementations of versions 1 and 2 of the Stack are illustrated in Figures 4.24 and 4.25 respectively. These outcomes are summarised in Table 4.2. For each implementation of both versions, this table counts the number of mutants that were *not* executed, mutants that were executed resulting in a *pass* and mutants that were executed resulting in a *fail*. Based on these counts, the

4.2. GATHERING MEASURES OF TESTABILITY

Version	Java				AspectJ			
	<i>not exe</i>	<i>pass</i>	<i>fail</i>	<i>rate</i>	<i>not exe</i>	<i>pass</i>	<i>fail</i>	<i>rate</i>
1	6	10	4	0.20	3	4	3	0.30
2	2	4	3	0.33	3	3	3	0.33

Table 4.2: Mutation Analysis Outcomes

rate of exposure is calculated as $rate = \frac{fail}{fail+pass+notexe}$.

Each measure represents the combined effects of two factors: implementation and maintenance. The rate of *fault exposure* for the Java implementation at version 1 is 0.20. This represents the effects of Java at the initial maintenance version. The rate for the Java implementation at version 2 is 0.33. This represents the effects of Java at the second maintenance version. The rate of *fault exposure* for the AspectJ implementation at version 1 is 0.30. This represents the effects of AspectJ at the initial maintenance version. The rate for the AspectJ implementation at version 2 is 0.33. This represents the effects of AspectJ at the second maintenance version.

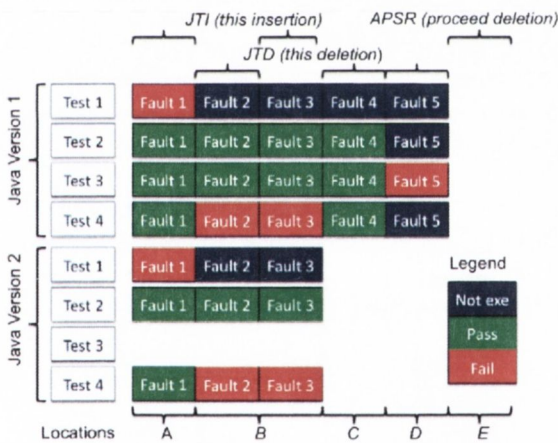


Figure 4.24: Java

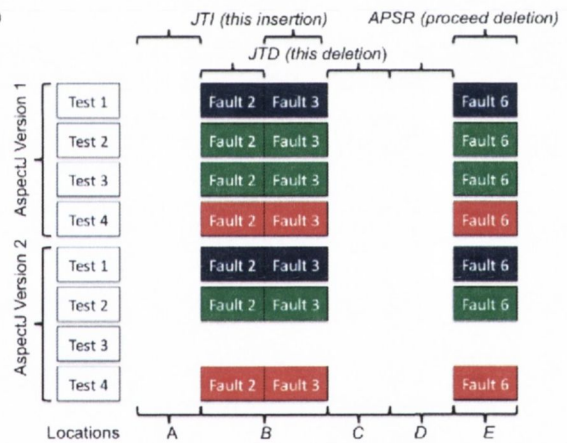


Figure 4.25: AspectJ

4.2.3 Summary

In the measurement phase of this study, mutation analysis (MA) is applied to AOP and OOP implementations of different versions of a program. This section has outlined an approach for gathering testability measures from implementations that can be analysed to identify the comparative effect of AOP and OOP on testability over maintenance activities. The approach ensures that the gathered measures represent the combined effects of only two factors: implementation and maintenance. Each measure gathered, represents the effects of either AOP or OOP at each maintenance version of the program. In the next section, the approach to analyse these measures is outlined.

Version	Java			AspectJ		
	<i>not exe</i>	<i>pass</i>	<i>fail</i>	<i>not exe</i>	<i>pass</i>	<i>fail</i>
1	6	10	4	3	4	3
2	2	4	3	3	3	3
3	4	16	10	7	13	3
4	6	18	8	8	12	4
5	6	20	14	10	12	3
6	8	22	16	14	16	6
7	10	22	18	16	14	6
8	12	26	18	18	18	5
9	16	26	20	20	22	10
10	10	20	16	14	16	8

Table 4.3: Mutation Analysis Outcomes

4.3 Analysing Measures of Testability

In the measurement phase of the study, measures are gathered that represent the combined effects of two factors: implementation and maintenance. Each measure represents the effects of either AOP or OOP at each maintenance version of the program. In this section, the approach to analysing these measures is outlined. This section shows how a mixture of informal graphical comparison and formal regression analysis approaches are used to compare the effects of AOP and OOP on the rate of fault exposure and understand the causes for differences in those effects. The first part of this section, shows how measures are graphed to illustrate the difference between the effects of AOP and OOP on rates and the second part shows how regression analysis is used to quantify the difference between these effects on rates.

4.3.1 Graphical Analysis

Graphical analysis [149, 14, 92, 97, 54, 87, 63] is a typical approach to analysing the measures collected in comparative studies of the effects of OOP and AOP on indicators of maintainability. Graphical analysis is widely used because it is an intuitive way to understand and compare measures.

Outputs of Mutation Analysis for Stack example

To demonstrate this it is applied to an extension of the `Stack` example used in Section 4.2.2 that are presented in Table 4.3. In this extension, mutation analysis is applied to a further eight versions of the Java and AspectJ implementations, illustrated in Listings 4.7 and 4.9. Table 4.3 presents the resulting counts of *not exe*, *pass* and *fail* outcomes for the Java and AspectJ implementations at a specific maintenance version of the program.

4.3. ANALYSING MEASURES OF TESTABILITY

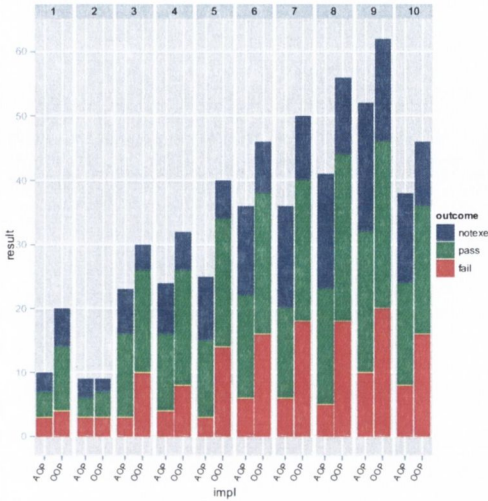


Figure 4.26: Visualising Outcomes

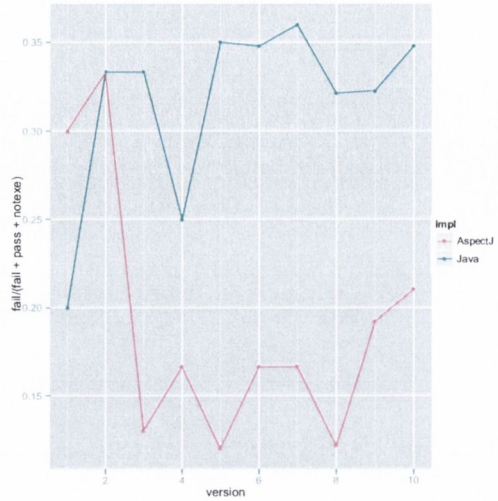


Figure 4.27: Fault Exposure

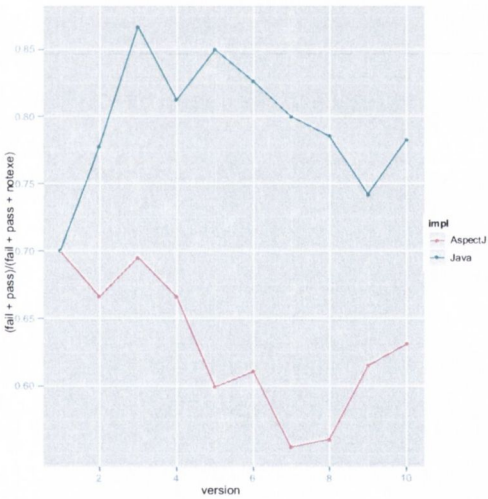


Figure 4.28: Fault Execution

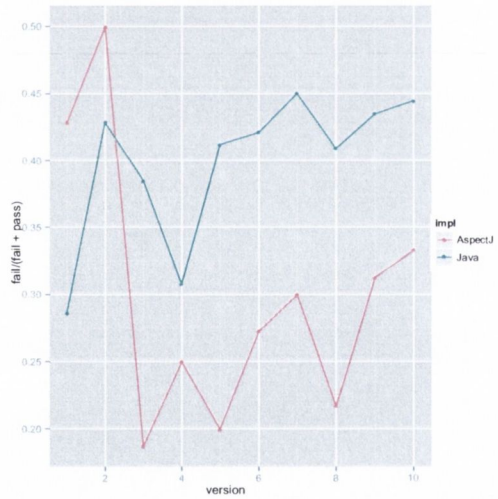


Figure 4.29: Infection and Propagation

Visualising Results

Figure 4.26 presents a bar chart that visualises number of *not exe*, *pass* and *fail* outcomes for each implementation in Table 4.3. Each bar in this chart represents the outcomes of the test-mutant executions for a specific implementation. The bars for the AspectJ and Java implementations of each version are placed directly beside one another to make comparing the number of *not exe*, *pass* and *fail* outcomes easier. This chart provides an easily comparable representation of the numbers of *not exe*, *pass* and *fail* outcomes for each pair of AspectJ and Java implementations.

For instance, from this chart it is clear that there are consistently less test-mutant executions for AspectJ compared to Java. This is because there are less mutants generated for the AspectJ implementations, as indicated in Figures 4.24 and 4.25. A trend of an increasing number of test-mutant executions up to version 9 of the stack can also be seen. From this graph, it is however more difficult to estimation of the rates of *fault exposure*, *fault execution* and *infection and propagation*.

This chart also enables the rates of *fault exposure*, *fault execution* and *infection and propagation* over versions to roughly estimated and compared. For example, at version 9, the rate of *fault exposure* ($rate = \frac{fail}{fail+pass+notexe}$) seems to be higher for AspectJ. The rate of *fault execution* ($rate = \frac{fail+pass}{fail+pass+notexe}$) seems to be higher and the rate of *infection and propagation* ($rate = \frac{fail}{fail+pass}$) also seems to be higher.

Analysing Rates

Figures 4.27, 4.28 and 4.29 are graphs that illustrate the rates of *fault exposure*, *fault execution* and *infection and propagation*, respectively. In these graphs the x-axis represents the version and the y-axis represents the rate. Each point on the graph represents a rate for each an AspectJ or Java implementations of each version of the HW. The points are differentiated by colour and a line connecting the points for AspectJ and Java implementations is provided to highlight the changes in the rate over versions of the HW for each.

Fault Execution

Figures 4.28 and 4.29 show the rates of *fault execution* and *infection and propagation* for each AspectJ or Java implementation. Figure 4.28 shows that rate of *fault execution* is consistently higher for Java implementations. The rate starts at 0.7 and ends at 0.63 for AspectJ and starts at 0.7 and ends at 0.79 for Java. This means that when the same tests are executed against Java and AspectJ **Stack** implementations there were more faults executed in the Java implementation.

Infection and Propagation

Figure 4.29 shows that rate of *infection and propagation* is higher for Java implementations. The rate starts at 0.43 and ends at 0.33 for AspectJ and starts at 0.29 and ends at 0.44 for Java. Besides versions 1 and 2 the rate of *infection and propagation* for Java are consistently higher compared to AspectJ. This means that when the same tests are executed against the faults in mutants of the Java and AspectJ **Stack** implementations, more faults are exposed in the Java implementation.

4.3. ANALYSING MEASURES OF TESTABILITY

Fault Exposure

As demonstrated in Section 4.1.2, the relationship between the rates of *fault execution* and *infection and propagation* and the rates of *fault exposure* is multiplicative. That is the rate of *Fault Exposure (FE)* is the product of the rates of *Fault eXecution (FX)* and *Infection and Propagation (IP)*, $FE = FX \times IP$. Figure 4.27 shows that the rates of *fault exposure* for AspectJ and Java implementations differ. Each of these rates is based on the product of the rates, illustrated in Figures 4.28 and 4.29.

Figure 4.27 shows that the rate of *fault exposure* for the AspectJ implementation at version 1 is $0.3 = 0.7 \times 0.43$ and decreases to $0.21 = 0.63 \times 0.33$ at version 10. The rate of *fault exposure* for Java implementation at version 1 is $0.3 = 0.7 \times 0.29$ and increases to $0.35 = 0.79 \times 0.44$ at version 10. The calculation of these rates shows that higher rates show that higher rates of *fault execution* and *infection and propagation* result in higher rates of *fault exposure*.

In Figure 4.27 the rate *fault exposure* is consistently higher for Java from version 3 to 10. This is because the rate *fault execution* and *infection and propagation* are consistently higher for Java over these versions. This means that in the Java implementations of versions 3 to 10, more faults are executed by tests and when these faults are executed more of them are exposed.

The figure shows that the rate at version 1 is higher for AspectJ and at version 2 the rate is the same for Java and AspectJ. The rate at version 1 is higher because the rate of *fault execution* is the same for both AspectJ and Java at version 1 but the rate of *infection and propagation* is higher for AspectJ. This means that in the AspectJ and Java implementations of versions 1 the same proportion of faults are executed by tests but more these faults are exposed when executed in the AspectJ implementation.

The rate at version 2 is equal because rate of *fault execution* is higher for the Java implementation but the rate of *infection and propagation* is higher for AspectJ. This means that although there are more faults executed in the Java implementation of versions 2, less of these faults are exposed when executed compared to the AspectJ implementation. These differences cancel one another out and result in an equal rate of *fault exposure* for the AspectJ and Java implementations of version 2.

Conclusions from Graphical Analysis

From the analysis of the bar chart in Figure 4.26, it can be concluded that there are more test-mutants executions for the Java implementations compared to the AspectJ implementations. The reason for this is that there are more mutants generated for the AspectJ implementation. The same number of tests are executed against the mutants of each implementation, which means that the only cause of the difference is due to the number of mutants generated.

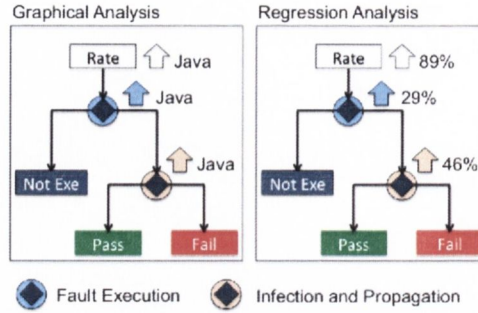


Figure 4.30: Conclusions of Analysis for Stack

The left hand side of Figure 4.30 illustrates the conclusions drawn from the analysis of the rates of *fault execution*, *infection and propagation* and *fault exposure* presented in Figures 4.27, 4.28 and 4.29. It indicates that the rate of fault exposure or testability is higher for the Java implementations of the **Stack**. It also illustrates that the reason for the observed difference is that over the different versions of the **Stack**, there are more faults executed by tests and more of these executed faults result in infection and propagation resulting in test failure and fault exposure for the Java implementations.

Graphical analysis, as has been demonstrated, allows conclusions to be drawn about the comparative effect, or differences in the effects, of AspectJ and Java on the testability of the stack. It also allows conclusions to be drawn about the comparative effect of AspectJ and Java on the determinants testability, the rates of *fault execution* and *infection and propagation*. As will be detailed next, binomial regression analysis [52], is used to quantify the comparative effects of AspectJ and Java on the testability of the stack and its determinants.

4.3.2 Binomial Regression Analysis

Binomial Regression Analysis (BRA) [52] is used to quantify the difference in the effects of OOP and AOP on the rates of *fault exposure*, *fault execution* and *infection and propagation*. BRA is a statistical technique for analysing the relationship between a binomial response and explanatory factors. In this application of binomial regression, the binomial response is the rate and the explanatory factors are the implementation and maintenance version. The relationship between these factors and the response is defined in a model. This relationship is captured in the regression model, $rate \sim implementation + version$, which indicates that each rate is caused by both the implementation and version of the program. This is the standard way in which a binomial regression model is specified [52]. In this case, there are three models specified, one for each rate of *fault exposure*, *fault execution* and *infection and propagation*.

4.3. ANALYSING MEASURES OF TESTABILITY

model	specification
1	<i>Fault Exposure</i> \sim <i>implementation</i> + <i>version</i>
2	<i>Fault eXecution</i> \sim <i>implementation</i> + <i>version</i>
3	<i>Infection and Propagation</i> \sim <i>implementation</i> + <i>version</i>

Table 4.4: Models

Model Fitting

The relationship between each rate and the factors in each model is measured by fitting the model over the measures in Table 4.3. In the model fitting process, the correlation between the effects of AspectJ and Java implementation approaches and each version on the observed rate is measured [52]. These correlations are used to measure the generalised effects of each factor on the rate of fault exposure [52].

Table 4.31 presents the results of fitting three models for the effects of the *Version* and *Implementation* factors on the rates of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. These models are presented in Table 4.4.

The first two columns of the table are *Version* and *Implementation*. The rows within these columns represent the specific versions (1-10) and implementations (AspectJ or Java) for to which the measures in Table 4.3 relate. For each model, the model fitting process analyses the strength of the correlation between each version and implementation approach on the rate. The resulting correlation measures for each of the three models are presented in rows in the columns labelled *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. Each measure represents the strength of the correlation between the observed rate and a specific version (1-10) and implementations (AspectJ or Java).

Comparative Effects

As detailed by Faraway [52], the measures of the effects presented in Table 4.31 are used to construct the graphs of the generalised effects of AspectJ and Java on the rates of *fault exposure*, *fault execution* and *infection and propagation*, presented in Figures 4.32, 4.33 and 4.34, respectively. In each of these figures, the difference between the Java and AspectJ lines is the measure of the comparative effect of AspectJ and Java. The measures of the difference between the AspectJ and Java lines is marked in red in Table 4.31. These are the measures of the comparative effect of AspectJ and Java on each rate. These are on the log odds scale [52] and need to be transformed by taking the exponent of each measure [52]. This results in a measure of the difference in the odds of AspectJ and Java exposing faults, executing faults and causing infection and propagation.

Version	Implementation	Fault Exposure (FE)	Fault eXecution (FX)	Infection and Propagation (IP)
1	AspectJ	-1.89630	-0.529032	-1.352227
	Java	0.63624	0.256460	0.381415
2		0.45421	0.069907	0.371564
3		0.07626	0.144253	-0.078863
4		-0.04978	0.088373	-0.150122
5		0.10965	0.078850	-0.009129
6		0.17326	0.063659	0.090059
7		0.19977	0.009482	0.165758
8		0.03550	0.001137	0.012562
9		0.17010	0.003726	0.155845
10		0.25076	0.045131	0.192354

Figure 4.31: Correlations between each Rates and versions and implementation factors

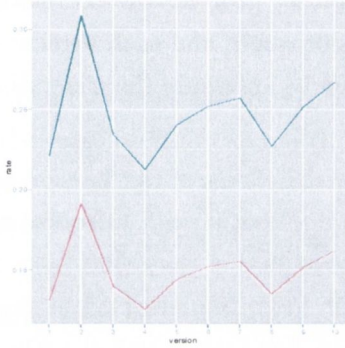


Figure 4.32: FE

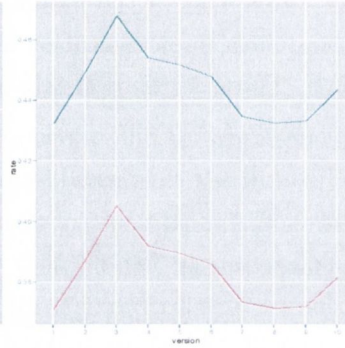


Figure 4.33: FX

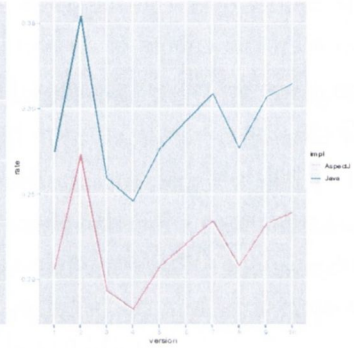


Figure 4.34: IP

Conclusions from Regression Analysis

The left hand side of Figure 4.30 illustrates the results of these transformations. It shows that, based on the results of binomial regression, that the odds of *fault exposure* are 89% ($1.889364 = \exp(0.63624)$) higher for Java. It shows that the odds of *fault execution* are 29% ($1.292347 = \exp(0.256460)$) higher for Java and that odds of *infection and propagation* are 46% ($1.464355 = \exp(0.381415)$) higher for Java.

This is explained further through the illustration in Figure 4.35. The boxes marked AspectJ and Java represent the total number of test-mutant executions for AspectJ and

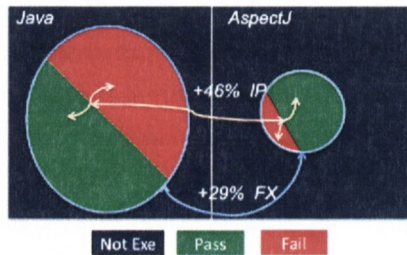


Figure 4.35: Causation for Comparative Effects

4.4. CHAPTER SUMMARY

Java implementations, respectively. The circles in these boxes represent the number of faults executed by tests in AspectJ and Java mutants. This representation shows that there are more faults executed in Java compared to AspectJ mutants. This difference is the cause of the 29% higher odds of *fault execution* for Java. The number of executed faults that result in pass and fails are represented inside the circle. This representation shows that there are proportionally less fails for AspectJ, indicating that less of the faults executed in AspectJ mutants result in infection and propagation. This difference is the cause of the 46% higher odds of *infection and propagation* for Java.

4.3.3 Threats to Analysis Validity

4.3.4 Summary

In the analysis phase of this study, the measures gathered from applying mutation analysis to AOP and OOP implementations of different versions of a program are analysed. This section has shown how a mixture of graphical and binomial regression analysis are used to analyse these measures and quantify the comparative effect of AOP and OOP on testability.

4.4 Chapter Summary

This chapter described the methodology followed in the measurement and analysis phase of the study. The first section selected and described mutation analysis as the approach used to measure testability. The second section described how this approach is applied to ensure the resulting pairs of measures isolate the combined effects the implementation and maintenance factors. The third section described how the effects of AOP and OOP are generalised and the comparative effect on testability is measured and how causation for this comparative is determined.

The methodology describes how the study will be carried out and outlines the approaches that will be used in the measurement and analysis phases of the study. The next chapter describes the inputs selected to fit into this methodology and as a basis for the study.

Chapter 5

Study Inputs

The previous chapter presents the methodology on which this study is based. The methodology describes how measures of testability are gathered and analysed to produce evidence of the comparative effect of AOP and OOP on testability.

The quality of this evidence is based on the implementations, mutants and tests used to gather each measure of testability. For evidence to be of high quality, the evidence must be generalisable [79]. Generalisability is the ability to draw general conclusions from evidence gathered in a specific context. General conclusions can be drawn from evidence gathered in contexts that are representative of the general case [79]. The first goal of this chapter is to show that the implementations, mutants and tests that form the context in which measures of testability are gathered, are selected because they are representative of the general case.

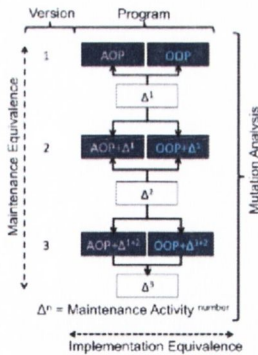


Figure 5.1: Program

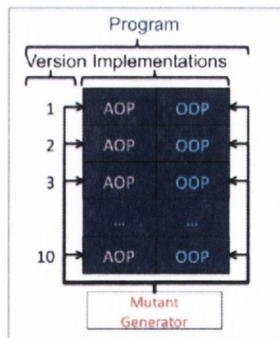


Figure 5.2: Mutants

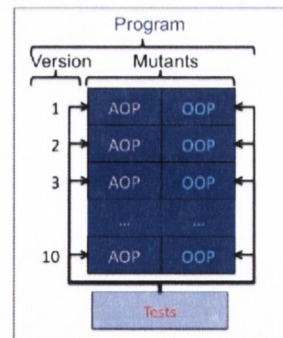


Figure 5.3: Tests

The second goal of this chapter is to show that the selected implementations, mutants and tests fit into the measurement gathering approach prescribed by the methodology. As explained in Section 2.10, the AOP and OOP implementations from which measures of testability are gathered must be implementation and maintenance equivalent. As illus-

trated in Figure 5.1, equivalence is assured by cumulatively applying the same maintenance activities to AOP and OOP implementations of a program that differ only in the approach used in their development. Figures 5.2 and 5.3 show that to measure the testability of each implementation using mutation analysis, mutants of each implementation must be generated and a set of tests must be executed against each implementation and mutants of each implementation.

The first section of this chapter describes the programs for which AOP and OOP implementations exist and the maintenance activities that have been cumulatively applied to these implementations to create several maintenance versions of both implementations. It describes the selection of a program which is representative of the general case and describes the selected program, maintenance activities and implementations in detail.

The second section of this chapter identifies the approach used to generate the mutants for each AOP and OOP implementation. It shows that the mutant generation approach is based on AOP and OOP fault models that are representative of the types of faults that are generally observed to occur in practice. The fault model is the basis for mutation operators that are used to generate mutants. The mutation operators defined for the AOP and OOP fault models are identified and described. The tool that implements these operators, and that is used to generate the mutants for each AOP and OOP implementation is also described.

The third section of this chapter identifies the approach used to select and automate tests that are executed against each implementation and mutants of each implementation in the study. The section outlines the different available test selection approaches and justifies the selected approach. It also describes the approach used to automate the execution of tests against implementations and mutants.

This chapter is concluded by summarising the selections made. This summary comments on how these selections impact on the degree to which general conclusions can be drawn from evidence gathered in the study.

5.1 Implementations

The section lists a set of candidate programs that have the potential to be used in the measurement gathering approach prescribed by the methodology. These are programs for which AOP and OOP implementations exist and maintenance activities have been cumulatively applied to these implementations to create several maintenance versions of both implementations. This section describes the approach used to select of the most representative implementations and describes the program and Maintenance Activities (MAT) on which they are based.

5.1. IMPLEMENTATIONS

program	studies	deployed	maintenance activities		
			adaptive	perfective	corrective
Health Watcher	[87, 63, 55]	yes	1	7	1
Online Shop	[14]	no	1	-	-
Library	[149]	no	2	1	3
Email Server	[92]	yes	2	1	-
Best Lap	[54]	yes	5	-	-
Mobile Media	[54]	no	8	-	-

Table 5.1: Candidates for Selection

5.1.1 Selection of Implementations

Table 5.1 lists the candidate programs that have the potential to be used in the measurement gathering approach prescribed by the methodology. This section describes the selection of the one that is most representative of the general case.

Selection Criteria

For each program, Table 5.1 identifies the *studies* which are based on the AOP and OOP implementations of the program. It identifies whether the program has been *deployed* and shows the distribution of *maintenance activities* applied to the AOP and OOP implementations of the program. These characteristics are used as indicators of how representative each candidate is of the general case.

Studies - The number of empirical studies in which the program’s AOP and OOP implementations is used as an indicator of representativeness. This is based on the assumption that these studies are based on implementations that are representative of the general case to ensure generalisability. It also assumes that if AOP and OOP implementations of a program are used across a number of empirical studies, then this indicates that they are representative of the general case.

Deployed - The intended audience of this thesis are industrial practitioners who are considering the adoption of AOP. For evidence to be generalisable for this audience, the program and the MATs on which these studies are based must be representative of the general industrial case. The fact that a program has been deployed indicates that it is a real program and that it contains the concerns such as a GUI, exception handling, persistence, concurrency, and distribution that are generally present in an industrial program. It implies that the program implements the diverse set of concerns, encountered in the industrial case, that must be addressed to ensure a robust program. The usage and deployment of a program are used as indicators that the program is representative of the general case and are used as criteria for selecting candidate implementations on which to base this study.

Maintenance activities - The MATs are more representative if the types of activities applied follow the distribution of types of MATs applied in general practice. There are

three main types of MAT: *perfective*, *adaptive* and *corrective*. *Perfective* MATs improve the quality of an implementation. *Corrective* MATs fix faults in the implementation and *adaptive* MATs implement new requirements. There is evidence [95, 96, 94, 124] to suggest that the general distribution of MAT types is that there are more *perfective* MATs applied to implementations in practice than *corrective* and *adaptive* MATs.

Selection

The Health Watcher (HW) program is the only candidate in Table 5.1 that matches the criteria defined for selection.

Studies - It has been used in three empirical studies that compare the effects of AOP and OOP on maintainability indicators. Each of the other candidates are used in one study. More confidence can be associated with the representativeness of the HW as it is used in more empirical studies than the other programs.

Deployed - The HW has been deployed in a real-world context since 2001. The HW contains concerns, such as view (view is a GUI concern), exception handling, persistence, concurrency, and distribution, which are generally present in an industrial program. Similar to all of the candidates, the AOP and OOP implementations of the HW are developed using the AspectJ and Java languages. AspectJ is the most popular [106] AOP language. It is an extension of Java, which is the most popular OOP language [136]. AspectJ extends Java through the introduction of new constructs such as pointcuts, advice and inter-type declarations to realise AOP concepts. The HW implementations are based on technologies, such as Servlets [71], RMI [65] and JDBC [126] that generally used in Java based implementations.

Maintenance activities - Since its deployment a number of MATs were applied to the initial deployed implementation. Nine MATs are selected based on those applied to the deployed implementation. These MATs are applied cumulatively to the initial equivalent implementations if the AOP and OOP implementations, resulting in ten releases of each implementation. The types of MAT also roughly follow the general distribution of MATs over types [124].

5.1.2 Health Watcher - Use Cases and Maintenance Activities

The purpose of the Health Watcher (HW) system is the registration and administration of complaints to the public health system. The initial release of the HW is based on sixteen use cases [74, 75]. These use cases are identified in Table 5.2 and detailed in full at the TAO website ¹. This website provides open access to the AspectJ and Java implementations of all ten versions of the Health Watcher and the use cases on which

¹<http://www.comp.lancs.ac.uk/greenwop/tao/>

5.1. IMPLEMENTATIONS

Use Case	Type	Target
1	Search	Employee Login
2		Complaint
3		List Complaints
4		List Health Units
5		Health Units by Specialty
6		List Specialties
7		Specialties by Health Unit
8		Disease Type
9		List Disease Types
10	Insert	Employee
11		Animal Complaint
12		Food Complaint
13	Special Complaint	
14	Update	Employee
15		Complaint
16		Health Unit

Table 5.2: Health Watcher Version 1: Use Cases [87]

Use Case	Type	Target
17	Search	Speciality
18		Symptoms
19	Insert	Health Unit
20		Symptoms
21		Speciality
22		Disease Type
23	Update	Speciality
24		Symptoms

Table 5.3: Health Watcher Version 9: Use Cases [87]

ID	Maintenance Activity	Type	Impact
1	Factor out multiple Servlets to improve extensibility	Perfective	View
2	Ensure the complaint state cannot be updated once closed to protect complaints from multiple updates.	Corrective	View/Business
3	Encapsulate update operations to improve maintainability using common software engineering practices.	Perfective	Business/View
4	Improve the encapsulation of the distribution concern for better reuse and customization.	Perfective	View/Distribution/Business
5	Generalize the persistence mechanism to improve reuse and extensibility.	Perfective	Business/Data
6	Remove dependencies on Servlet response and request objects to ease the process of adding new GUI.	Perfective	View
7	Generalize distribution mechanism to improve reuse and extensibility.	Perfective	Business/View/Distribution
8	New functionality added to support querying of more data types	Additive	Business/Data/View
9	Modularize exception handling and include more effect error recovery behaviour into handlers	Perfective	Business/Data/View

Table 5.4: Health Watcher Versions 2 - 10: Maintenance Activities

these implementations are based.

To summarise, the HW enables different types of complaints to be registered, each complaint details the symptoms of the person registering the complaint, the health system employees to be assigned to deal with the complaints at specific health units or clinics, all

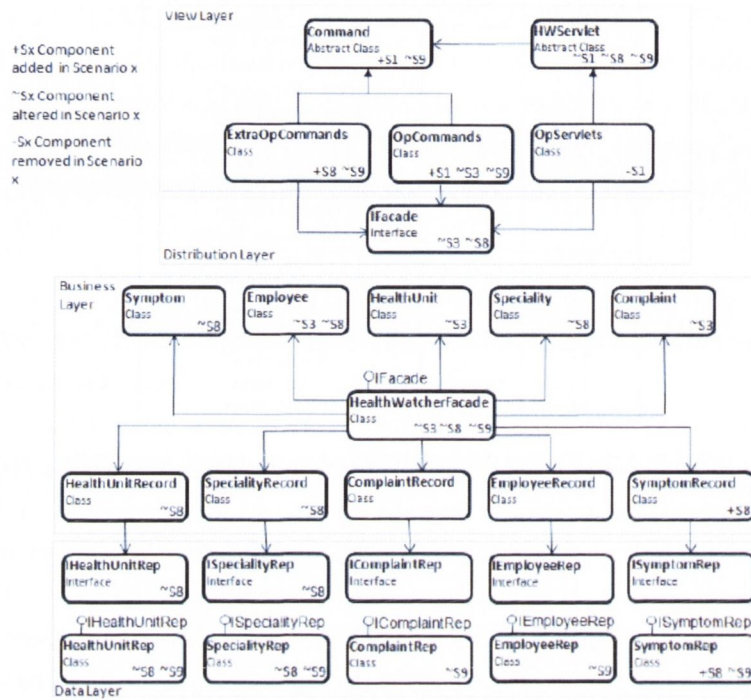


Figure 5.4: Object Oriented Health Watcher [63]

of which specialise in treating different types of diseases. Each use case does one of the following actions: search, insert and update. The table shows which actions are associated with each use case.

Table 5.4 lists each MAT cumulatively applied to the initial HW program. The table provides a brief description of the MAT, its type and the layers to which it applies. The majority of the MATs are perfective, one is corrective and one is adaptive. The eighth MAT is adaptive and implements five new use cases, these are listed in Table 5.3.

5.1.3 Health Watcher - Java and AspectJ Implementations

Figures 5.4 and 5.5 present overviews of the Java and AspectJ implementations of the initial HW. They identify the architectural layers in both implementations and the core modules that implement each layer.

Maintenance Activity Impacts

Each figure identifies the impacts of each MAT (or scenario as termed in the legends of these figures) on the implementation. As illustrated in the legends, Sx identifies a MAT, where x identifies the specific MAT. These numbers are placed in the core modules of each

5.1. IMPLEMENTATIONS

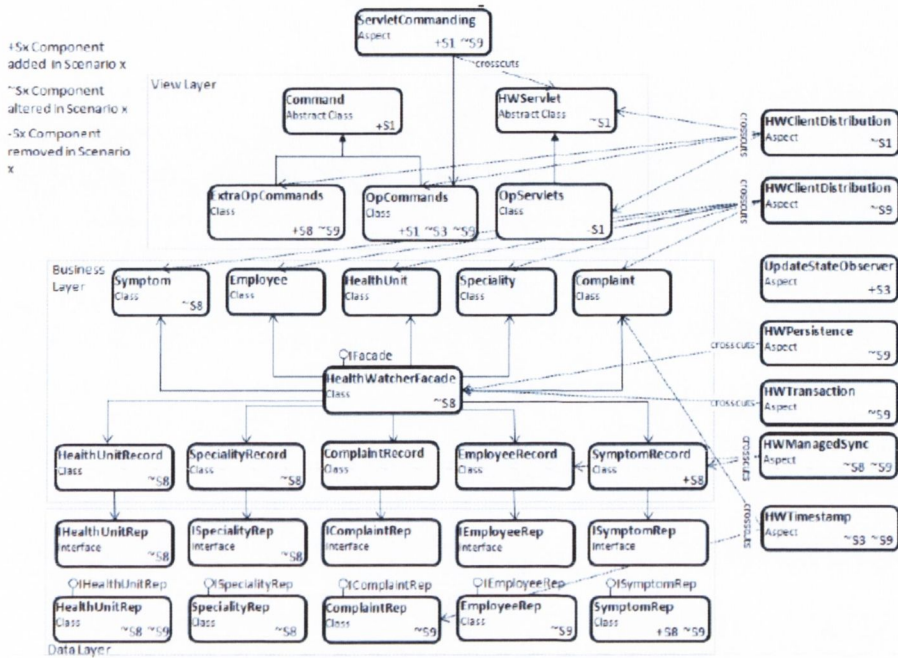


Figure 5.5: Aspect Oriented Health Watcher [87]

implementation to demonstrate the impact of the MAT. If the impact is the addition of a module, then a + symbol is prefixed to the MAT identifier. The ~ symbol indicates that the MAT alters the module and the - indicates that the impact is to remove the module.

For example, at the top of Figure 5.4 the `Command` module is added in the first MAT ($S1+$) and altered in the ninth ($S9$) and the `OpServlets` module is removed in the first MAT ($S1-$).

Client

The figures show that both implement view, distribution, exception handling, concurrency, business, data and persistence concerns. Both implementations follow client-server and layered architectural styles where the view, distribution, business and data concerns are structured as architectural layers. They also show that for both implementations, the view layer is a web client implemented using Java Servlets [71]. This layer accepts http requests and returns http responses. When a request is made, the corresponding command is called on the server through the distribution layer. This call is relayed to the server using RMI [65].

Server

On the server, a client request is invoked on the business layer. The business layer contains the business logic, which manages the data layer by creating, inserting, updating and deleting instances of entities under the management of the HW. The data layer defines entities represented by the system. These entities include complaints, symptoms, health system employees, health units, specialities, disease types and are mapped to persistent database storage. To ensure that the HW can handle many concurrent users, access to entities is managed by the concurrency concern. The implementation of this concern is based on the standard Java blocking mechanism. Once access to an entity is granted and a change is applied, the change must be persisted. The persistence concern manages this process and is implemented using JDBC [126].

During the execution of HW, there are many points of failure at which exceptions are thrown. To handle these failure cases gracefully, the HW implements an exception handling concern. This concern provides facilities to throw exceptions where a failure is recognised, and to catch and deal with these exceptions.

Differences and Equivalence

In the AspectJ implementation, the concurrency, exception handling, distribution and persistence concerns are implemented separately in aspect modules. In the Java implementation these concerns are scattered and tangled across the modules that comprise the Java implementation. This is the only significant difference between these implementations. The initial AspectJ implementation is a refactoring of the Java implementation in which these concerns are aspectised [63].

5.1.4 Summary

The section listed the programs that have the potential to be used in the measurement gathering approach prescribed by the methodology. These are programs for which AOP and OOP implementations exist and MATs have been cumulatively applied to these implementations to create several maintenance versions of both implementations. It described the program selected as most representative of the general case. It also detailed the implementations and maintenance activities associated with the program through their use in empirical studies.

5.2 Mutants

The testability of the HW implementations is measured by applying mutation analysis to each implementation. Mutation analysis measures the testability of an implementation as

5.2. MUTANTS

the rate at which it exposes faults under testing. This rate is derived by executing tests against mutants generated from the implementation. This section describes the approach used to generate the mutants for each AspectJ and Java implementation.

The generation of mutants is based on a fault model. The fault model specifies a list of the types of faults that occur at features of a language [100, 20]. The first part of this section describes the AspectJ and Java fault models on which mutant generation is based in this study.

Mutation operators specify how to generate these types of faults. The second part of this section presents the mutation operators that are used to generate mutants of AspectJ and Java implementations of the HW of the types specified in the fault model.

Mutation operators are applied to implementation through a mutant generation tool. The third and final part of this section describes the mutant generation tool that applies these operators to the AspectJ and Java implementations in the study.

5.2.1 Fault Model

In mutation analysis, the rate of fault exposure for an implementation is the proportion of faults exposed when tests are executed against mutants generated from the implementation. The generated mutants are approximations of the types of faults that occur in practice. These mutants are the basis for the fault exposure rate derived from applying mutation analysis to an implementation. Because the mutants are approximations, the resulting rate of fault exposure is also an approximation. The more representative the generated mutants are of the types of faults that occur in practice, the more the approximated rate of fault exposure is.

Here, the AspectJ and Java fault models are selected as a basis for mutant generation that match **indicators** that these fault models contain the types of faults that occur in practice.

Indicators

There are two ways in which the representativeness of a fault model can be indicated. The first is by tracing its evolution. A fault model defines the types of faults that are observed to occur at features of a language. As languages evolve so too do fault models. When a new language such as Java evolves, the language introduces some new features but inherits many of the features from the languages from which it has evolved. Fault models evolve in the same way. When a new language evolves from older languages, the fault types associated with the inherited features are also inherited into the fault model for the new language. These can be refined in the inheritance process. For the new features, various fault types are proposed based on empirical studies or through retrospective observation. These are then amalgamated through convergence. Convergence occurs when a set of

independent researchers converge on the same set of fault types for new language features. If a fault model is highly evolved, then this indicates that the fault types in the model have undergone generations of refinement to ensure that they are highly representative of the types of faults that really do occur in practice.

The second indicator that a fault model specifies the types of faults that are representative, is based on how often used it is. If the fault model is the basis for mutant generation in a large number of scientific studies, then this indicates a pragmatic acceptance by consensus that the fault types specified in this model are representative of the types of faults that occur in practice.

Fault Model Selection

The Java fault model, introduced by Ma et al [98], and the AspectJ fault model, introduced by Ferrari et al [53] were chosen. These fault models are selected because they are the most evolved and contain the most refined set of fault types for the features defined the Java and AspectJ languages. The fault types specified in this model are also accepted by consensus to be representative of the types of faults that occur in practice.

Java Fault Model - The Java language is based on features inherited from imperative languages and new object oriented features.

The types of faults defined for the imperative features of Java have evolved from fault models defined for Ada, C and FORTRAN [98, 86, 127, 44, 118]. The C language is a predecessor of Ada and FORTRAN can be seen as the basis for C [16]. As these imperative languages evolved, the elements of older languages that were considered useful were used as the basis for new languages. The types of faults associated with these elements were also brought forward into the newer fault models. The FORTRAN fault model was derived from studies of programmer errors and corresponds to simple errors that competent programmers typically make [86]. The C fault model is an evolution and refinement of the FORTRAN fault model [127] and the Ada fault model is a further refinement of both the C and FORTRAN fault models [118]. In both cases, these refinements were designed to ensure that the fault models were representative of those observed in practice. The fault types in these models are further refined by Offutt et al [114] for use in the Java fault model [98],

The types of faults defined in the Java fault model [98] are based on a number of fault models that have been refined over time [56, 77, 78, 115, 37, 99, 98]. Kim et al. [77, 78] identify a number of studies in which specific types of faults occur in practice at object-oriented features of the Java language. They combine these into an initial Java fault model based on object-oriented specific elements including dynamic binding and inheritance related types of fault. Offutt et al. also identify an initial fault model based on the same object-oriented features. Firesmith [56] and Chevalley [37] identify fault

5.2. MUTANTS

types based on their experience that are based on common programmer mistakes when using object oriented and Java specific features. Ma et al. [99] selectively amalgamate, categorise and refine these fault models into a set of fault types that are representative of those observed in practice.

The Java fault model has been widely used the basis for mutation analysis in many empirical testing studies [151, 122, 133, 104, 103, 102, 105, 130, 134]. This usage indicates an acceptance by consensus that the fault types specified in this model are representative of the types of faults that occur in practice.

AspectJ Fault Model - The types of faults defined for the AspectJ features are based on the evolutionary refinement of candidate fault models [3, 12, 36, 47, 145] and fault classification [90]. There have been a number of candidate fault models proposed for AspectJ, the first of which was proposed by Alexander et al. [3]. This fault model identified a number of roughly defined fault types that could occur in pointcuts and advice. Bakken and Alexander then refined these rough definitions into a more precise set of fault types [12]. Ceccato et al. [36] extend the set of faults types from those that can occur at inter-type declarations. Van Deursen et al. add conditional and pattern based fault types to the set of known fault types. Lemos et al. strengthen the classification of types of faults that occur at pointcuts [90]. Eaddy et al. further refine and extend the set of fault types by recognising contextual and object identify based fault types [47].

AspectJ is relatively new and has not yet "crossed the chasm" [107] to widespread deployment. Because of this, evidence of the types of faults that occur at the AOP specific features of AspectJ in practice is sparse. Instead, the state of practice is inferred by researchers through convergence [79]. This occurs when a set of independent researchers converge on the same set of fault types for AspectJ. Convergence is illustrated by Zhang and Zhao [155]. They identify a number of bug patterns for AspectJ. A bug pattern is similar to a fault type but the intention of their bug pattern is to help in debugging AspectJ programs rather than to generated faults for them. These bug patterns are similar to the refined AspectJ fault model [3, 12, 36, 47, 145]. This illustrates convergence on this model. More recent work on testing AspectJ programs have selectively used these fault models as a basis for their research [119, 11, 91, 90, 17, 18]. This indicates a convergence and acceptance of these models as the current state of the practice for use in testing and testability related research focused on AOP.

5.2.2 Mutant Operators

Fault models are the basis for mutation operators. Mutation operators specify how to generate mutants containing faults of the types identified in the selected fault model. In this subsection, the mutation operators for the selected Java and AspectJ fault models are described using examples.

Java Operators

Table 5.5 lists the mutation operators for the Java fault model, introduced by Ma et al [98]. Each mutation operator is associated with an OOP feature of the Java language and generates mutants that contain faults of a specific type. The action the mutation operator takes to generate a fault is briefly described. This description is also representative of the type of fault generated by the operator.

Examples of operators in Table 5.5 are the Java `this` Deletion (JTD) and Java `this` Insertion (JTI) operators. These operators are associated with the `this` keyword which is classified as a specific feature of the Java language. The application of these operators is illustrated in Section 4.1 to demonstrate the mutant analysis procedure. The JTI operator inserts the `this` keyword. The JTD operator deletes the `this` keyword. The resulting faults are representative of the types of faults that occur when developers forget to use the `this` keyword where they had intended or used it where they had not intended. A complete description of all of these operators is provided by Ma et al. [98] on a website² dedicated to MuJava, the tool that implements these operators.

AspectJ Operators

Table 5.6 lists the mutation operators which are based on the selected AspectJ fault model, introduced by Ferrari et al [53]. Each operator is associated with an AOP specific feature of the AspectJ language and the action taken by the operator to generate a fault of a specific type is briefly described. An example of an operator in Table 5.6 is the Around Proceed Statement Removal (APSR) operator. The application of this operator is demonstrated in Section 4.2. The APSR operator removes statements in which calls using the `proceed` keyword are made. The faults generated by the APSR operator are representative of the type of fault that occurs when the developer forgets the `proceed` call in AspectJ `around` advice. A complete description of all of these operators is provided by Ferrari et al [53].

5.2.3 Mutant Generation Tool

MuJava is a well-established tool for generating Java mutants. However, when work began on the study described in this thesis, there was no tool available for generating AspectJ mutants. This section describes the MuJava and its extension for AspectJ mutant generation.

²<http://cs.gmu.edu/~offutt/mujava/>

5.2. MUTANTS

Fault Type	Operator	Description
Traditional	AORB	Binary Arithmetic Operator Replacement
	AORS	Short-cut Arithmetic Operator Replacement
	AOIU	Unary Arithmetic Operator Insertion
	AOIS	Short-cut Arithmetic Operator Insertion
	AODU	Unary Arithmetic Operator Deletion
	AODS	Unary Arithmetic Operator Deletion
	ROR	Relational Operator Replacement
	COR	Conditional Operator Replacement
	COD	Conditional Operator Deletion
	COI	Conditional Operator Insertion
	SOR	Shift Operator Replacement
	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
ASRS	Assignment Operator Replacement	
Inheritance	IHI	Hiding variable insertion
	IHD	Hiding variable deletion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
Overloading	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java Specific	JTI	this keyword insertion
	JTD	this keyword deletion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialisation deletion
	JDC	Java-supported default constructor creation
Common	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Table 5.5: Java Fault Model

MuJava

The MuJava tool [98] implements the Java mutation operators identified in Table 5.5. This tool, an overview of which is illustrated in Figure 5.6, generates mutants given a Java source file. The mutant generation process begins by parsing the source file into a model of the code. This model is traversed to identify locations containing features with which mutation operators are associated. Once a location is identified, the mutation operators associated with the feature at the location are applied to a copy of the Java source file. The copied source is altered in line with the description given for the operator in Table 5.5. This alteration results in a new version of the source code that deviates from

Fault Type	Operator	Description
Pointcut	PWSR	Replace a type with its immediate supertype
	PWIW	Insert wildcards into pointcut expressions
	PWAR	Remove annotation tags from patterns
	POPL	Change parameter lists
	POAC	Change after advice clauses
	POEC	Changing exception throwing clauses
	PSSR	Replace a type with its immediate subtype
	PSWR	Remove wildcards from pointcut expressions
	PSDR	Remove declare @ statements
	PCTT	Replacing a this pointcut designator with a target one
	PCCE	Switch pointcuts designators
	PCGS	Replace a get pointcut designator with a set one
	PCCR	Replace individual parts of a pointcut composition
PCLO	Change composition operators	
PCCC	Replace a cflow with a cflowbelow	
Advice	ABAR	Replace a before clause with an after
	APSR	Remove invocations to proceed statement
	AJSC	Replace a join point reference with enclosing
	ABHA	Removing implemented advices
	ABPR	Replace pointcuts which are bound to advices
Declaration	DAPC	Alter the order of aspects in declare precedence
	DAPO	Remove declare precedence
	DSSR	Remove declare soft
	DEWC	Change declare error/warning
	DAIC	Changing instantiation clauses

Table 5.6: AspectJ Fault Model

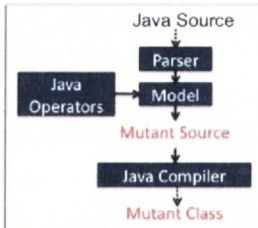


Figure 5.6: MuJava

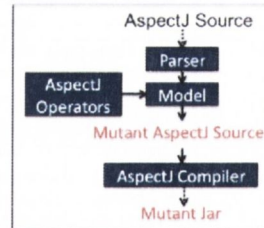


Figure 5.7: AspectJ Extension

the original source. This deviation represents a fault.

The mutant version of the source code contains a fault of the type represented by the mutation operator used to generate the mutant. The mutant versions of the source code is then compiled using the Java compiler to create a mutant class. The mutants that do not compile are discarded. The mutants that compile are recorded in a mutation log. This log lists mutants that pass compilation and the locations for which the mutant is generated.

MuJava/AspectJ

MuJava/AspectJ is an extension of MuJava, developed specifically for this study, in which the operators identified in Table 5.6 are implemented. As illustrated in Figure 5.7, this extension uses the ajdt [40] parser to parse AspectJ source code into a code model. This

5.3. TESTS

model is traversed to identify locations containing the AOP specific features presented in Table 5.6. When these features are identified their associated mutation operators are applied to the location in precisely the same way as MuJava. For each application of a mutation operator, a copy of the AspectJ source code is generated. The mutation operator then introduces a deviation in the copied code. This code is compiled by the AspectJ compiler and if compiled, the mutant is logged.

One difference between MuJava and this extension is that when the mutant AspectJ source code is compiled, a new mutant Jar of the entire HW program containing the fault is generated rather than simply a new mutant class. This is because the mutation of compositional features, such as pointcuts, require the reweaving of the entire program. Compositional features at the source code level, such as pointcuts, specify dependencies that are generated at the byte-code level. Mutation of these compositional features can cause a change to the intended dependencies. To ensure that mutation at the source code level is reflected at the byte-code level the entire program including the mutant AspectJ source code must be compiled using the AspectJ compiler.

5.2.4 Summary

This section described the approach used to generate the mutants for each AspectJ and Java implementation. This section described AspectJ and Java fault models on which mutant generation is based in this study. These fault models are highly refined are agreed to be representative of the types of faults that occur in practice. The mutation operators that generate faults of these types are outlined and the tool that applies these to AspectJ and Java implementations is described.

5.3 Tests

This section describes the approach used to select and develop tests for used in mutation analysis in the study. The section outlines the different test selection approaches that could have been followed and justifies the chosen approach. It describes the chosen approach, its application to the HW and the resulting selected tests. It also describes the approach used to automate the execution of these tests against implementations and mutants.

5.3.1 Choosing a Test Selection Approach

There are two candidate approaches to test selection. The first candidate is white box test selection. White box test selection is based on knowledge of the internal structure of the implementation. In this approach, knowledge of the internal structure is used as a basis for selecting tests. An example of white box test selection is test selection based on control flow analysis. As illustrated in Sections 4.1.1 and 3.1, control flow analysis

identifies the paths of execution through an implementation. These are used as a basis for test selection. In these approaches, test inputs are selected to exercise these identified paths.

The second candidate is a black box test selection approach. Black box test selection is based on external knowledge of the implementation, such as its requirements. An example of black box test selection is a use cases driven approach to test selection [5, 23]. In this approach, test inputs are selected to exercise the requirements of a program.

Selection Criteria

There are two criteria for choosing an approach to test selection.

Representative - The first criterion is that the approach and resulting tests must be representative of what is used in practice for selecting tests at the implementation level. Implementation level testing is testing focused on the programs interface [5] .

Practical - The second is that resulting test selection must be practical in terms of the resources required to apply the approach.

Applicable - As illustrated in Figure 5.3, the methodology on which this study is based requires one test suite that is applicable to both sets of implementations. The third and final criterion is that the approach must result in a set of tests that is applicable to the Java and AspectJ implementations of each version of the HW program.

Approach Selection

A black box approach is selected because it is more representative, practical and easily applicable than a white box approach.

Representative - White box approaches to test selection are based on analysis of the internal structure of the implementation. In practice, this approach is used at the unit, module and integration levels of testing. Black box approaches to test selection is based on analysing of the programs. In practice, this approach is used at the implementation level, i.e at the level of the programs interface.

Practical - A white box approach to test selection is expensive in terms of the resources needed to apply the approach. In contrast the application of a black box test selection approach is relatively inexpensive.

Applicable - White box approaches to test selection are based on analysis of the internal structure of the implementation. A white box approach applied to the Java and AspectJ implementations of each of the ten version of the HW program will result is twenty different sets of tests, one for each implementation. Each test set would be directly applicable to the implementation for which it was selected but would not be directly applicable to the other implementations.

5.3. TESTS

These test sets could be merged into a super test suite that would then be applicable to all implementations but would subsume each selected test suite. White box test selection is known to be expensive [6]. The expense incurred by applying this approach at a program level is impractical [5, 23] and not generally used in practice [113, 61].

A black box approach to test selection is based on analysing of the requirements and interface of the program. Because each version of the HW share the same requirements and interface tests can be selected using this approach that are applicable to all implementations. This approach is inexpensive, relative to white box test selection, and is generally used in practice [113, 61] at the program level [5, 23].

5.3.2 Black Box Test Selection

A black box approach to test selection, based on use cases, is chosen to select tests for use in this study [74, 75]. To illustrate this approach, its application to the *Employee Login* use case from the HW program is presented. The application of this approach to the use cases identified in Tables 5.2 and 5.3, on which the HW program is based, is the described.

Example Use Case

The *Employee Login* use case is presented in Table 5.7. The table is based on the full description of all use cases available at the TAO [64] website. This is the website at which all of the HW implementations and use cases are available.

The inputs for the use case are an employee id and a password. The output of the use case is a validation of the password. The use case also describes a main and alternative flow. The main flow describes the typical steps in the login process when a valid employee id and password pair are used to login. In this case, the HW program declares these as being valid and logs the employee into the program. The alternative flow describes the steps when an invalid employee id and password pair are used to login. The HW program declares these as being invalid and informs the user that they cannot be logged in.

Test Selection

Test inputs are selected to exercise the flows through the use case. In the *Employee Login* use case, there are two flows. The first is the main flow, in which a valid employee id and password pair are provided to the program. The second is the alternative flow, in which an invalid employee id and password pair are provided to the program.

To exercise the main and alternative flows, valid and invalid employee id and password pairs are needed. The validity of an employee id and password pair depends on whether

Use case	<i>Employee login</i>	
Summary	Log in to the HW system	
Inputs	employee id and password	
Output	Password validation	
Flow	Step	Description
Main 1	1.1	User provides valid employee Id and password pair
	1.2	Employee id and password pair are identified as valid
	1.3	Employee is logged into the HW
Alternative 2	2.1	User provides an invalid employee Id and password pair
	2.2	Employee id and password pair are identified as invalid
	2.3	Employee is not logged into the HW

Table 5.7: Employee Login Use Case

Flow	Test	Employee id	Password
Main	1	Andrew	TCD
	2	Siobhán	Trinity
Alternative	3	Andrew	Trinity
	4	Siobhán	TCD
	5	Bill	USA
	6	Bill	TCD

Table 5.8: Examples of Test Selection

Employee id	Password
Andrew	TCD
Siobhán	Trinity

Table 5.9: Test Data

the pair is in the HW database. To select tests that exercise these flows sets of employee id and password pairs and a HW database containing employee test data are needed.

Table 5.8 illustrates sample employee id and password pairs, and employee test data to populate the HW database is illustrated in Table 5.9. These are selected to exercise the main and alternative flows of the *Employee Login* use case.

Tests 1 and 2 exercise the main flow of the *Employee Login* use case. These tests contain employee id and password pairs that are valid because they are contained in the employee test data and should result in a successful login. Tests 3 to 6 exercise the alternative flow of the *Employee Login* use case. These tests contain employee id and password pairs that are invalid because they are not contained in the employee test data and should result in an unsuccessful login.

Application to HW Use Cases

The use case driven approach to test selection is applied to the use cases, listed in Tables 5.2 and 5.3. The use case driven approach to test selection was applied to the use cases, identified in Table 5.2 and 5.3, by a group independent of this study. There were seven software testing professionals, all from different Irish software development companies, with a minimum of four years of industrial experience in this group. The group were not told about the study presented in this thesis.

They were provided with the full description of each use case identified in Tables 5.2 and 5.3 and screen shots of the HW web interfaces associated with each use case. The database tables used by the HW program and some sample data to illustrate the types

5.3. TESTS

Versions	Use Case	Type	Target	Tests
1-10	1	Search	Employee Login	18
	2		Complaint	16
	3		List Complaints	4
	4		List Health Units	8
	5		Health Units by Specialty	13
	6		List Specialties	4
	7		Specialties by Health Unit	13
	8		Disease Type	17
	9		List Disease Types	4
	10	Insert	Employee	13
	11		Animal Complaint	14
	12		Food Complaint	14
	13	Special Complaint	14	
	14	Update	Employee	15
	15		Complaint	14
	16		Health Unit	14
9-10	17	Search	Speciality	4
	18		Symptoms	4
	19	Insert	Health Unit	10
	20		Symptoms	10
	21		Speciality	10
	22		Disease Type	10
	23	Update	Speciality	11
	24		Symptoms	11

Table 5.10: Tests Selected for Use Cases

of data held in each table were also provided. The use of software testing professionals to select tests ensured that the application of the use case driven approach and the resulting tests are representative of the general practice.

Selected Tests

Table 5.10 illustrates the number of tests selected for each use case. Overall, there are two hundred and sixty five tests selected. For each use case, tests are selected to exercise each flow in the use case. The table also shows a subset of the test set is applicable to all ten versions of the HW program. This is because additional use cases are introduced in the eighth maintenance activity applied to the HW program. This means that the entire test set is applicable to only versions nine and ten of the HW program.

5.3.3 Test Execution Automation

There are twenty implementations of the HW. In the location execution phase of Mutation Analysis in this study, detailed in Section 4.1, each of the selected tests must be regressively executed against all twenty implementations. In the fault exposure phase, also detailed in Section 4.1, each test must be regressively executed against all mutants of all twenty implementations. This repeated execution of tests is manually unfeasible and as such the execution of tests was automated.

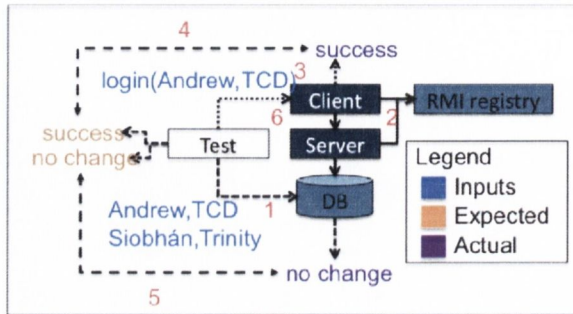


Figure 5.8: Automated Employee Login Test

Automation Steps and Frameworks

Each automated test executes six steps, as illustrated in Figure 5.8. This test exercises the employee login function of the HW. The test has two inputs. The first are pairs of employee id and passwords used to populate the database. The second is the *Andrew, TCD* employee id and password pair that are used as arguments to employee login function.

The first step in the automated test execution process is populating the database with the input test data provided by the test. Each test is based on the db-unit framework [67] which provides the facility to automatically load test data into the database as a precursor for test execution.

The second step is to initialise the HW program, which requires starting an RMI registry [65], the server and the client in a strict sequence. In this sequence, the RMI registry is started first. The server is started next. The server establishes connections to the database and initialises its data layer, which is used to represent the data held in the database. The server registers with the RMI registry and once registered, the client is started. The client is web based and exposes a http interface. The client retrieves a reference to the server from the RMI registry and waits for http requests. To initialise these components in the correct sequence, a framework for starting a distributed process, called spawn [70] is used.

The third step is to execute the function exposed through the web client's http interface. To execute a function requires that a http request be sent to the web client. Each test is based on the http-unit framework [140]. This enables tests to send http requests to web interfaces. Once the request is received by the web client, it is executed. The web client executes and forwards the request to the server, which checks to see whether the employee id and password pair are in the database. The test data used to populate the database contains pair and as such the server returns that the login attempt is valid.

The web client sends a response to the request that indicates a successful login. The fourth step is to check whether this response is the response that was expected. The

5.3. TESTS

expected response is a successful login. This matches the actual response.

The fifth step is to check whether the execution has left the database as expected. The test's execution has left the database unchanged which is as expected. This is asserted through the db-unit framework [67], which provides the facility to check the contents of the database after the execution of a test. Because both the actual response and the database are as expected after the test executes the test passes. If these were not as expected then the test would fail.

The sixth and final step is to stop the HW program. This requires the client, server and the RMI registry to be stopped. The client is stopped first, followed by the server and the RMI registry. This stopping sequence is controlled by the spawn framework.

Location Execution

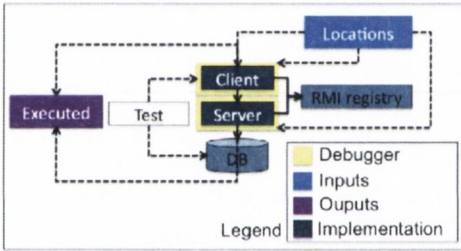


Figure 5.9: Location Execution

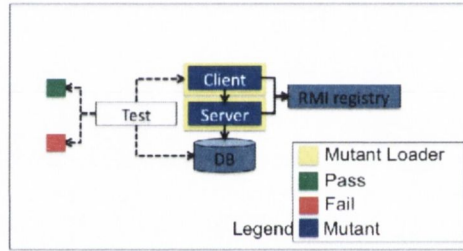


Figure 5.10: Fault Exposure

In the location execution phase of mutation analysis, detailed in Section 4.1, each test is executed against each implementation to identify the locations executed by each test. These locations are identified by executing the tests against the program's implementation.

Figure 5.9 illustrates the execution of the test against the program's implementation. To automatically identify the locations executed by each test, the client and server components of the HW are wrapped by two debuggers. The debuggers take the identified locations as inputs and converts them into breakpoints. As the client and server execute, the breakpoints that are hit during the execution are recorded.

The debuggers are based on the standard Java Debugger (JDB) tool shipped with the Java Standard Development Kit [10]. The JDB is customised to function as a wrapper around the distributed client and server components of the HW.

Fault Exposure

In the fault exposure phase of mutation analysis, also detailed in Section 4.1, each test is executed against each mutant. Figure 5.10 illustrates the execution of the test against a mutant.

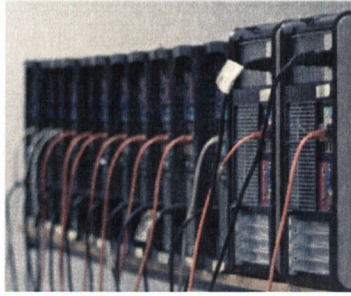


Figure 5.11: Distribution of Test Executions

To execute a mutant requires a mutant loader. This is a modified class loader that loads the mutant implementation. There are two kinds - one for Java mutants and another for AspectJ mutants. The Java mutants are mutant classes which are loaded with unmutated classes. The AspectJ mutants are mutant Jar files which contain the class files for the entire mutant implementation.

Once the mutant is loaded, the test is executed against the mutant implementation. The mutant contains a fault at an executed location. This location may be on the server, on the client or on both. The test executes and if the test passes then the test is not exposed. If the test fails the fault is exposed. The outcomes of all test-mutant executions are recorded for analysis.

Execution Environment

Mutation Analysis is known to be computationally expensive [57, 135, 116]. In the fault exposure phase, each test is regressively executed against all mutants of all twenty implementations. This is highly computationally expensive because an exhaustive set of mutants are generated for each implementation. This means that there are a large number of mutants that need to be executed by tests. As will be shown in the next chapter, thousands of mutants are generated for each implementation. Each of the 265 tests can potentially be executed against each mutant. This means that there is a very large number of tests need to be executed against mutants in this stage.

To address these issues, the mutants generated for each implementation were deployed onto separate linux machines. Some of these machines are illustrated in Figure 5.11. The set of automated tests were also deployed on each machine along with the required infrastructure for mutant execution. This ensured that the test-mutant execution could be carried out in parallel for each implementation. Parallelisation of the mutation analysis procedure has been identified as an approach for addressing the need for a large amount computational resources [57]. Without parallelisation, this study would have been unfeasible.

5.4. CHAPTER SUMMARY

5.3.4 Summary

This section described why a use case driven approach to test selection is representative of the general case. It indicates that professional software testers applied this approach to the use cases on which the Health Watcher is based, to create a set of tests, which ensures the resulting tests are representative of the general case. It also described how the selected tests were automated and the environments in which these tests were used in the mutation analysis procedure, detailed in Section 4.1.

5.4 Chapter Summary

An inherent goal of the study presented in this thesis is to gather evidence from which general conclusions about the comparative effect of AOP and OOP on testability can be drawn. General conclusions can only be drawn from evidence gathered in contexts that are representative of the general case. This chapter has described the selection of the three inputs to create a context that is representative of the general case. This is done to maximise the degree to which general conclusions can be drawn from evidence gathered in the study.

In each section inputs are selected that have the potential to be used in the measurement gathering approach, prescribed by the methodology detailed in Chapter 4. The first section selected the evolutionary implementations of the Health Watcher program from a set of candidates because they best fitted the defined indicators of representativeness. The second section selected the MuJava/AspectJ tool to generate mutants of these implementations because this tool generates mutants that contain faults that are representative of the types of faults that occur in practice. The third and final section describe an approach to select a set of representative tests to execute against the mutants generated for each AspectJ and Java implementation of the Health Watcher program.

These selections maximise generalisability of evidence gathered in this study because together these inputs form a context that is representative of the general case. The results of applying MA in this context and evidence derived by analysing these results are presented in the next chapter.

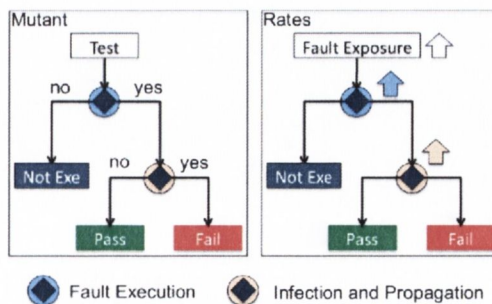
Chapter 6

Study Results and Analysis

In Mutation Analysis (MA), mutants that contain a fault are generated from implementations. Over the phases of MA, the outcomes of executing a test set against each mutant are derived. As illustrated on the left-hand side of Figure 6.1, for each test-mutant execution there are three possible outcomes. The test can exercise a path through the implementation where the fault is *not executed*. Alternatively, if the fault is executed, then the fault can either cause state infection and propagation of that infected state into the output, or not. In the first case, the output will differ from what is expected and result in a *fail* outcome. In the second case however, the output of the test will be as expected and result in a *pass* outcome.

In this study, MA is applied to AspectJ and Java implementations of ten versions of the Health Watcher (HW) program. The outcomes of the test-mutant executions for each implementation are used as a basis to derive rates of *fault exposure*, *fault execution* and *infection and propagation* for those implementations. To calculate these rates for each implementation the number of *not exe*, *pass* and *fail* outcomes from the test-mutant executions are counted.

The primary focus of this thesis, as outlined in chapters 1 and 4, is to compare the



effects of AOP and OOP on the rate of *fault exposure*. As detailed in chapters 3 and 4, the rate of *fault exposure* is caused by the rates of *fault execution* and *infection and propagation*. A secondary focus of this thesis is to understand why the effects of AOP and OOP on the rate of *fault exposure* differ. This is achieved by comparing and analysing the effects of AOP and OOP on the rates of *fault execution* and *infection and propagation*.

The rate of *fault execution* is calculated as $rate = \frac{fail+pass}{fail+pass+notexe}$. This rate represents the proportion of test-mutant executions where the fault contained in the mutant is executed. The rate of *infection and propagation* is calculated as $rate = \frac{fail}{fail+pass}$. This rate represents the proportion of test-mutant executions in which the fault is executed result in state infection and propagation, resulting in test failure and *fault exposure*. The rate of *fault exposure* is calculated as $rate = \frac{fail}{fail+pass+notexe}$. This rate represents the overall proportion of test-mutant executions that result in test failure and *fault exposure*.

The right hand side of Figure 6.1 illustrates the causal relationship between these rates. The rate of *fault exposure* is directly caused by the rates of *fault execution* and *infection and propagation*. The higher the rate of *fault exposure*, the more faults that are executed. The more faults that are executed, the more possibilities there are for state infection and propagation of infected state. That is, the higher the rate of *infection and propagation*, the more fail outcomes. The more fail outcomes, the higher the rate of *fault exposure*.

In this chapter, the results of applying MA to the AspectJ and Java implementations of the ten versions of the Health Watcher (HW) program are presented. The number of *not exe*, *pass* and *fail* outcomes from the test-mutant executions are counted for each implementation. Based on these counts, the rates of *fault exposure*, *fault execution* and *infection and propagation* for each implementation are calculated. The effects of AspectJ and Java on these rates are compared over the ten versions of the HW program using graphical analysis. The comparative effects of these AspectJ and Java on these rates are then quantified by applying binomial regression analysis to each rate.

This analysis is based on the outcomes from executing tests against mutants generated for each implementation. Before the analysis of rates is presented, an analysis of the mutants generated for each implementation is presented. As outlined in Section 4.3, mutant equivalence is an assumption on which this study is based. The generated mutants are equivalent if they differ only in the implementation from which they are generated.

The remainder of this chapter is organised as follows. The first section of this chapter presents an analysis of the mutants generated for each implementation. The second section compares the rates of *fault exposure*, *fault execution* and *infection and propagation* for AspectJ and Java implementations. The third section presents the results of quantifying the comparative effects of AspectJ and Java on each rate. The fourth section outlines the

6.1. COMPARISON OF GENERATED MUTANTS

threats to the validity of the results and analysis and discusses the impact they have on how the results can be interpreted. The chapter is concluded by summarising each step in the comparative analysis and extracting the evidence of the comparative effect of AOP and OOP on testability from the results.

6.1 Comparison of Generated Mutants

The MuJava/AspectJ mutant generation tool is applied to the Java and AspectJ implementations of the (HW) in the mutant generation phase. Table 6.2 presents the number of *mutants* that are generated by this tool for each *implementation* of each *version* of the HW. It also lists the number of *locations* in each implementation at which mutants are generated.

6.1.1 Results of Mutant Generation

The information in Table 6.2 is visualised in the graph illustrated in Figure 6.3. The x-axis of this graph represents the version of the HW and the y-axis represents the number of mutants generated. Each point in the graph represents the number of mutants generated in a Java or AspectJ implementation. The points for both types of implementation are differentiated by the colour of each point. The size of each point represents the number of locations at which the mutants were generated. There are two lines through the graph, one that connects the points for the Java implementation and the other that connects the points for the AspectJ implementation. Each line indicates the change in the numbers of mutants generated over versions for the AspectJ and Java.

Figure 6.3 shows that the number of mutants that are generated for Java and AspectJ implementations increase steadily over versions. The 4367 mutants generated for the initial Java implementation, increases steadily to 4551 at version 8. The number of mutants generated then jumps to 4815 in version 9 and to 4961 in version 10. This jump is explained by the implementation of five additional use cases in the maintenance activity implemented in version 8 and the addition of a logging feature in the maintenance activity implemented in version 9.

There are 4447 mutants generated for the initial AspectJ implementation. This is 80 mutants more than the equivalent Java implementation. The number of mutants generated jumps from 4472 at version 2, to 4726 at version 3. This jump is a response to a significant re-factoring of the AspectJ implementation needed to perform a corrective maintenance activity to version 2. This increases the difference between the number of mutants generated for the AspectJ implementation to approximately 300 mutants over the versions 3 to 8. A steady increase is then observed up to version 8 where another jump from 4727 of mutants generated to 4815 at version 9 and 5061 at version 10. These jumps

are also in response to the additional use cases and logging features added at versions 9 and 10. At version 10 the difference in the number of mutants generated for the Java and AspectJ implementations is 100 mutants.

This difference between the lines shows that more mutants are generated for the AspectJ implementation over each of the ten versions of the HW. A statistical comparison of these measures, presented in appendix Listing A.1, indicates that this difference is not significant when the overall number of mutants generated for both implementations is considered. A comparison of the size of the points on each line shows that the number of locations at which mutants are generated are similar, but slightly larger for the Java implementation. Another statistical comparison of these measures, presented in appendix Listing A.2, indicates that the difference in the number of locations at which mutants are generated for Java and AspectJ implementations is not significant. This indicates that the number of locations at which mutants are generated is roughly the same for Java and AspectJ which reaffirms the equivalence of the mutants generated for each pair of implementation.

Because both sets of implementations are equivalent and similar mutation operators are applied to these implementations using precisely the same mutation generation process we expect there to be no significant differences between the number of mutants generated for Java and AspectJ implementations of the number of locations at which mutants are generated. Both statistical comparisons match our expectation and suggest that the results of the mutant generation for the Java and AspectJ implementations are equivalent.

6.1.2 Mutant Equivalence

Section 4.2 shows that this study is based on implementation and maintenance equivalence. If each pair of AspectJ and Java implementations differ only in the approach used for their development then these are implementation equivalent. If the only difference between respective AspectJ and Java implementations of HW versions is the maintenance activity that created the new version of the HW, then these implementations are maintenance equivalent. A goal of the mutant generation phase, identified Section 4.2, is to generate mutants for AspectJ and Java implementations that preserve implementation and maintenance equivalence.

Equivalence is preserved if the only differences between mutants is that they were generated from AspectJ or Java implementations of different versions of the HW. A two-pronged approach was followed to evaluate whether the generated mutants had preserved implementation and maintenance equivalence. First, the number of mutants generated is compared with the size of the implementations. If they are correlated, then this indicates that the mutants are reflective of the implementation from which they were generated. If the number of mutants reflect the size of implementations from which they are generated,

6.1. COMPARISON OF GENERATED MUTANTS

impl	version	locations	mutants
Java	1	1014	4367
	2	1032	4445
	3	1031	4471
	4	1033	4473
	5	1059	4528
	6	1068	4545
	7	1076	4547
	8	1079	4551
	9	1192	4815
	10	1241	4961
AspectJ	1	1004	4447
	2	1012	4472
	3	1050	4726
	4	1049	4729
	5	1049	4729
	6	1055	4749
	7	1063	4753
	8	1058	4727
	9	1135	4889
	10	1209	5061

Figure 6.2: Mutant Generation

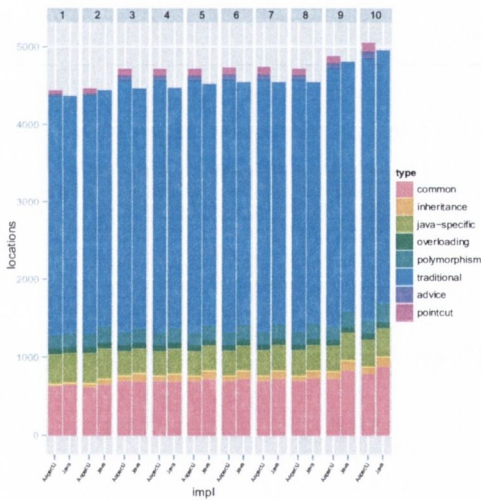


Figure 6.4: Types of Faults

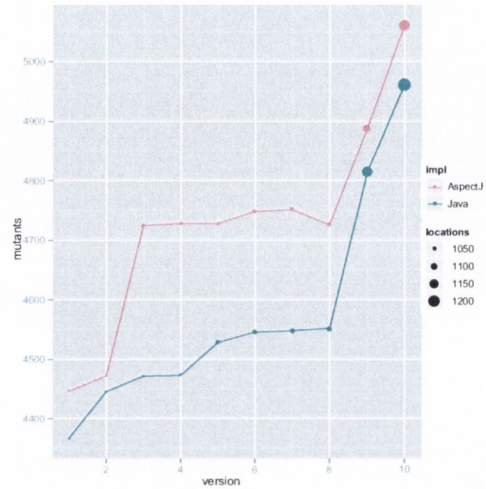


Figure 6.3: Measure Visualised

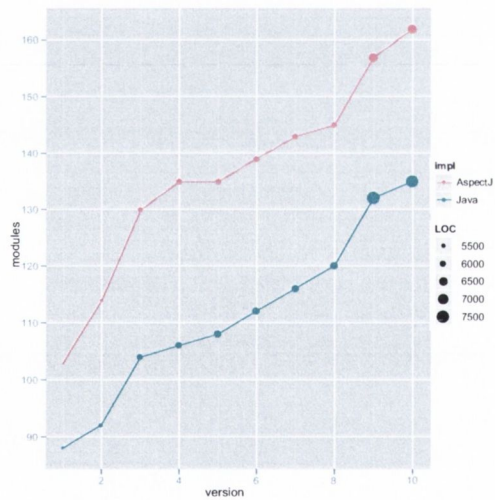


Figure 6.5: Size for Correlation

then this suggests that these mutants are equivalent.

Second, the types of mutants generated for each implementation are compared. As shown in Section 5.1, the Java and AspectJ implementations are similar. This is because the initial AspectJ implementation is the result of refactoring of the initial Java implementation. This similarity means that similar types of faults should be generated in mutants for Java and AspectJ implementations. If the mutants generated for Java and AspectJ contain similar types of faults, then this suggests that these mutants are equivalent.

Correlation of Mutants Generated and Implementation Sizes

A correlation between the number of mutants generated and the size of the HW is illustrated by comparing Figures 6.3 and 6.5. Figure 6.3 shows the changes in the number of mutants generated for each implementation over versions of the HW. Figure 6.5 shows the changes in the size¹ of the AspectJ and Java implementations over the ten versions of the HW. The x-axis represents the version of the HW and the y-axis represents the number of modules in an implementation. Each point in the graph represents the number of modules in a Java or AspectJ implementation. The points for both types of implementation are differentiated by the colour of each point. The size of each point represents the number of Lines Of Code (LOC) in an implementation. There are two lines through the graph, one that connects the points for the Java implementation and the other that connects the points for the AspectJ implementation. Each line indicates the changes in the size of Java and AspectJ implementations over versions.

Comparing Figures 6.3 and 6.5 shows that the changes in the size of AspectJ and Java implementations over versions are reflected by similar changes in the number of mutants generated for the AspectJ and Java implementations. This similarity demonstrates a rough correlation. This correlation indicates that the mutants generated for implementations are reflective of the equivalent implementations. It suggests that the mutants generated for each implementation preserve equivalence.

Comparing Distribution of Fault Types in Mutants

Figure 6.4 presents the distribution of fault types, generated in mutants at locations in one specific implementation². Each coloured segment represents the number of locations for which mutants are generated that contain a fault of a specific type.

The bars for the AspectJ and Java implementations of each version are placed directly beside one another to make comparing the distribution of fault types easier. This shows that the types of fault generated in mutants of each AspectJ and Java implementation are similar. For each pair of AspectJ and Java implementations, the distribution of shared fault types are similar.

The shared fault types are: common, Java-specific, overloading, polymorphism and traditional. The number of mutants generated for each pair are very similar. There are slightly more common, inheritance and polymorphism fault types generated in Java mutants. There are slightly more traditional fault types generated in AspectJ mutants. The number of mutants generated containing the remaining fault types are roughly equal.

The advice- and pointcut-based fault types are only generated in mutants of the AspectJ implementation. The large amount of similarities and very slight differences between

¹Size is based on metrics gathered from the HW by Greenwood et al. [63]

²A more detailed version of this chart is available in Figure A.1

6.2. ANALYSIS OF OUTCOMES AND RATES

the types of faults generated in mutants of each pair of AspectJ and Java implementations indicate that these mutants are equivalent.

6.1.3 Summary

This section presented the number of mutants that are generated for each implementation of each version of the HW. The number of mutants generated for AspectJ and Java implementations were compared. This comparison showed that there were slightly, but not significantly, more mutants generated for the AspectJ implementation. The number of mutants were correlated with the size of the implementations from which they were generated to provide evidence of mutant equivalence. The distribution of the types of faults generated in mutants for each pair of AspectJ and Java implementations were also correlated to provide further evidence of mutant equivalence.

6.2 Analysis of Outcomes and Rates

The outcomes of the test-mutant executions for each Java and AspectJ implementation are used as a basis to derive rates of *fault exposure*, *fault execution* and *infection and propagation* for those implementations. To derive these rates for each implementation, the number of *not exe*, *pass* and *fail* outcomes from the test-mutant executions are counted. The first part of this section presents and analyses the outcomes from the test-mutant executions for each implementation. The second, third and fourth parts of this section present and analyse the rates of *fault exposure*, *fault execution* and *infection and propagation* derived from these outcomes. This section is concluded by summarising the results of the analysis.

6.2.1 Outcomes

These number of *not exe*, *pass* and *fail* outcomes for the Java and AspectJ implementations of each version of the HW program are presented in Table 6.6. Figure 6.7 presents a bar chart that visualises the number and the type of outcomes for each implementation, provided in Table 6.6. Each bar in this chart represents the number of *not exe*, *pass* and *fail* outcomes of test-mutant executions for a specific implementation. The bars for the AspectJ and Java implementations of each version are placed directly beside one another to make comparing the number of outcomes easier.

This chart shows that for each implementation, the number of *not exe* outcomes is much larger than the number of *pass* and *fail* outcomes for all implementations. This is expected because each test executes one path through the mutant implementation, and given the large size and high number of potential paths through the implementation, it is not surprising that most of the test-mutant executions result in a *not exe* outcome.

Impl	Version	<i>not exe</i>	<i>pass</i>	<i>fail</i>
Java	1	788807	34182	28576
	2	797607	36583	32585
	3	800944	38612	32289
	4	802845	38118	31272
	5	813743	38500	30717
	6	822344	36052	27879
	7	813106	39516	34043
	8	812933	40165	34347
	9	1175755	48556	51664
	10	1210095	54336	50234
AspectJ	1	790743	36697	39725
	2	783007	51810	37223
	3	829170	55237	37163
	4	832999	53294	35862
	5	832999	53294	35862
	6	835886	54518	35651
	7	835536	54952	36347
	8	830536	54936	36293
	9	1175962	70468	49155
	10	1214042	77764	49359

Figure 6.6: Outcomes

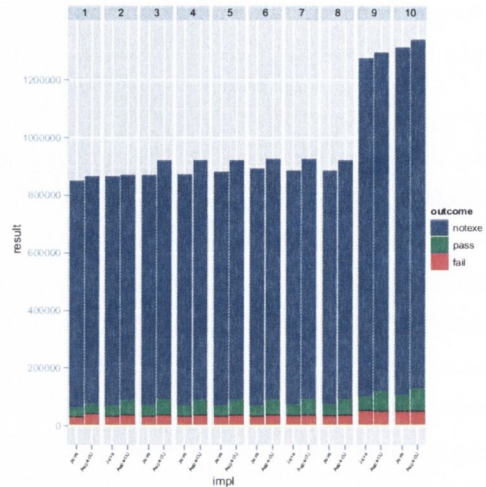


Figure 6.7: Outcomes Visualised

The chart also shows, there are consistently more outcomes for the AspectJ implementations. As shown earlier, the number of outcomes for each implementation is the product of executing the test set against each of the mutants generated for the implementation. Figure 6.3 shows that there are more mutants generated for the AspectJ implementation. The same test set is executed against the mutants generated for each pair of Java and AspectJ implementations. This means that the only determinant for the difference in the number of outcomes for Java and AspectJ implementations is the number of mutants generated from them.

6.2.2 Fault Execution

The bar chart presented in Figure 6.7, suggests that the rate of fault execution is higher for AspectJ implementations. The rate of *fault execution* is calculated as $rate = \frac{fail+pass}{fail+pass+notexe}$. This rate represents the proportion of test-mutant executions where the fault contained in the mutant is executed.

Comparing Rates of Fault Execution

Figure 6.8 shows the result of calculating the rates of *fault execution* for each Java and AspectJ implementation. In this graph, the x-axis represents the version and the y-axis represents the rate of *fault execution*. Each point on the graph represents the rate of *fault execution* for each of the AspectJ and Java implementations of each version of the HW. The points are differentiated by colour, and a line connecting the points for AspectJ and Java implementations is also provided to highlight the changes in these rates over versions of the HW.

6.2. ANALYSIS OF OUTCOMES AND RATES

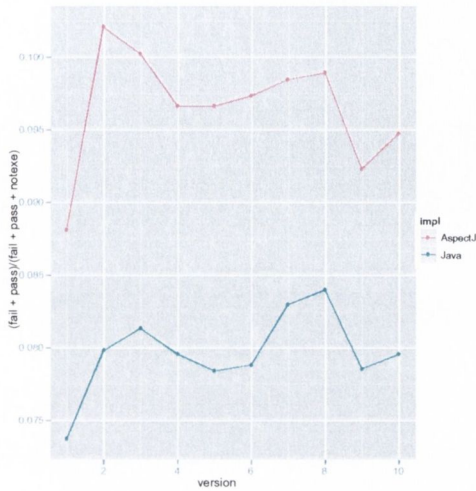


Figure 6.8: Rates of Fault Execution

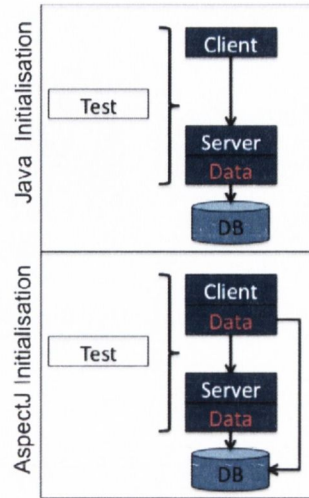


Figure 6.9: Data Layer Initialisation

This graph shows that the rate of *fault execution* is higher for the AspectJ implementations of each version. Figure 6.8 shows that the proportions of faults that are executed in mutants of AspectJ implementations are between 0.088 and 0.104. This figure also shows that for Java implementations, the proportions of faults executed is between 0.078 and 0.0804. A comparison of these rates shows that AspectJ has a higher rate of *fault execution*. This means that there are proportionally more faults executed by tests in mutants of AspectJ implementations.

As illustrated in Figure 6.1 and explained in the introduction to this chapter, the higher the rate of *fault execution*, the higher rate of *fault exposure*. This is because the more faults that are executed, the more possibilities there are for state infection and propagation of infected state. The rate of *fault execution* is higher for AspectJ implementations which means there is more potential for a higher rate of *fault exposure* for these implementations compared to Java implementations.

Cause of Higher Rates for AspectJ Implementations

Figure 6.8 shows that the rate of fault execution is higher for AspectJ. The most significant cause of this difference is that in the AspectJ implementation, there was a large amount of redundant execution every time this implementation was initialised for test execution. Figure 6.9 illustrates the initialisation of the Java and AspectJ implementations for a test.

As described in Section 5.3.3, in the initialisation of the HW, the server is started before the client. When the server is started, it initialises a data layer which acts as an interface to a database, through a persistence layer, that holds the information managed by the HW system. When the client is started it establishes a connection to the server

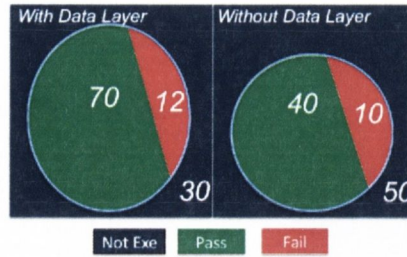


Figure 6.10: Example of the Impact Of Data Layer Initialisation On Client

and exposes a http interface to the server for managing information in the HW. This initialisation sequence for the Java implementation is illustrated at the top half of Figure 6.9.

In the initialisation of the AspectJ implementation, illustrated at the bottom half of Figure 6.9, the data layer is initialised in both the server and client. Although the data layer is initialised on the client, it is not executed within the client after initialisation. This redundant initialisation is due to a poorly defined pointcut that triggers the initialisation of this layer in the client. The intention of the pointcut is to initialise the data layer on the server only. This means that tests executed against mutants that contain faults generated in the data layer are more likely to execute these faults. The consequence of this, as observed in Figure 6.8, is a higher rate of *fault execution* for AspectJ.

The rate of fault execution is increased because more tests execute the faults present in the data layer. When the data layer is instantiated on the client during initialisation, the context of instantiation is not as expected. This causes faults to execute that would otherwise not. It also means that tests execute faults that would otherwise have not. The increase in the number of faults that are executed and the number of tests that execute faults result in an overall higher proportion of faults executed by tests per implementation.

Besides the inflation of rate of fault execution, the redundancy also has a knock on effect on the rate of *infection and propagation*. As more tests are executing more faults in the AspectJ implementation due to the redundant execution, the potential for test failure due to *infection and propagation* increases. However, this potential is not realised because the data layer is not used by the client and failures in the data layer in the client are less likely to cause test failure due to *infection and propagation*. The result of this is a deflation of the rate of *infection and propagation*. This deflation occurs because although the number of *fail* outcomes fall relative to the number of *pass* outcomes.

This inflation and deflation is explained further through the simple example presented in Figure 6.10. The numbers used in this figure are conjured for the purpose of explanation only. The figure shows the number of *not exe*, *pass* and *fail* outcomes for the client component of the AspectJ implementation with and without the data layer on the right

6.2. ANALYSIS OF OUTCOMES AND RATES

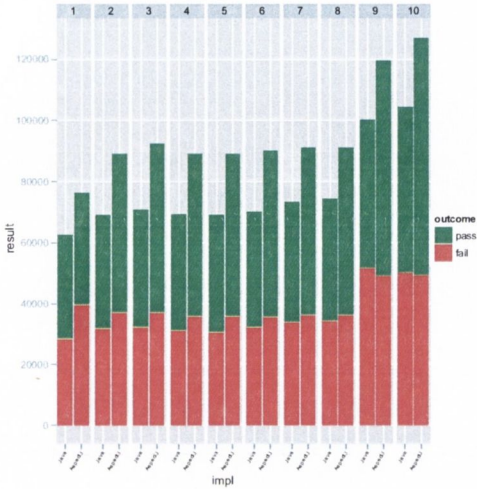


Figure 6.11: Fail and Pass Outcomes

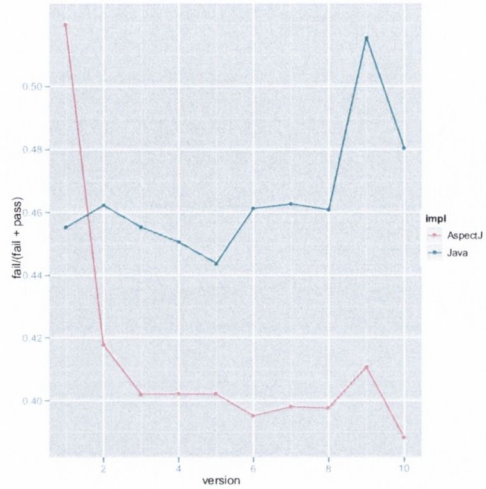


Figure 6.12: Rates of Infection and Prop.

and left hand of the figure.

This illustrates the inflation in the rate of *fault execution* from $\frac{50}{100}$ to $\frac{70}{100}$. It also demonstrates the deflation of the *infection and propagation* rate from $\frac{10}{40}$ to $\frac{12}{70}$. Moreover, it also shows that although the rate of *infection and propagation* is deflated the overall effect of the redundant data layer initialisation is the inflation of the *fault exposure* rate from $\frac{10}{100}$ to $\frac{12}{100}$. As the example shows, the increase in the number of faults executed results in a small proportion of fail outcomes which inflates the overall rate of *fault exposure*.

6.2.3 Infection and Propagation

Figure 6.11 presents a bar chart that visualises the number of *fail* and *pass* outcomes from test-mutant executions in which the fault contained in the mutant is exercised, provided in Table 6.6. Each bar in this chart represents the number of *pass* and *fail* outcomes of these test-mutant executions for a specific implementation. The bars for the AspectJ and Java implementations of each version are placed directly beside one another to make comparing the number outcomes easier.

This chart shows that for each implementation, the number of outcomes is much larger for AspectJ implementations. This is because there are more mutants that contain faults generated for the AspectJ implementations, as illustrated in Figure 6.3, and more of these faults are executed by tests, as illustrated in Figure 6.11. This chart also indicates that the rates of *infection and propagation* are generally lower for AspectJ implementations.

Rates of Infection and Propagation

The rate of *infection and propagation* is calculated as $rate = \frac{fail}{fail+pass}$. This rate represents the proportion of test-mutant executions in which the execution of the fault results in state infection and propagation, which in turn results in test failure and fault exposure.

The graph presented in Figure 6.12 shows the result of calculating the rates of *infection and propagation* for each Java and AspectJ implementation, based on the numbers of *fail* and *pass* outcomes provided in Table 6.6. In this graph, the x-axis represents the version and the y-axis represents the rate of *infection and propagation*. Each point on the graph represents the rate of *infection and propagation* for each of the AspectJ and Java implementations of each version of the HW. The points are differentiated by colour. A line connecting the points for AspectJ and Java implementations is also provided to highlight the changes in these rates over versions of the HW.

Figure 6.12 shows that for AspectJ implementations, the rate of failure declines from an initial high of 0.52 to a low of 0.39. This indicates that over maintenance activities the rate of *infection and propagation* for AspectJ decreases. This figure also shows that for Java implementations, the rate of failure improves from an initial rate of 0.455 to a final rate of 0.048. This indicates that over maintenance activities the rate of *infection and propagation* increases for Java. A comparison of these rates shows that overall, Java has a higher rate of failure outcomes for the mutants executed.

As illustrated in Figure 6.1 and explained in the introduction to this chapter, the higher the rate of *infection and propagation*, the higher rate of *fault exposure*. This is because the more test failures, the more faults that are exposed and the higher the rate of *fault exposure*. The rate of *infection and propagation* is higher for Java implementations which means there are more faults exposed when tests execute faults in mutants for these implementations compared to AspectJ implementations.

Causes of Higher Rates for Java Implementations

The graph presented in Figure 6.13 decomposes the number of *fail* and *pass* outcomes presented in Figure 6.11 for each implementation by fault type. This shows that different fault types influence the rate of *infection and propagation* for each implementation. The figure contains a bar chart for each fault type. Each chart directly compares the number of pass and fail outcomes of test-mutant executions for Java and AspectJ implementations of each version of the HW.

The reason the rates of *infection and propagation* are higher for Java is because the rates of *infection and propagation* for fault types are generally higher for Java compared to AspectJ. The rates of *infection and propagation* for the common, inheritance, overloading and traditional fault types are higher for Java implementations compared to AspectJ. The rates of *infection and propagation* for these fault types are highly influential on the

6.2. ANALYSIS OF OUTCOMES AND RATES



Figure 6.13: Fault Types

overall rates of *infection and propagation* for Java and AspectJ implementations. This is because the number of outcomes for these types account for a large proportion of overall number of outcomes for the Java and AspectJ implementations. The rates of *infection and propagation* for these fault types pulls the overall rates of *infection and propagation* for Java implementations over those for AspectJ implementations.

The rates of *infection and propagation* for the advice and pointcut fault types are low, which pulls the overall rates of *infection and propagation* for AspectJ implementations down further compared to Java implementations. The rates of *infection and propagation* for the java-specific and polymorphism fault types are higher for AspectJ implementations compared to Java. The rates of *infection and propagation* for these fault types is not as influential on the overall rates of *infection and propagation* for Java and AspectJ implementations. This is because the number of outcomes for these types account for a smaller proportion of overall number of outcomes for the Java and AspectJ implementa-

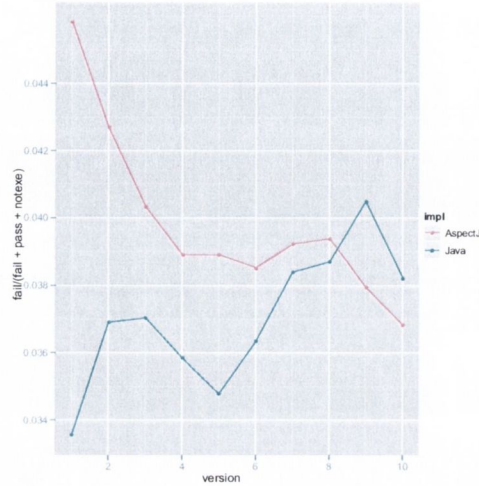


Figure 6.14: Rates of Fault Exposure

tions. The rates of *infection and propagation* for these fault types reduces the difference between rates of *infection and propagation* for the Java and AspectJ implementations.

6.2.4 Fault Exposure

The rate of *fault exposure* is calculated as $rate = \frac{fail}{fail+pass+notexe}$. This rate represents the overall proportion of test-mutant executions that result in fault execution, state infection and propagation, which in turn results in test failure and fault exposure.

Comparing Rates of Fault Exposure

Figure 6.14 presents a graph that illustrates the difference between the rates of fault exposure for the AspectJ and Java implementations of each version of the HW. In this graph, the x-axis represents the version and the y-axis represents the *fault exposure*. Each point on the graph represents the rate of *fault exposure* for each of the AspectJ and Java implementations of each version of the HW. The points for AspectJ and Java implementations are differentiated by colour. A line connecting the points for AspectJ and Java implementations is also provided to highlight the changes in these rates over versions of the HW.

The graph presented in Figure 6.14 shows that over the versions the rate at which faults are exposed by executing mutants is generally higher for the AspectJ implementations. The rate of fault exposure for the AspectJ implementation of the initial version of the HW is much higher at 0.046 than the Java implementation at 0.033. The rate of *fault exposure* for AspectJ drops up to version six of the HW, where the rate recovers slightly

6.2. ANALYSIS OF OUTCOMES AND RATES

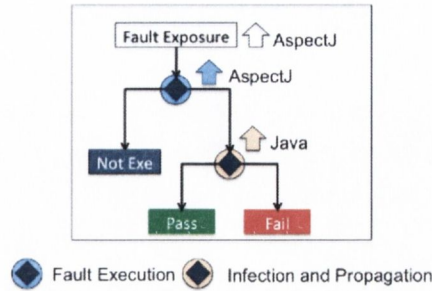


Figure 6.15: Comparative Effects

until version eight, where the rate drops again to a final rate of 0.038.

The overall rate of fault exposure is higher for the AspectJ compared to Java implementations. However, it does seem that the rates are converging in the later versions of the HW.

Causes of Higher Rates for AspectJ Implementations

Figure 6.15 illustrates the comparative effects of AspectJ and Java on the rates of *fault exposure*, *fault execution* and *infection and propagation*. It shows that the general effect of AspectJ is to improve the rate of *fault exposure* over Java. This means that the proportion of faults exposed from executing the test set against the mutants generated for AspectJ implementations is higher than the proportion of faults exposed from executing the test set against the mutants generated for Java implementations.

The figure also shows that the general effect of AspectJ is to improve the rate of *fault execution* over Java. This means that the proportion of faults executed in mutants of the AspectJ implementations is higher than the proportion of faults executed in mutants of the Java implementations. It also shows that the general effect of Java is to improve the rate of *infection and propagation* over AspectJ. This means that the proportion of faults exposed, when faults in mutants of the Java implementations are executed, is higher than the proportion of faults exposed when faults in mutants of the AspectJ implementations are executed.

As mentioned earlier and illustrated again in Figure 6.15, there is a causal effect between the rates of *fault execution* and *infection and propagation* on the rates of *fault exposure*. The rate of *fault exposure* is directly caused by the rates of *fault execution* and *infection and propagation*. The higher the rate of *fault exposure*, the more faults that are executed. The more faults that are executed, the more possibilities there are for state infection and propagation of infected state. The higher the rate of *infection and propagation*, the more fail outcomes. The more fail outcomes, the higher the rate of *fault exposure*.

These results suggest that although a smaller proportion of executed faults result in state infection and propagation of the infected state causing test failure and fault exposure, there are proportionally more faults exposed in the AspectJ implementations. This is because there are more faults executed in mutants of the AspectJ implementation and, although there are proportionally fewer faults exposed compared to Java implementations, the increased volume of faults executed results in a higher number of instances of state infection and propagation resulting in test failure and fault exposure for AspectJ implementations.

6.2.5 Summary

This section presented the number of *not exe*, *pass* and *fail* outcomes for the Java and AspectJ implementations of each version of the HW program. These outcomes are used to derive rates of *fault exposure*, *fault execution* and *infection and propagation* for the AspectJ and Java implementations. The rates for the AspectJ and Java implementations are compared. The results show that the rates of *fault exposure* are higher for AspectJ implementations. The results also show that the rates of *fault execution* are higher for AspectJ implementations but the rates of *infection and propagation* are higher for Java implementations. The analysis of these rates suggests that the reason for this is, that proportionally more faults in mutants of the AspectJ implementation are executed, causing more state infection and propagation, which in turn results in test failure and fault exposure.

6.3 Quantifying the Comparative Effects

The analysis of the effects of Java and AspectJ on the rates of *fault exposure*, *fault execution* and *infection and propagation* in the previous section shows that there is a difference between the effects of Java and AspectJ. This analysis does not however quantify the comparative effects. This section applies binomial regression analysis to the outcomes presented in Table 6.6. This approach supports the quantification of the comparative effects of Java and AspectJ on these rates. The first part of this section explains the application of binomial regression, while the second part presents the measures of the comparative effects of Java and AspectJ on each rate. The third and final part summarises this section.

6.3.1 Binomial Regression

Binomial Regression Analysis (BRA) [52] is used to quantify the difference in the effects of Java and AspectJ on the rates of *fault exposure*, *fault execution* and *infection and propagation*. BRA is a statistical technique for analysing the relationship between a

6.3. QUANTIFYING THE COMPARATIVE EFFECTS

model	specification
1	$Fault\ Exposure \sim implementation + version$
2	$Fault\ eXecution \sim implementation + version$
3	$Infection\ and\ Propagation \sim implementation + version$

Table 6.1: Models

binomial response, (i.e., *pass* or *fail*) and explanatory factors. In this application of binomial regression, the binomial response is the rate and the explanatory factors are the implementation and maintenance version. This relationship is captured in the binomial regression model, $rate \sim implementation + version$, which indicates that each rate is explained by both the AspectJ and Java *implementation* approaches and *version* of the program. This is the standard way in which a binomial regression model is specified [52].

Model Fitting

The relationship between each rate, and the implementation and version factors in each model, is measured by fitting the model over the measures in Table 6.6. In the model fitting process, the correlation between the effects of the levels of each factor on the observed rate is measured [52]. These measurements reflect the strength of the correlation between the rates (of *fault exposure*, *fault execution* and *infection and propagation*) and levels 1-10 of the version factor and the Java and AspectJ levels of the implementation factors. These correlations are used to measure the generalised effects of each factor on the rate of fault exposure [52].

Table 6.6 presents the results of fitting three models for the effects of the *Version* and *Implementation* factors on the rates of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. These models are presented in Table 6.1. As will be shown next, the measures resulting from the model fitting process are the basis for calculating the comparative effects of Java and AspectJ on the rates of *fault exposure*, *fault execution* and *infection and propagation*.

6.3.2 Comparative Effects

As detailed by Faraway [52], the measures of the effects presented in Table 6.16 are used to construct the graphs of the generalised effects of AspectJ and Java on the rates of *fault exposure*, *fault execution* and *infection and propagation*, presented in Figures 6.17, 6.18 and 6.19, respectively. In each of these figures, the difference between the Java and AspectJ lines is the measure of the comparative effect of AspectJ and Java. The measures of the difference between the AspectJ and Java lines is marked in red in Table 6.16. These are the measures of the comparative effect of AspectJ and Java on each rate. These measures are on the log odds scale [52] and need to be transformed by taking the

6.3. QUANTIFYING THE COMPARATIVE EFFECTS

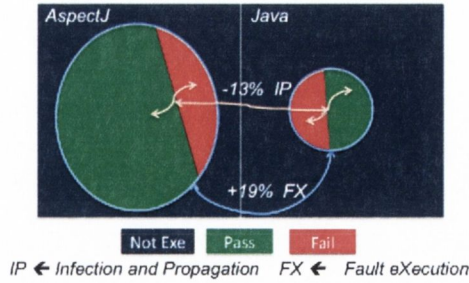


Figure 6.21: Causation

estimated to account for a significant amount of the maintenance cost [33, 22, 141] and for those considering the adoption of AOP to reduce maintenance costs [39], this evidence is encouraging.

Fault exposure is a direct consequence of *fault execution* and state *infection and propagation*. If more faults are executed, then there are more chances for state infection and propagation, resulting in fault exposure. The more executed faults that cause state infection and propagation, the more faults that are exposed. If more faults are exposed, then the odds of *fault exposure* increase.

Figure 6.20 shows that the odds of *fault execution* are 19% ($0.8114715 = \exp(-0.208906)$) higher for AspectJ and that odds of *infection and propagation* are 13% ($1.138142 = \exp(0.129397)$) lower for AspectJ. This means that in the AspectJ implementations there are more faults executed. However, it also means that compared to Java implementations, proportionally less of the executed faults cause state *infection and propagation*, resulting in lower odds of fault exposure.

This is explained further through the illustration in Figure 6.21. The boxes marked AspectJ and Java represent the total number of test-mutant executions for AspectJ and Java implementations, respectively. The circles in these boxes represent the number of faults executed by tests in AspectJ and Java mutants. This representation shows that there are more faults executed in AspectJ compared to Java mutants. This difference is the cause of the 19% higher odds of *fault execution* for AspectJ. The number of executed faults that result in pass and fails are represented inside the circle. This representation shows that there are proportionally fewer fails for AspectJ, indicating that less of the faults executed in AspectJ mutants result in infection and propagation. This difference is the cause of the 13% lower odds of *infection and propagation* for AspectJ.

6.3.3 Summary

In this section, binomial regression analysis was applied to the outcomes presented in Table 6.6. Models that represent the effects of Java and AspectJ implementations and

each version on the rates were fitted to the outcomes. The result of this fitting process are relative measures of the effects of Java and AspectJ implementations and each version on the rates. Based on these measures, the comparative effect of Java and AspectJ on each rate was quantified.

6.4 Threats to Validity

The goal of this thesis is to gather empirical evidence of the comparative effect of AOP and OOP on testability through a study. In the study, the testability of equivalent AspectJ and Java implementations are measured for each maintenance version of the Health Watcher program. The result of which are pairs of testability measures that represent the effects of AspectJ and Java on testability, one pair for each version of the program. In this chapter these measures are presented and analysed to identify and quantify the comparative effect of Aspect and Java on testability.

Chapters 4 and 5 detail the decisions taken to form a methodology and select inputs to ensure a valid result. The methodology defines testability and outlines how it is measured, using mutation analysis, as the rate of fault exposure. It describes how mutation analysis was applied to ensure that all factors that may affect the measures are fixed to ensure that the factors of interest, implementation approach and version, are isolated for study and defines how the resulting measures are analysed to understand and quantify the comparative effect of AOP and OOP on testability.

In the study, mutation analysis is applied to AspectJ and Java implementations of maintenance version of the HW program. In each application, mutants of these implementations are generated. A set of tests is executed against these mutants to provide empirical evidence of the comparative effect of AspectJ and Java on testability in this context. The AspectJ and Java implementations of the HW program, the mutants and the test set were selected because they created a context that was highly representative of general case. The more representative the context is of the general case the more generalisable the results of the study are.

This section discusses the decisions taken in the formation of the methodology and selection of inputs for the study. It analyses these decisions and identifies potential threats to the validity that arise from them. Where threats are identified their impact on how the results can be interpreted are discussed. In particular, the chapter analyses the decision relating to the selection of an approach to measure testability, the decision to focus on the HW program and the selection of the test set used in the application of mutation analysis to each implementation.

6.4. THREATS TO VALIDITY

6.4.1 Testability Measurement

In this thesis, the testability of an implementation is defined as the ease with which faults can be exposed through testing [148]. As identified in Chapter 4, the ease with which faults can be exposed through testing can be measured from two alternate perspectives, *implementation* and *test*. The *implementation* perspective measures testability as the proportion of mutants, generated from the implementation, exposed by a set of tests. The *test* perspective measures the effort needed to create tests that will expose the faults in the mutants. The measurements in this thesis are taken from the implementation perspective. In this section, the threat of this selection on the construct validity of the results is discussed.

From the *implementation* perspective, there are two different approaches to measuring testability. As outlined in Chapter 4, the first is based solely on the the proportion of faults that are exposed and the second is based on the proportion of test-fault executions that led to exposure. The second approach was used to measure testability. The impact of this selection on the construct validity of the results is also discussed in this section.

Perspective Selection

The construct validity of the result is dependent on how well the measurement approach captures the property being measured. The decision to measure testability from the *implementation* perspective over the *test* perspective is a threat to construct validity and has an impact on the way in which the results can be interpreted.

The ease with which faults can be exposed through testing is dependent on the interaction of two elements - tests and the implementation they are applied to. The *test* perspective measures the effort to create tests that will expose faults and the *implementation* perspective measures the ease with which the implementation exposes faults. To fully understand and quantify how testable an approach to software implementation is both perspectives should be considered.

The measurement approach used in this study focuses solely on the differences in fault exposure between AOP and OOP implementations of a program. It does not address the question of whether it is easier or more difficult to create tests for these implementations. The impact of this is that only one perspective of the testability property is captured in the measures on which the results presented in this chapter. This means that the interpretation of the result is limited in that it only identifies and quantifies the comparative effect of AspectJ and Java on the ease with which faults present in an implementation can be exposed. It does not provide an indication of the comparative effect of AspectJ and Java on the ease of creating tests that will expose faults. This is a limitation of this study.



Figure 6.22: Measuring Testability In A Comparative Context

Approach Selection

In most studies that measure testability from the *implementation* perspective measure testability as the proportion of faults that are exposed through testing [31, 116, 114]. This study measures testability in terms of the proportion of test-fault executions that result in exposure. To assure readers that this deviance from the typical approach is not a threat on the construct validity a simple example is used to clarify and justify the approach taken. The simple example illustrated in Figure 6.22.

In this example one set of four tests are executed against two implementations (1 and 2) of a program, similar to this study. Both implementations contain the same set of five faults. If the fault based measurement approach is used then the testability of both implementations (1 and 2) is $\frac{4}{5}$, because 4 of the 5 faults are exposed in each respective implementation. If the test-fault execution based approach to measurement is used, then the testability of implementation 1 is $\frac{4}{20}$ and the testability of implementation 2 is $\frac{16}{20}$. This example shows that the test-fault based approach is a more detailed approach to measurement and can be more informative in a comparative context.

Because the approach used provides a more detailed measurement and is more appropriate in a comparative context, the impact is not to threaten validity but to make the measurements more representative for the comparative context in which they are used, bolstering construct validity.

6.4.2 Program Selection

Chapter 5 identifies external validity or generalisability as a key challenge for the study. As outlined in Chapter 5, the approach taken to address this challenge was to identify candidate programs that fitted the methodology detailed in Chapter 4 as a basis for the study and select the candidate that was most representative of the general case. This subsection identifies the threats that arise from this selection and the impact of these

6.4. THREATS TO VALIDITY

threats on the way in which the results of the study can be interpreted.

HW Selection

Chapter 4 identifies then HW program as the most representative of the general case of the candidates available for study. As the most representative candidate, the HW program was considered the selection that would maximise generalisability.

As identified previously in Section 6.2, the AspectJ implementations it was observed that there was a significant amount of redundant execution for each test while the program was being initialised. This was caused by a poorly defined pointcut that triggered the initialisation of the data layer in the client component of the Health Watcher which is redundant. This redundant has an affect on the measures of testability for AspectJ implementations. The redundant execution inflates the number of faults executed in AspectJ mutants, inflating the odds of *fault execution*, deflating the odds of *infection and propagation* and inflating the overall odds of *fault exposure*.

To quantify the impact of this redundant execution, traces driven by tests through the AspectJ implementations were obtained. These traces enabled the identification of the additional mutants that were executed by the tests during mutation analysis. Although difficult to accurately quantify, analysis of these mutants indicated that the impact on the odds of *fault exposure*, *fault execution* and *infection and propagation* was to reduce the differences or comparative effects of AspectJ and Java to a point where they were negligible.

There are two ways the effects of redundant execution can be viewed. The first is that it is representative of the types of issues that occur when AOP is adopted in practice. The second is that the redundant execution is not representative of the types of issues that occur if AOP is adopted. Because AOP has not been widely deployed, it is difficult to know if the occurrence of pointcut issues, resulting side effects such as redundant execution, are a characteristic of AOP or not. This is because there is no empirical evidence to confirm or deny that these issues are are a characteristic of AOP in practice. Dependent on the point of view the reader prescribes to, the affect on validity can be argued from either perspective.

Sample Size of One

Basing the study solely on the HW program threatens the degree to which the results can be generalised because the results are specific to the HW. If more of the candidates identified in Chapter 5 were selected to increase the sample size the results would be less specific to the HW.

The reason that more programs were not selected for study was base on cost. The cost of including the program in the study include a setup cost and a computation cost. The

setup cost is the cost of generating mutants for the implementations of the program and generating realistic test cases to execute against those mutants. The computational cost is the computational cost of this execution, which is well known to be highly expensive [31, 116, 114].

There was a fixed amount of resources for setup and computational resources available. These resource were consumed by the inclusion of the HW program into the study. This meant it was not possible to increase the sample size and make the results less HW centric. This means that it is difficult to confidently generalise the identified comparative effect of AOP and OOP on testability. This is a limitation to the external validity of this study.

6.4.3 Test Selection

As outlined in Chapters 4, 5 and this chapter, after selecting the Health Watcher program as the basis for the study, mutants were generated for the implementations of the different maintenance versions of the program and tests were selected for use in mutation analysis. To maximise the generalisability of the results a set of tests were selected that were representative of the general case. There are two potential threats to validity that are based on the test selection. The first is based on how effects of the oracles used in the evaluation of the outcome of the test-fault executions. The second is based on the using a test set sample size of one. These potential threat are explained, their impact is outlined and their existence is verified.

Oracles

A test is made up of an input and an oracle or expected outcome. If a fault is executed by a test there are two factors have an influence on exposure. The first factor, as detailed in Chapter 3, is the structure of the program that lets the test case observe a failure. The second factor is the ability of the oracle to identify this failure. If the oracle factor was able to identify test failures in the Java implementation but not the AspectJ implementation then this would introduce an uncontrolled factor into the study that would unbalance the comparison, threaten the internal validity of the identified and quantified comparative effect.

As explained in Chapter 5, the tests were developed based on the use cases used as the basis for the Java and AspectJ implementations. The oracles were developed based on sample data used in the test creation process. The test and in particular oracle selection process was completely independent from the processes of Java and AspectJ implementation. This ensured there could be not oracle selection bias toward either implementation. Furthermore, both implementations expose precisely the same interface. This means that both implementations must produce precisely the same output given an

6.4. THREATS TO VALIDITY

Test Set Size	FE		FX		IP	
	min	max	min	max	min	max
50	0.0084	0.1209	0.1499	0.2227	-0.2331	-0.0557
100	0.0211	0.1093	0.1645	0.2048	-0.1979	-0.0724
150	0.0311	0.0938	0.1733	0.2048	-0.1864	-0.0961
200	0.0443	0.0950	0.1716	0.2042	-0.1830	-0.0971
250	0.0463	0.0901	0.1773	0.2060	-0.1802	-0.10511
300	0.0460	0.0869	0.1795	0.2010	-0.1689	-0.1068
350	0.0522	0.0887	0.1823	0.2034	-0.1596	-0.1090
400	0.0495	0.0847	0.1827	0.1991	-0.1569	-0.1093

Table 6.2: Max and Min results for rates for randomly selected test sizes

input. The fact that there is no dependence or bias of test oracles and implementation this means that the oracle factor is not a threat to internal validity.

Test Set Sample Size of One

Chapter 5 outlines that the test set used in the measurement phase of the study was developed by a group of seven software testing professionals, all from different Irish software development companies, with a minimum of four years of industrial experience in this group. The group used a best practice, use case driven test selection process. The process and professionals were used to ensure that the resulting test set was highly representative of the complexity encountered in the general case.

The group selected the test set based on the use cases and related design documentation on which the implementation was based. The result of the selection process was a set of 265 tests. This set was then used in the application of mutation analysis to the AspectJ and Java implementations of the program.

This means that the results of the study are based on one set of tests. Basing the study on one set of tests is a threat to the external validity or generalisability of the results. This is because the use of a different set of tests may have yielded measure of the comparative effect of AspectJ and Java on the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. To investigate this threat, the analysis presented in this chapter was performed on different subsets of the selected tests. Table 6.2 presents the max and min results for each rate based on analysing 1000 randomly selected tests, taken from the original set of 265 tests, in groups of 50, 100,150, 200, 250, 300, 350, 400.

The max and min represent the most extreme values from performing the analysis on these new subsets. The most extreme values are of most interest because they are can be used as measures of the smallest and largest observed difference in the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. The difference is small if it is close to 0 and large the further from 0 it is. The difference between the max and min represent the interval between the smallest and largest difference observed from applying the analysis to the 1000 randomly selected sets of tests.

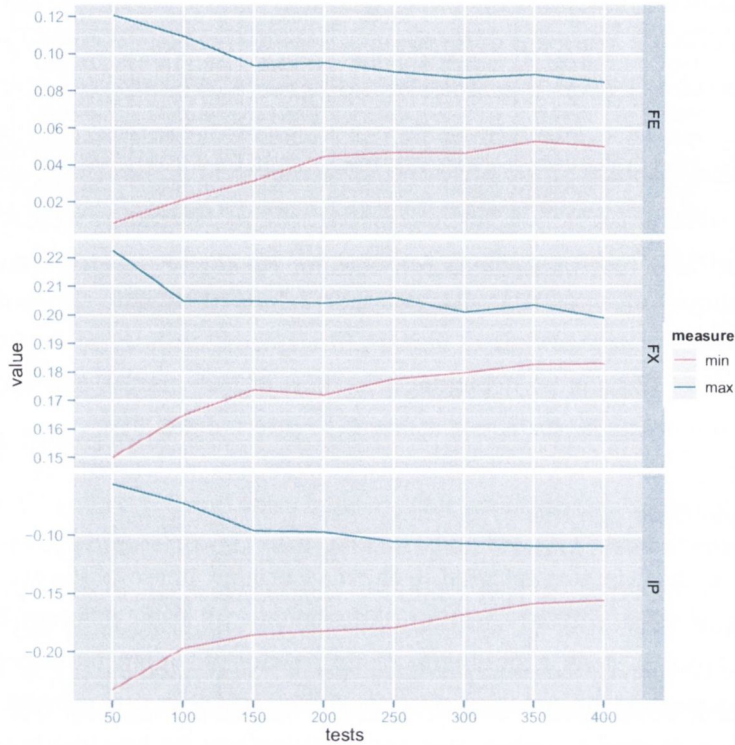


Figure 6.23: Convergence as test set size increases

Figure 6.23 illustrates the max and min measures of the comparative effects of AspectJ and Java on the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation*. It shows that for the smaller test sets (< 200) the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation* can be small or large. There is a relatively wide interval between the max and min for each which suggests that based on these test sets the observed difference could be negligible or significant.

For the larger test sets (≥ 200) the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation* become more consistent. The odds of *Fault Exposure* and *Fault eXecution* are consistently higher for AspectJ and the odds of *Infection and Propagation* are consistently lower for AspectJ. As the number of tests in the randomly selected test set increases there seems to be a convergence toward a narrower interval.

From Figure 6.23 three conclusions can be drawn. The first is that at in the worst case scenario (smaller subsets of test) for AspectJ, the difference between the odds of *Fault Exposure*, *Fault eXecution* and *Infection and Propagation* is negligible. The second is that at in the best case scenario (smaller subsets of test) for AspectJ, the difference between the odds of is significant.

6.5. CHAPTER SUMMARY

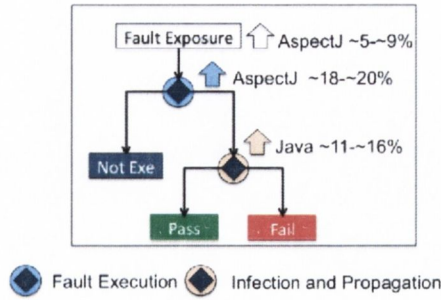


Figure 6.24: Intervals for the difference between AspectJ and Java

The third and final conclusion that can be drawn is that as the number of tests increases, a consistent difference between the odds emerges. At the test set size of 400 the difference, as illustrated in Figure 6.24, between AspectJ and Java is that AspectJ consistently results in an improvement in the odds of *Fault Exposure* by between approximately 5 and 9%. This is caused by an increase in the odds of *Fault eXecution* of between 18 and 20% and a decrease in the odds of *Infection and Propagation* by between 11 and 16%.

These conclusions impact the interpretation of the results of the study in that they reduce the threat to the external validity or generalisability. By performing the analysis on 1000 randomly selected tests, taken from the original set of 265 tests, in groups of 50, 100, 150, 200, 250, 300, 350, 400 more confidence can be associated with the general conclusion that for the HW program there is a moderate increase in the odds of *Fault Exposure* of between 5 and 9%.

6.4.4 Summary

This section discussed the decisions taken in the formation of the methodology and selection of inputs for the study. Specifically the decisions to focus on a specific perspective on testability, on the HW program and a specific set of tests are discussed. In each of these discussions, the potential threats to validity can emerge from these decisions are identified. The existence of these threats are analysed and the impact of these threats on the validity of the results is examined.

6.5 Chapter Summary

The first section presented the number of mutants that are generated by this tool for each implementation of each version of the HW. The number of mutants generated for AspectJ and Java implementations were compared. This comparison showed that there were slightly, but not significantly, more mutants generated for the AspectJ implementa-

tion. The number of mutants were correlated with the size of the implementations from which they were generated to provide evidence of mutant equivalence. The distribution of the types of faults generated in mutants for each pair of AspectJ and Java implementations were also correlated to provide further evidence of mutant equivalence.

This second section presented the number of *not exe*, *pass* and *fail* outcomes for the Java and AspectJ implementations of each version of the HW program. These outcomes are used to derive rates of *fault exposure*, *fault execution* and *infection and propagation* for the AspectJ and Java implementations. The rates for the AspectJ and Java implementations are compared. The results show that the rates of *fault exposure* are higher for AspectJ implementations. The results also show that the rates of *fault execution* are higher for AspectJ implementations but the rates of *infection and propagation* are higher for Java implementations. The analysis of these rates indicates that the reason for cause of this is that proportionally more faults in mutants of the AspectJ implementation are executed, causing state infection and propagation resulting in test failure and fault exposure.

In the third section, binomial regression was applied to the outcomes presented in Table 6.6. Models that represent the effects of Java and AspectJ implementations and each version on the rates were fitted to the outcomes. The result of this fitting process are relative measures of the effects of Java and AspectJ implementations and each version on the rates. Based on these measures, the comparative effect of Java and AspectJ on each rate was quantified. The next chapter draws conclusions from these findings.

In the fourth and final section, the decisions taken in the formation of the methodology and selection of inputs are reviewed to identify and discuss threats to validity that they have introduced. The decisions are reviewed in terms of their impact on construct, internal and external validity. Where threats are identified, their impact on the way in which the results can be interpreted are outlined.

Chapter 7

Conclusions and Future Work

Proponents of AOP have claimed that it reduces the cost of maintenance compared to OOP by improving maintainability [84, 83, 50, 68, 125, 89, 40]. This claim has led organisations using OOP to consider adopting AOP [39, 2].

Testability, analysability, changeability and stability are key indicators of maintainability [73]. Existing studies [149, 14, 92, 97, 54, 87, 63] contribute empirical evidence of the effects of AOP and OOP on analysability, changeability and stability, but not testability. The lack of evidence of the effects of AOP and OOP on testability represents an evidential gap illustrated in Figure 7.1.

The existing empirical evidence indicates that AOP improves analysability, changeability and stability over OOP. This is encouraging for those considering the adoption of AOP. Testability is an key component of maintainability [33, 22, 141]. Without evidence of the comparative effect of AOP and OOP on testability, the maintainability claim cannot be fully tested and the adoption of AOP cannot be objectively considered [39, 129].

This thesis gathers empirical evidence of the comparative effect of AOP and OOP on testability through a study. This study is conducted in two phases. In the first phase, the testability of equivalent AspectJ and Java implementations are measured for each maintenance version of the Health Watcher program. The result of this phase are pairs

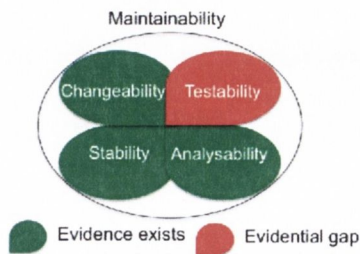


Figure 7.1: Gap

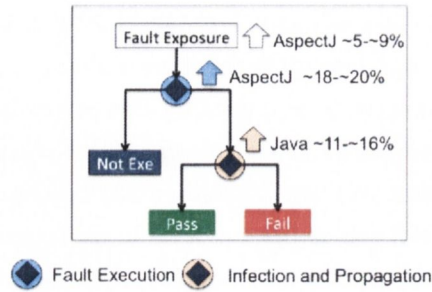


Figure 7.2: Comparative Effects

of testability measures that represent the effects of AspectJ and Java on testability, one pair for each version of the program. In the second phase, these measures are analysed to identify the comparative effect of Aspect and Java on testability.

The first section of this chapter describes what the evidence contributed by this study means for those who are considering the adoption of AOP. The second section outlines further research that is needed to further broaden and strengthen this evidence.

7.1 Conclusions

This section describes what the evidence means for those who are considering the adoption of AOP. Based on the results of the study, advice is offered to those who do adopt AOP and those who want to ensure a high level of testability.

7.1.1 Comparative Effect of AOP and OOP on Testability

The primary contribution of the study presented in this thesis is evidence to indicate that the effect of AOP is to increase testability over OOP. The results of the study are illustrated in Figure 7.2. This figure shows that the odds of *fault exposure* are between 5 and 9% higher for the AspectJ implementations of the health watcher program. This means that, for the health watcher program, faults are easier to expose in AspectJ compared to Java implementations.

Although this evidence is difficult to generalise from because it is derived from a study of one program, it does provide some evidence to indicate that the effect of AOP may be to increase testability over OOP. Testability can have a significant effect on maintenance costs [33, 22, 141] and for those considering the adoption of AOP to reduce maintenance costs [39], this evidence is encouraging.

As outlined in Chapter 2, the existing empirical evidence indicates that AOP improves analysability, changeability and stability over OOP. This evidence indicates that AOP improves testability over OOP, subject to the caveats raised by the fact that the Health

7.1. CONCLUSIONS

Watcher's behaviour may be influenced by the redundant execution issue presented in Section 6.4. This means that for all key indicators of maintainability, there is evidence that AOP is beneficial. This body of evidence enables the adoption of AOP to be more objectively considered.

7.1.2 Causes of Comparative Effect

A secondary contribution of the study, also presented in Figure 7.2, is to identify the causes of the 5 to 9% difference in the effects of AspectJ and Java on the odds of *fault exposure*. *Fault exposure* is a direct consequence of *fault execution* and state *infection and propagation*. If more faults are executed, then there are more chances for state infection and propagation, resulting in fault exposure. The more executed faults that cause state infection and propagation, the more faults that are exposed. If more faults are exposed, then the odds of *fault exposure* increase.

Figure 7.2 shows that the odds of *fault execution* are between 18 and 20% higher in the AspectJ implementations and that the odds of state *infection and propagation* are between 11 and 16% lower in the AspectJ implementations. This means that in the AspectJ implementations there are more faults executed. However, it also means that compared to Java implementations, proportionally less of the executed faults cause state *infection and propagation*, resulting in lower odds of fault exposure.

This is explained further through the illustration in Figure 7.3. The boxes marked AspectJ and Java represent the total number of test-mutant executions for AspectJ and Java implementations, respectively. The circles in these boxes represent the number of faults executed by tests in AspectJ and Java mutants. This representation shows that there are more faults executed in AspectJ compared to Java mutants. This difference is the cause of the 18-20% higher odds of *fault execution* for AspectJ. The number of executed faults that result in pass and fails are represented inside the circle. This representation shows that there are proportionally less fails for AspectJ, indicating that less of the faults executed in AspectJ mutants result in infection and propagation. This difference is the cause of the 11-16% lower odds of *infection and propagation* for AspectJ.

Figure 7.3, indicates that even though there is proportionally less fail to pass outcomes from AspectJ test-mutant executions, the odds of fault exposure is between 5 and 9% higher because the volume of fail outcomes is higher for AspectJ. The volume is higher because the number of faults executed in mutants (or pass and fail outcomes) is higher for AspectJ compared to Java. Again this evidence of causation is difficult to generalise from because it is derived solely from the health watcher program.

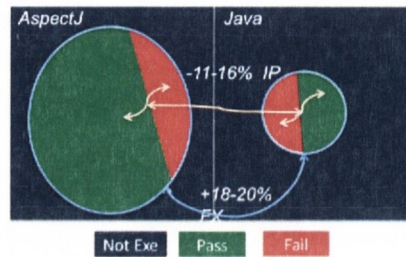


Figure 7.3: Reason for Comparative Effects

7.1.3 Advice for Adoption of AOP

One of the dangers of adopting AOP is the issues with pointcuts. This study is based on implementations of the Health Watcher program, which has been the basis of three studies that provide evidence to indicate that AOP implementations are more stable than OOP implementations. Until this study, the poorly defined pointcut had not been detected. This is because the subtle symptoms of the issue only become apparent after detailed analysis. This shows that issues related to pointcuts can be difficult to identify and can go unnoticed.

These pointcut related issues have been identified as a problems that are likely to occur in practice [43, 142, 80, 139]. Based on this observation, the advice offered to those adopting AOP is to be careful to ensure that issues related to pointcuts are detected. There are a number of approaches that have been proposed to address these issues [43, 80, 139]. These include a test driven approach to pointcut development that is designed to identify pointcut related issues early [43] and a model based approach to manage pointcut evolution [80, 139].

7.1.4 Issues to Consider when Adopting AOP

A study that aims to provide evidence about maintainability of AOP vs. OOP is not relevant without considering more global issues of engineering software using Aspect- and Object-Oriented approaches. To correctly build AO software requires a complete revision of the software development process. The adoption of AOP to achieve more maintainable software requires changes to requirements gathering phase, the architectural and design phases and the implementation phase of the process [131]. It also requires changes to how software is tested [69, 152]. These changes require more work to identify crosscutting concerns in requirements, model them architecturally, design software to modularise them and implement these designs. Once software is implemented testing the resulting aspects and the implementations into which they are woven becomes more complex [69, 152].

This means that those who are considering the adoption of AOP over OOP must be

7.2. FUTURE WORK

aware of the investment required to adopt AOP. The perspective adopter must consider the trade-off between the required investment needed to adopt AOP against the potential savings in maintainability costs.

7.2 Future Work

This study gathers evidence of the comparative effect of AOP and OOP on testability. Further studies are needed to broaden this evidence, to provide a detailed understanding of the effect of AOP and OOP on testability across different conditions.

7.2.1 Pointcut Issues

As mentioned in Section 7.1.3, one of the dangers of adopting AOP is the issues with pointcuts. Pointcut related issues have been identified as a problems that are likely to occur in practice [142, 80, 139, 43]. However, there is no empirical evidence of the frequency with which these issues arise in practice. There is also no evidence of the effects that these issues have on the cost of testing and maintenance.

Further studies are needed to gather evidence of the frequency with which pointcut related issues arise in practice and the effects that these issues have on the cost of maintenance. The detail from this study would provide more contextual evidence, that would make the results of studies, such as the study presented in this thesis, easier to interpret.

7.2.2 Causation of Lower Infection and Propagation Odds for AOP

The results of the study provide evidence to indicate that the effect of AOP is to reduce the odds of infection and propagation. Figure 6.13 shows that some types of faults are harder to expose in an AOP implementation while others are easier. It also shows that AOP specific faults tend to have a low rate of exposure meaning they are hard to expose through testing.

The results indicate the reason the odds of infection and propagation are lower for AOP implementation is because there are more faults of types that have a lower rate of fault exposure in AOP implementations, than those that have a higher rate of fault exposure. Further studies are needed to understand why some types of faults are harder to expose in AOP compared to OOP implementations and why others are easier to expose. By understanding the reasons behind these differences, guideline can be derived to help software engineers using AOP and/or OOP to make their implementations more testable.

7.2.3 Testing

The evidence gathered in this study is based on applying system-level functional tests to AOP and OOP implementations over maintenance activities. In an industrial setting, there are various testing conditions that are used to validate the correctness of implementations. For instance, integration, module and unit level testing and different types of testing such as non-functional testing are used [110, 137].

Further studies are needed to gather evidence of the comparative effects of AOP and OOP on the rate of fault exposure at these levels and under non-functional testing conditions. Conducting such studies would give a broader perspective on the implications of the effects of AOP and OOP on testability.

7.2.4 Program

The evidence gathered in this study is based on AspectJ and Java implementations of the Health Watcher program over maintenance activities. Chapter 5 showed that the Health Watcher program was carefully selected from candidates that fitted the study methodology because it was deemed to be the most representative of the general case. Gathering measures in a context which is representative of the general case maximises the degree to which the evidence is representative of general difference in effects of AOP and OOP on the testability of all implementations.

To further maximise generalisability, further studies are needed that are based on AOP and OOP implementations of a variety of representative programs. By broadening the number of programs from which evidence is gathered becomes even more representative of general difference in effects of AOP and OOP on the testability of all implementations.

7.2.5 Mutant Generation

The computational cost of applying mutation analysis in this study was very high. To be feasible, it required a number of machines to parallelise the application of the mutation analysis process. This computational cost is a barrier to conducting studies of this kind. Approaches to reduce this cost, by reducing the number of mutants that need to be generated to get an accurate result and optimising the mutant testing process, have been proposed for older languages and different fault models [116, 135, 112, 111, 114]. New research is needed to apply these approaches reduce the cost of applying mutation analysis to Java and AspectJ implementations. This would reduce the barrier to carrying out comparative studies of this type, making evidence of the effects of AOP and OOP on testability easier for researchers to gather.

Appendix A

Additional Results

```
1 t.test(aop$mutants,oop$mutants)
2
3   Welch Two Sample t-test
4
5 data:  aop$mutants and oop$mutants
6 t = 1.9761, df = 17.995, p-value = 0.06367
7 alternative hypothesis: true difference in means is not equal to 0
8 95 percent confidence interval:
9   -9.974042 325.774042
10 sample estimates:
11 mean of x mean of y
12   4728.2   4570.3
```

Listing A.1: Comparing number of mutants generated for AspectJ and Java implementations

```
1 t.test(aop$locations-oop$locations)
2
3   One Sample t-test
4
5 data:  aop$locations - oop$locations
6 t = -2.0444, df = 9, p-value = 0.07125
7 alternative hypothesis: true mean is not equal to 0
8 95 percent confidence interval:
9   -29.701597  1.501597
10 sample estimates:
11 mean of x
12   -14.1
```

Listing A.2: Comparing number of locations at which mutants are generated for AspectJ and Java implementations

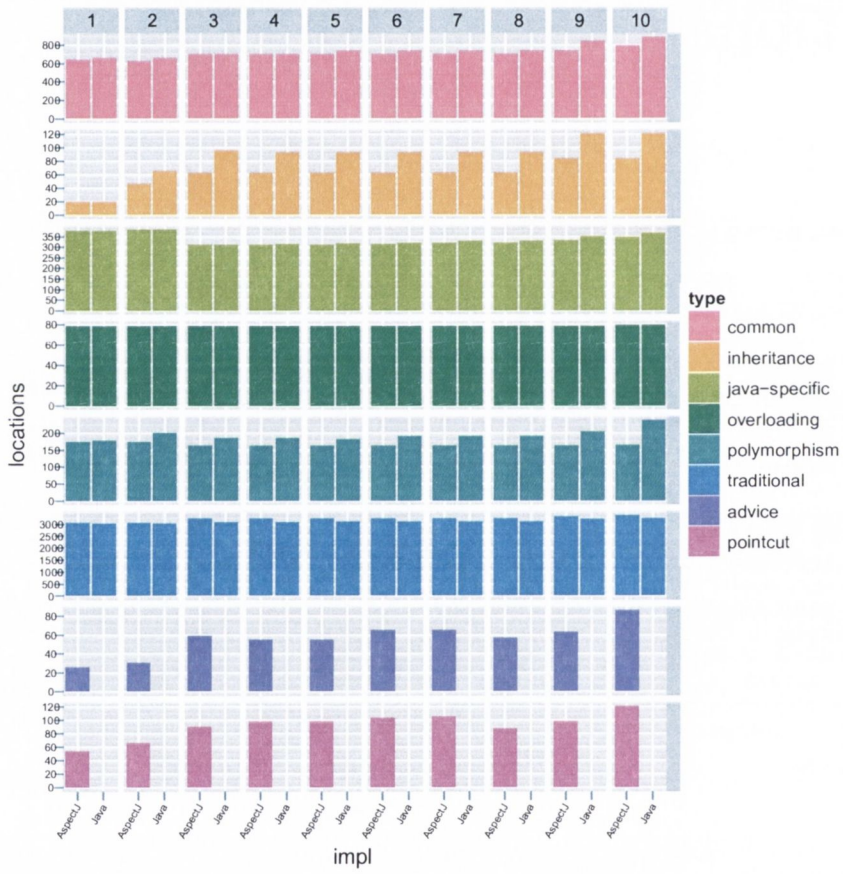


Figure A.1: Distribution of Fault Types


```

1 Call:
2 glm(formula = cbind(fail, (pass + pass_le + fail)) ~ impl + version,
3     family = binomial, data = data)
4
5 Deviance Residuals:
6     Min       1Q   Median       3Q      Max
7 -23.347244  -4.151534   0.009409   4.210211  21.515544
8
9 Coefficients:
10      Estimate Std. Error z value Pr(>|z|)
11 (Intercept) -3.195532   0.004061 -786.853 < 2e-16 ***
12 implJava    -0.061564   0.002356 -26.135 < 2e-16 ***
13 version2     0.001853   0.005500   0.337  0.73611
14 version3    -0.026266   0.005494  -4.781 1.74e-06 ***
15 version4    -0.060756   0.005539 -10.969 < 2e-16 ***
16 version5    -0.074845   0.005550 -13.487 < 2e-16 ***
17 version6    -0.059539   0.005519 -10.788 < 2e-16 ***
18 version7    -0.023780   0.005476  -4.343 1.41e-05 ***
19 version8    -0.017766   0.005471  -3.247 0.00117 **
20 version9    -0.013261   0.005053  -2.625 0.00868 **
21 version10   -0.057832   0.005064 -11.421 < 2e-16 ***
22 ---
23 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
24
25 (Dispersion parameter for binomial family taken to be 1)
26
27 Null deviance: 3028.6 on 19 degrees of freedom
28 Residual deviance: 1865.9 on 9 degrees of freedom
29 AIC: 2134.3
30
31 Number of Fisher Scoring iterations: 4

```

Listing A.3: Regression output for fault exposure

APPENDIX A. ADDITIONAL RESULTS

```

1 Call:
2 glm(formula = cbind(fail, (pass + fail)) ~ impl + version, family = binomial,
3     data = data)
4
5 Deviance Residuals:
6     Min       1Q   Median       3Q      Max
7 -20.72526  -1.95571  -0.01760   2.01268  18.51048
8
9      Estimate Std. Error z value Pr(>|z|)
10 (Intercept) -0.769506   0.004839  -159.03 <2e-16 ***
11 implJava    0.129397   0.002776   46.62 <2e-16 ***
12 version2   -0.115633   0.006530  -17.71 <2e-16 ***
13 version3   -0.143651   0.006510  -22.07 <2e-16 ***
14 version4   -0.148400   0.006563  -22.61 <2e-16 ***
15 version5   -0.155383   0.006573  -23.64 <2e-16 ***
16 version6   -0.146910   0.006541  -22.46 <2e-16 ***
17 version7   -0.141546   0.006491  -21.80 <2e-16 ***
18 version8   -0.143749   0.006485  -22.17 <2e-16 ***
19 version9   -0.072144   0.006028  -11.97 <2e-16 ***
20 version10  -0.136110   0.006018  -22.61 <2e-16 ***
21 ---
22 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
23
24 (Dispersion parameter for binomial family taken to be 1)
25
26     Null deviance: 4307.2  on 19  degrees of freedom
27 Residual deviance: 1097.6  on  9  degrees of freedom
28 AIC: 1359.4
29
30 Number of Fisher Scoring iterations: 3

```

Listing A.4: Regression output for fault exposure given execution

```

1 Call:
2 glm(formula = cbind((pass + fail), notexe) ~ impl + version,
3     family = binomial, data = data)
4
5 Deviance Residuals:
6     Min       1Q   Median       3Q      Max
7 -8.64845 -2.95916 -0.03712  2.85840  7.84903
8
9 Coefficients:
10      Estimate Std. Error z value Pr(>|z|)
11 (Intercept) -2.330187   0.002890 -806.35 <2e-16 ***
12 implJava    -0.208906   0.001606 -130.07 <2e-16 ***
13 version2     0.128189   0.003845  33.34 <2e-16 ***
14 version3     0.126485   0.003817  33.14 <2e-16 ***
15 version4     0.093387   0.003840  24.32 <2e-16 ***
16 version5     0.086284   0.003841  22.46 <2e-16 ***
17 version6     0.093219   0.003829  24.34 <2e-16 ***
18 version7     0.125251   0.003808  32.89 <2e-16 ***
19 version8     0.134074   0.003804  35.24 <2e-16 ***
20 version9     0.059340   0.003578  16.58 <2e-16 ***
21 version10    0.081284   0.003544  22.94 <2e-16 ***
22 ---
23 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
24
25 (Dispersion parameter for binomial family taken to be 1)
26
27 Null deviance: 19493.19 on 19 degrees of freedom
28 Residual deviance: 304.89 on 9 degrees of freedom
29 AIC: 588.68
30
31 Number of Fisher Scoring iterations: 3

```

Listing A.5: Regression output for location execution

Bibliography

- [1] Zuhoor Al-Khanjari, Martin Woodward, and Haider Ali Ramadhan. Critical analysis of the pie testability technique. *Software Quality Control*, 10(4):331–354, 2002.
- [2] Roger T. Alexander and James M. Bieman. Editorial: Aspect-oriented technology and software quality. *Software Quality Journal*, 12(2):93–97, 2004.
- [3] Roger T. Alexander, James M. Bieman, and Anneliese A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical report, Colorado State University, Fort Collins, Colorado, USA, 2004.
- [4] Mohammad Alshayeb and Wei Li. An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on Software Engineering*, 29(11):1043–1049, 2003.
- [5] Paul Ammann and Jefferson Offutt. *Introduction to Software Testing*. Cambridge University Press, 1 edition, 2008.
- [6] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *TACAS 07: Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 134–138, Braga, Portugal, March 2007.
- [7] James H. Andrews, Lionel C. Senior Briand, and Yvan Labiche. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.
- [8] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [9] William Ross. Ashby. *Design for a brain*. New York :Wiley,, 1954. <http://www.biodiversitylibrary.org/bibliography/6969>.
- [10] Calvin Austin and Monica Pawlan. Advanced programming for the java 2 platform. 2000.

- [11] Chitra Babu and Harshini Ramnath Krishnan. Fault model and test-case generation for the composition of aspects. *SIGSOFT Software Engineering Notes*, 34(1):1–6, 2009.
- [12] Jon S. Baekken and Roger T. Alexander. A candidate fault model for aspectj point-cuts. In *ISSRE 06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 169–178, Washington, DC, USA, 2006. IEEE Computer Society.
- [13] Carliss Baldwin and Kim Clark. *Design Rules vol. 1: The Power of Modularity*. MIT Press, Cambridge, MA, 2000.
- [14] Marc Bartsch and Rachel Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control*, 16(1):23–44, 2008.
- [15] Victor R. Basili, Lionel C. Briand, and Walclio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.
- [16] Thomas J. (Tim) Bergin. A history of the history of programming languages. *Communications of the ACM*, 50(5):69–74, 2007.
- [17] Mario L. Bernardi and Giuseppe A. Di Lucca. Testing coverage criteria for aspect oriented programs. *Software Quality Professional Journal*, 10(4):27–38, 2008.
- [18] Mario Luca Bernardi and Giuseppe Antonio Di Lucca. Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods. *Quality of Information and Communications Technology, International Conference on the*, 0:65–76, 2007.
- [19] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE 07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.
- [20] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1999.
- [21] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86, New York, NY, USA, 1986. ACM.

BIBLIOGRAPHY

- [22] Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.
- [23] Lionel Briand, Yvan Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. *ICSM 02: Proceedings of the 18th IEEE International Conference on Software Maintenance*, 0:0252, 2002.
- [24] Lionel C. Briand. A critical analysis of empirical research in software testing. In *ESEM 07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.
- [25] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering*, 27(6):513–530, 2001.
- [26] Lionel C. Briand, W. James Dzidek, and Yvan Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In IEEE Computer Society, editor, *ICSM 05: Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, September 25-30*, pages 687–690. IEEE, 2005.
- [27] Frederick P. Brooks. *No Silver Bullet Essence and Accidents of Software Engineering*, volume 20. IEEE Computer Society Press, 1987.
- [28] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.
- [29] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In *SCAM 04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.
- [30] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006.
- [31] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL 80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM.
- [32] Yuanfang Cai and Kevin J. Sullivan. A value-oriented theory of modularity in design. *SIGSOFT Softw. Eng. Notes*, 30(4):1–4, 2005.

- [33] F. Calzolari, P. Tonella, and G. Antoniol. Maintenance and testing effort modeled by linear and nonlinear dynamic systems. *Information and Software Technology*, 43(8):477 – 486, 2001.
- [34] Walter Cazzola and Alessandro Marchetto. Aop and hidden metrics: Separation, extensibility and adaptability in sw measurements. *Journal of Object Technology, OOPS Track at SAC 2007*, 7(2):53–68, (2008).
- [35] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *ARE 04: Proceedings of the 1st Workshop on Aspect Reverse Engineering*, 2004.
- [36] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. Is aop code easier or harder to test than oop code? In *AOSD 05: On-line Proceedings of the Workshop on Testing Aspect-Oriented Programs*, 2005.
- [37] Philippe Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. volume 0, page 267, Los Alamitos, CA, USA, 2001. IEEE Computer Society.
- [38] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [39] A. Colyer, R. Harrop, R. Johnson, A. Vasseur, D. Beuche, and C. Beust. Point/-counterpoint. *Software, IEEE*, 23(1):72–75, Jan.-Feb. 2006.
- [40] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.
- [41] William Jay Conover. *Practical nonparametric statistics*. Wiley series in probability and mathematical statistics. Wiley, New York [u.a.], 2. ed edition, 1980.
- [42] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Journal of Empirical Software Engineering*, 1(2):109–132, 1996.
- [43] Romain Delamare, Benoit Baudry, Sudipto Ghosh, and Yves Le Traon. A test-driven approach to developing pointcut descriptors in aspectj. In *ICST 09: Proceedings of International Conference on Software Testing, Verification, and Validation*, pages 376–385, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [44] Marcio E. Delamaro, Jose C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

BIBLIOGRAPHY

- [45] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [46] Edward. W. Dijkstra. Ewd 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.
- [47] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2007.
- [48] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.
- [49] Stephan G. Eick, Tod L. Graves, Alan F. Karr, James S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan 2001.
- [50] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001.
- [51] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
- [52] Julian J. Faraway. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. Texts in Statistical Science. Chapman & Hall/CRC, December 2005.
- [53] Fabiano Cutigi Ferrari, Jose Carlos Maldonado, and Awais Rashid. Mutation testing for aspect-oriented programs. *ICST 08: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, 0:52–61, 2008.
- [54] Eduardo Figueiredo, Nelio Cacho, Claudio Sant’Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE 08: Proceedings of the 30th international conference on Software engineering*, pages 261–270, New York, NY, USA, 2008. ACM.
- [55] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília Mary F. Rubira. Exceptions and aspects: the devil

- is in the details. In *SIGSOFT 06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–162, New York, NY, USA, 2006. ACM.
- [56] Donald Firesmith. Testing object-oriented software. In Raimund K. Ege, Madhu S. Singh, and Bertrand Meyer, editors, *TOOLS (11)*, pages 407–426. Prentice Hall, 1993.
- [57] Vladimir N. Fleyshgakker and Stewart N. Weiss. Efficient mutation analysis: a new approach. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 185–195, New York, NY, USA, 1994. ACM.
- [58] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. *SIGSOFT Software Engineering Notes*, 23(6):153–162, 1998.
- [59] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.
- [60] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.
- [61] Leonard Gallagher and Jefferson A. Offutt. Test sequence generation for integration testing of component software. *The Computer Journal*, 52(5):514–529, August 2007.
- [62] Alessandro F. Garcia, Cláudio Sant’Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos José Pereira de Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect Oriented Software Development*, 1:36–74, 2006.
- [63] Phil Greenwood, Thiago T. Bartolomei, Eduardo Figueiredo, Marcos Dósea, Alessandro F. Garcia, Nélio Cacho, Cláudio Sant’Anna, Sérgio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP 07: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 176–200, 2007.
- [64] Phil Greenwood, Alessandro Garcia, Awais Rashid, Eduardo Figueiredo, Claudio Sant’Anna, Nelio Cacho, Americo Sampaio, Sergio Soares, Paulo Borba, Marcos Dosea, Ricardo Ramos, Uira Kulesza, Thiago Bartolomei, Monica Pinto, Lidia Fuentes, Nadia Gamez, Ana Moreira, Joao Araujo, Thais Batista, Ana Medeiros, Francisco Dantas, Lyrene Fernandes, Jan Wloka, Christina Chavez, Robert France, and Isabel Brito. On the contributions of an end-to-end aosd testbed. In *EA 07:*

BIBLIOGRAPHY

- Proceedings of the Early Aspects at ICSE*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.
- [65] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [66] Priti. S. Grover, Rajesh Kumar, and Avadhesh Kumar. Measuring changeability for generic aspect-oriented systems. *SIGSOFT Software Engineering Notes*, 33(6):1–5, 2008.
- [67] Florian Haftmann, Donald Kossmann, and Eric Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1):145–164, 2007.
- [68] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA 02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.
- [69] Mark Harman, Fayezi Islam, Tao Xie, and Stefan Wappler. Automated test data generation for aspect-oriented programs. In *Proc. 8th International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 185–196, March 2009.
- [70] Daniel Hughes, Phil Greenwood, and Geoff Coulson. A framework for testing distributed systems. In *P2P 04: Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing*, pages 262–263, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [71] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [72] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [73] IEEE, editor. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, 1990.
- [74] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
- [75] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

- [76] Voas Jeffrey M. Factors that affect software testability. Technical report, 1991.
- [77] Sun-Woo Kim John, John A. Clark, and John A. Mcdermid. Assessing test set adequacy for object-oriented programs using class mutation. In *SoST'99: Proceedings of the International Symposium on Software Technology*, pages 72–83, 1999.
- [78] Sunwoo Kim John, John A. Clark, and John A. Mcdermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of Net ObjectDays*, pages 9–12, 2000.
- [79] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer, 2001.
- [80] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP 2006: Proceedings of 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.
- [81] Rasul A. Khan and K. Mustafa. Metric based testability model for object oriented design (mtmood). *SIGSOFT Software Engineering Notes*, 34(2):1–6, 2009.
- [82] Gregor Kiczales. Aspect-oriented programming. *ACM Computer Survey*, 28(4es), December 1996.
- [83] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
- [84] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.
- [85] Sun-Woo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability*, 11(3):207–225, 2001.
- [86] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.
- [87] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM 06: Proceedings of the 22nd IEEE International*

BIBLIOGRAPHY

- Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.
- [88] Avadhesh Kumar, Rajesh Kumar, and PS Grover. An evaluation of maintainability of aspect-oriented systems: a practical approach. *International Journal of Computer Science and Security*, 1(1):1–9, 2007.
- [89] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [90] Otávio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero, and Cristina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP 06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38, New York, NY, USA, 2006. ACM.
- [91] Otavio Augusto Lazzarini Lemos, Auri Marcelo Rizzo Vincenzi, Jose Carlos Maldonado, and Paulo Cesar Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal Systems Software*, 80(6):862–882, 2007.
- [92] Jingyue Li, Axel Anders Kvale, and Reidar Conradi. A case study on improving changeability of cots-based system using aspect-oriented programming. *Journal of Information Science and Engineering*, 22(2):155 – 174, March 2006.
- [93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems Software*, 23(2):111–122, 1993.
- [94] Wei Li and Sallie Henry. An empirical study of maintenance activities in two object-oriented systems. *Journal of Software Maintenance*, 7(2):131–147, 1995.
- [95] Bennet P. Lientz, Burton E. Swanson, and Gerry E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.
- [96] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.
- [97] Cristina Videira Lopes and Sushil Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. *Trasnactions on Aspect Oriented Software Development*, pages 1–35, 2006.
- [98] Yu S. Ma, Jefferson A. Offutt, and Yong R. Kwon. Mujava: an automated class mutation system: Research articles. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

- [99] Yu-Seung Ma, Yong-Rae Kwon, and Jefferson A. Offutt. Inter-class mutation operators for java. In *Software Reliability Engineering, 2002. ISSRE 2002. Proceedings. 13th International Symposium on*, pages 352–363, 2002.
- [100] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *WWW 2007: Proceedings of the 16th International Conference on World Wide Web*, pages 667–676, May 2007.
- [101] Robert Martin. *Stability*. Engineering Notebook, 1997.
- [102] Johannes Mayer. Efficient and effective random testing based on partitioning and neighborhood. In Kang Zhang, George Spanoudakis, and Giuseppe Visaggio, editors, *SEKE 06: Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, pages 479–484, San Francisco Bay, USA, July 2006. Knowledge Systems Institute Graduate School.
- [103] Johannes Mayer and Ralph Guderlei. An empirical study on the selection of good metamorphic relations. In *COMPSAC 06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 475–484, Washington, DC, USA, 2006. IEEE Computer Society.
- [104] Johannes Mayer and Ralph Guderlei. On random testing of image processing applications. In *QSIC 06: Proceedings of the Sixth International Conference on Quality Software*, pages 85–92, Beijing, China, October 2006. IEEE Computer Society.
- [105] Johannes Mayer and Christoph Schneckenburger. An empirical analysis and comparison of random testing techniques. In *ISESE 06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 105–114, New York, NY, USA, 2006. ACM.
- [106] Russell Miles. *AspectJ Cookbook*. O’Reilly Media, Inc., 2004.
- [107] Geoffrey Moore. *Crossing the Chasm*. HarperBusiness, New York, 2002.
- [108] Jennifer Munnely, Serena Fritsch, and Siobhan Clarke. An aspect-oriented approach to the modularisation of context. In *PERCOM 07: Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications*, pages 114–124, Washington, DC, USA, 2007. IEEE Computer Society.
- [109] Gail C. Murphy, Robert J. Walker, and Elisa L.A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999.

BIBLIOGRAPHY

- [110] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
- [111] Akbar Siami Namin and James H. Andrews. Finding sufficient mutation operators via variable reduction. In *MUTATION '06: Proceedings of the 2nd Workshop on Mutation Analysis*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.
- [112] Akbar Siami Namin and James H. Andrews. On sufficiency of mutants. In *ICSE COMPANION 07: Companion to the Proceedings of the 29th International Conference on Software Engineering*, pages 73–74, Washington, DC, USA, 2007. IEEE Computer Society.
- [113] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.
- [114] Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5, 1996.
- [115] Jefferson A. Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. *Software Reliability Engineering, International Symposium on*, 0:84, 2001.
- [116] Jefferson A. Offutt and Lee Stephan D. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.
- [117] Jefferson A. Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, 1996.
- [118] Jefferson A. Offutt, Jeff Voas, and Jeff Payne. Mutation operators for ada. Technical report, George Mason University and Reliable Software Technologies, 1996.
- [119] Reza M. Parizi and Abdul A. Ghani. A survey on aspect-oriented testing approaches. In *ICCSA 07' International Conference on Computational Science and its Applications*, pages 78–85, Aug. 2007.
- [120] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 139–150, 1979.
- [121] Derek Partridge and Wojtek Krzanowski. Software diversity: practical statistics for its measurement and exploitation. *Information and Software Technology*, 39(10):707 – 717, 1997.

- [122] Macario Polo, Sergio Tendero, and Mario Piattini. Integrating techniques and tools for testing automation: Research articles. *Software Testing, Verification and Reliability*, 17(1):3–39, 2007.
- [123] George Polya. *How To Solve It - A New Aspect of Mathematical Method*. Princeton University Press, Princeton, 2nd edition edition, 1971.
- [124] R. Purushothaman and D.E. Perry. Toward understanding the rhetoric of small source code changes. *Software Engineering, IEEE Transactions on*, 31(6):511–526, June 2005.
- [125] Awais Rashid and Lynne Blair. Editorial: Aspect-oriented programming and separation of crosscutting concerns. *Computer Journal*, 46(5):527–528, 2003.
- [126] George Reese. *Database Programming with JDBC and Java, Second Edition*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.
- [127] Hiralal Agrawal Richard, Richard A. Demillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Software Engineering Research Center, Purdue University, 1989.
- [128] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, 1993.
- [129] Dieter H. Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, 13(3):344–354, 1987.
- [130] Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Simulation-based test adequacy criteria for distributed systems. In *SIGSOFT 06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–241, New York, NY, USA, 2006. ACM.
- [131] Pablo Sánchez, Lidia Fuentes, Andrew Jackson, and Siobhán Clarke. Aspects at the right time. *T. Aspect-Oriented Software Development*, 4:54–113, 2007.
- [132] Stephen R. Schach. Testing: principles and practice. *ACM Computing Surveys*, 28(1):277–279, 1996.
- [133] Rakesh Shukla, David Carrington, and Paul Strooper. A passive test oracle using a component’s api. In *APSEC 05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 561–567, Washington, DC, USA, 2005. IEEE Computer Society.

BIBLIOGRAPHY

- [134] Rakesh Shukla, Paul A. Strooper, and David A. Carrington. A framework for statistical testing of software components. *International Journal of Software Engineering and Knowledge Engineering*, 17(3):379–405, 2007.
- [135] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE 08: Proceedings of the 30th International Conference on Software Engineering*, pages 351–360, New York, NY, USA, 2008. ACM.
- [136] TIOBE Software. Tiobe programming community index for march 2009. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, March 2009.
- [137] Ian Sommerville. *Software engineering (2nd ed.)*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1985.
- [138] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, pages 205–232, 1979.
- [139] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM 05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.
- [140] A. Tappenden, P. Beatty, and J. Miller. Agile security testing of web-based systems via httpunit. In *ADC 05: Proceedings of the Agile Development Conference*, pages 29–38, Washington, DC, USA, 2005. IEEE Computer Society.
- [141] Gregory Tassej. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
- [142] Tom Tourwe, Johan Brichau, and Kris Gybels. On the existence of the AOSD-evolution paradox. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.
- [143] Shiu Lun Tsang, S. Clarke, and E. Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. In *OORTDC 04: Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 291–300, May 2004.
- [144] Shiu Lun Tsang, Siobhn Clarke, and Elisa Baniassad. Object oriented metrics for aspect systems: Limiting empirical inference based on modularity. Technical report, Trinity College Dublin, 2004.

-
- [145] Arie van Deursen, Marius Marin, and Leon Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw. *CoRR*, abs/cs/0503015, 2005.
- [146] Jeffery M. Voas and Keith W. Miller. Software testability: the new verification. *Software, IEEE*, 12(3):17–28, May 1995.
- [147] Jeffrey Voas. Testability of object-oriented systems. Technical Report GCR-95-675, National Institute of Standards and Technology, 1995.
- [148] Jeffrey Voas, Larry Morrel, and Keith Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, 1991.
- [149] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99, In Proceedings of the 21st International Conference on Software Engineering*, pages 120–130, 22–22 May 1999.
- [150] W. Eric Wong, Aditya P. Mathur, and Jose C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *Software Quality and Productivity: Theory, practice and training*, pages 258–265, London, UK, UK, 1995. Chapman & Hall, Ltd.
- [151] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE 04: Proceedings 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, September 2004.
- [152] Tao Xie and Jianjun Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *AOSD 06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM.
- [153] Stephan S. Yau and James S. Collofello. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, SE-11(9):849–856, Sept. 1985.
- [154] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference, 1979.
- [155] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. In *COMPSAC 07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 431–438, Washington, DC, USA, 2007. IEEE Computer Society.

BIBLIOGRAPHY

- [156] Chuan Zhao and Roger T. Alexander. Testing aspectj programs using fault-based testing. In *WTAOP 07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 13–16, New York, NY, USA, 2007. ACM.