Supporting Project Comprehension with Revision Control System Repository Analysis

ANDREW JAMES BURN

Supporting Project Comprehension with Revision Control System Repository Analysis

PhD Thesis

Andrew Burn

Technology Enhanced Learning
School of Engineering and Computing Sciences
Durham University

Copyright

Declaration

No part of the material provided has previously been submitted by the author for a higher degree in Durham University or in any other university. All the work presented here is the sole work of the author and no-one else.

Acknowledgements

This dissertation was begun six years ago, and would never have been completed without the support, guidance and endless patience of my supervisor, Liz Burd.

I would also like to thank Stephen Cummins for the time and effort spent reviewing the thematic analysis, potentially the least interesting task I have ever asked of anybody.

Further thanks go to David Budgen, Barbara Kitchenham and the rest of the EPIC team for the advice and feedback they gave me, and also for helping me to fully understand the necessity of clear, repeatable empirical research in Computer Science.

An honourable mention goes to Sidhe Interactive and Module for their "80's new wave, stadium rock and intergalactic space rock opera" soundtrack to the video game "Shatter", which seems specifically crafted to help me write, code and think.

Finally, my gratitude and love go to my wife Tamora, who has supported me throughout my PhD with the continuous provision of pies, cakes and drinks.

Abstract

Context: Project comprehension is an activity relevant to all aspects of software engineering, from requirements specification to maintenance. The historical, transactional data stored in revision control systems can be mined and analysed to produce a great deal of information about a project.

Aims: This research aims to explore how the data-mining, analysis and presentation of revision control systems can be used to augment aspects of project comprehension, including change prediction, maintenance, visualization, management, profiling, sampling and assessment.

Method: A series of case studies investigate how transactional data can be used to support project comprehension. A thematic analysis of revision logs is used to explore the development process and developer behaviour. A benchmarking study of a history-based model of change prediction is conducted to assess how successfully such a technique can be used to augment syntax-based models. A visualization tool is developed for managers of student projects with the aim of evaluating what visualizations best support their roles. Finally, a quasi-experiment is conducted to determine how well an algorithmic model can automatically select a representative sample of code entities from a project, in comparison with expert strategies.

Results: The thematic analysis case study classified maintenance activities in 22 undergraduate projects and four real-world projects. The change prediction study calculated information retrieval metrics for 34 undergraduate projects and three real-world projects, as well as an in-depth exploration of the model's performance and applications in two selected projects. File samples for seven projects were generated by six experts and three heuristic models and compared to assess agreement rates, both within the experts and between the experts and the models.

Conclusions: When the results from each study are evaluated together, the evidence strongly shows that the information stored in revision control systems can indeed be used to support a range of project comprehension activities in a manner which complements existing, syntax-based techniques. The case studies also help to develop the empirical foundation of repository analysis in the areas of visualization, maintenance, sampling, profiling and management; the research also shows that students can be viable substitutes for industrial practitioners in certain areas of software engineering research, which weakens one of the primary obstacles to empirical studies in these areas.

Contents

Declaration of Authorship i

Acknowledgements ii

Abstract iii

List of Figures ix

List of Tables xiii

List of Abbreviations xv

1 Introduction 1

1.1 Background 1

1.2 Research Aims 3

1.2.1 Profiling Projects 7

1.2.2 Change Prediction and Impact Analysis 8

1.2.3 Software Visualization 9

1.2.4 Sampling 9

1.3 Summary 10

2 Literature Review 11

2.1 Introduction 11

2.2 Overview 12

2.3 Project Comprehension: Understanding Software . . . 13
2.4 Software Change – Evolution and Maintenance . . . 15
2.4.1 Software Evolution . . . 15
2.4.2 Software Maintenance . . . 19
2.5 Impact Analysis and Change Prediction . . . 21
2.5.1 Syntax-Based Impact Analysis . . . 22
2.6 Software Visualization . . . 24
2.6.1 Types of Visualization . . . 25
2.6.2 Software Visualization Tools . . . 27
2.6.3 Empirical Studies on Software Visualization . . . 29
2.7 Summary . . . 33

3 Mining Revision Control Repositories 34
3.1 Introduction . . . 34
3.2 Revision Control . . . 35
3.3 Structured Review . . . 35
3.3.1 Overview . . . 36
3.3.2 Data Mining . . . 37
3.3.3 Impact Analysis and Change Prediction Using Historical Data . . . 42
3.3.4 Supplementary Data Sources . . . 46
3.3.5 Revision Control in Education . . . 47
3.4 Summary . . . 49

4 Design and Implementation 50
4.1 Introduction . . . 50
4.2 Requirements . . . 51
4.2.1 Thematic Analysis . . . 51
4.2.2 History-Based Change Prediction . . . 51
4.2.3 File Sampling . . . 52
4.2.4 Software Visualization . . . 52

4.3 Data Extraction and Storage 52
4.4 Perceive 57
4.4.1 Preprocessing 58
4.4.2 Class Design and Data Structures 59
4.4.3 File Information 59
4.4.4 Limitations and Future Development 60
4.5 Workflow 60
4.6 Summary 61

5 Case Study: Thematic Analysis 63
5.1 Introduction 63
5.2 Case Study Design 64
5.2.1 Research Goals 64
5.2.2 Research Questions 65
5.2.3 Evaluation Paradigm and Techniques 66
5.2.4 Practical Issues 66
5.2.5 Ethical Issues 67
5.2.6 Evaluation and Discussion of Results 67
5.3 Thematic Analysis 67
5.4 Study 1.1: 12 Group Projects 68
5.4.1 Research Questions 69
5.4.2 Design 70
5.4.3 Limitations and Threats to Validity 73
5.4.4 Evaluation 73
5.4.5 Conclusions 78
5.5 Study 1.2: 22 Group Projects and Four
Open Source Projects 81
5.5.1 The Projects 82
5.5.2 Research Questions 83
5.5.3 Thematic Analysis 84
5.5.4 Limitations and Threats to Validity 85
5.5.5 Evaluation 85

5.5.6 Conclusions 92
5.6 Study 1.3: A Complete Open Source Project Analysis 93
5.6.1 Research Questions 94
5.6.2 Thematic Analysis 94
5.6.3 Limitations and Threats to Validity 94
5.6.4 Evaluation 95
5.6.5 Conclusions 102
5.7 Case Study Discussion 104
5.7.1 *Research Question 2* 105
5.7.2 *Research Question 3* 106
5.8 Case Study Conclusions 107
5.9 Summary 107

6 Case Study: History-Based Change Prediction 108
6.1 Introduction 108
6.2 Case Study Design 110
6.2.1 Research Goals 111
6.2.2 Research Questions 112
6.2.3 Evaluation Paradigm and Techniques 113
6.2.4 Practical Issues 113
6.2.5 Ethical Issues 116
6.2.6 Evaluation and Discussion of Results 116
6.3 Perceive 116
6.4 Study 2.1: Empirical Benchmarking Study 117
6.4.1 Research Questions 117
6.4.2 The Models 118
6.4.3 Phases 119
6.4.4 Implementation Details 121
6.4.5 Limitations and Threats to Validity 122
6.4.6 Results 123
6.4.7 Evaluation 125
6.4.8 Discussion 140

6.4.9 Conclusions . . . 143
6.5 Study 2.2: Project Applications . . . 144
6.5.1 Introduction . . . 144
6.5.2 The Projects . . . 145
6.5.3 Overview of Perceive . . . 146
6.5.4 Case Study 2.2A: SEG . . . 146
6.5.5 Case Study 2.2B: PuTTY . . . 152
6.6 Study 2.3: Improving Perceive . . . 154
6.6.1 Research Questions . . . 155
6.6.2 Perceive . . . 155
6.6.3 Study Design . . . 155
6.6.4 Results . . . 156
6.6.5 Evaluation . . . 156
6.6.6 Discussion . . . 157
6.6.7 Conclusions . . . 158
6.7 Case Study Discussion . . . 158
6.7.1 *Research Question 4* . . . 158
6.7.2 *Research Question 5* . . . 159
6.7.3 *Research Question 6* . . . 159
6.8 Case Study Conclusions . . . 160
6.9 Future Work . . . 160
6.10 Summary . . . 162

7 Case Studies: Project Management and Assessment 163
7.1 Introduction . . . 163
7.2 Case Study: Software Visualization and Project Management . 164
7.2.1 Case Study Design . . . 164
7.2.2 Study 3.1: Software Visualization and Project Management Using Perceive . . . 167
7.2.3 Case Study Discussion . . . 187
7.2.4 Case Study Conclusions . . . 188
7.3 Case Study: Project Sampling . . . 189

7.3.1 Case Study Design 189
7.3.2 Study 4.1: Supporting Group Project Assessment . . . 191
7.3.3 Case Study Discussion 204
7.3.4 Case Study Conclusions 205
7.4 Conclusions 205
7.5 Summary 206

8 Conclusions and Future Research 208
8.1 Introduction 208
8.2 Case Study 1: Profiling Projects 209
8.2.1 *Research Question 2* 210
8.2.2 *Research Question 3* 211
8.2.3 Summary 211
8.3 Case Study 2: Change Prediction 212
8.3.1 *Research Question 4* 212
8.3.2 *Research Question 5* 213
8.3.3 *Research Question 6* 213
8.3.4 Summary 214
8.4 Case Study 3: Software Visualization 214
8.4.1 *Research Question 7* 215
8.4.2 *Research Question 8* 216
8.4.3 Summary 216
8.5 Case Study 4: Sampling 217
8.5.1 Summary 218
8.6 Future Research 220
8.7 *RQ 1*: How Can Data Mining of Revision Control Systems be Applied to Support Project Comprehension? 221

A Structured Literature Review References 223

List of Figures

1.1 Study Structure 5
1.2 Structure of the research questions and studies 6

2.1 Dependency Graph 28
2.2 SeeSoft 29
2.3 sv3D 30

3.1 A manual classification of large commits 39
3.2 Correlation between grade and LOC 41
3.3 Cluster analysis based on modification request relationships . 44

4.1 The MySQL design 54
4.2 A simple web interface for Perceive 56
4.3 The complete Perceive workflow 61
4.4 The simplified Perceive workflow 61

5.1 Structure of the research questions and studies of the project profiling case study 65
5.2 Activity types broken down by group 76
5.3 Activity over time 76
5.4 Comparison of activity types across campuses 79
5.5 A broad comparison of work levels on each campus over time . 79
5.6 C1 students – Breakdown of the various activity types 80
5.7 C2 students – Breakdown of the various activity types 80
5.8 Activity Types Between SEG Project Years 87

5.9 Breakdown of activity types between campuses 89
5.10 Activity Types Between Sets of Projects 90
5.11 Breakdown of desirable and undesirable activities 91
5.12 Distribution of activity types for PuTTY 95
5.13 How the activity distribution of the primary activities changes over the course of the project 96
5.14 Cumulative activity breakdown over the development history of PuTTY 97
5.15 Comparison of activity distributions for SEG, PuTTY1 and PuTTY2 98
5.16 Activities preceding project milestones 99
5.17 Breakdown of activity types by the PuTTY developers. 100
5.18 Breakdown of activity types by the PuTTY developers 101
5.19 Owner Visualization of PuTTY 103

6.1 Structure of the research questions and studies of the change prediction case study 111
6.2 Phase 1: Effects of the support parameter on academic projects 129
6.3 Phase 1: Effects of the confidence parameter on academic projects 129
6.4 Phase 1: Effects of the support parameter on FOSS projects . 131
6.5 Phase 1: Effects of the confidence parameter on FOSS projects 131
6.6 Phase 1: Effects of the support parameter on large projects . . 132
6.7 Phase 1: Effects of the confidence parameter on large projects 133
6.8 Phase 2: Effects of the support parameter on academic projects 136
6.9 Phase 2: Effects of the confidence parameter on academic projects 136
6.10 Phase 2: Effects of the support parameter on FOSS projects . 137
6.11 Phase 2: Effects of the confidence parameter on FOSS projects 137
6.12 Phase 2: Effects of support and file selection 138
6.13 Phase 2: Effects of confidence and file selection 139
6.14 Phase 2: Overview of support and confidence on precision . . 139
6.15 Phase 2: Overview of support and confidence on recall 140
6.16 Phase 2: Effects of support on `Perceive` and $\texttt{Perceive}_2$ . . . 141

6.17 Phase 2: Effects of confidence on `Perceive` and $\texttt{Perceive}_2$ . . 141

7.1 Structure of the research questions and studies of the software visualization case study 164
7.2 Overview of a student project 169
7.3 The list of revisions 170
7.4 File activity 171
7.5 Developer information 172
7.6 File information 172
7.7 Project overview 173
7.8 The Modified Icicle Plot (MIPVis) 174
7.9 The MIPVis with a single user selected 175
7.10 Radial Visualization 176
7.11 Files plotted along a Hilbert Curve 177
7.12 HilbertVis with coloured by activity 178
7.13 HilbertVis with the files coloured by change type 179
7.14 Flow Visualization 180
7.15 Owner Visualization 181
7.16 Bar chart – cumulative revision 183
7.17 Line chart – changes by time 184
7.18 Pie chart – change types 185
7.19 Progress report 185
7.20 Structure of the research questions and studies of the file sampling case study 190
7.21 Sampling agreement rates 199
7.22 Overall performance of the three Perceive models 202

List of Tables

5.1 Distribution of maintenance activities using the second set of codes 70
5.2 Distribution of maintenance activities 74
5.3 Distribution of maintenance activities 86

6.1 Map of research questions to studies 112
6.2 Phase 1: Overview 123
6.3 Phase 1: All models, code files 123
6.4 Phase 1: All models, extended file set 124
6.5 Phase 1: Project category, code files 124
6.6 Phase 1: Project category, extended file set 124
6.7 Phase 1: Correlation between metrics and performance 125
6.8 Phase 2: Overview 126
6.9 Phase 2: Effects of file selection 126
6.10 Phase 2: Effects of project category 126
6.11 Phase 2: Effects of fail-weight (FW) 126
6.12 Phase 2: Correlation between metrics and performance 127
6.13 Overview of the projects 145
6.14 Performance of `Perceive` for various project groups 147
6.15 Performance of `Perceive` for various project groups 152
6.16 Overview of the performance of the new change-prediction models 156
6.17 Peak F-Scores of each model for the two project sets 156

7.1 Allocations of projects to assessors 193
7.2 Summary of the number of files suggested for each project . . 197
7.3 Number of files suggested for each project by each assessor . . 197
7.4 Coverage achieved by each expert and model 198
7.5 Agreement rate 200
7.6 Agreement rate in singles, doubles and triples 201

A.1 Overview of the results of the structured literature review . . . 227

List of Abbreviations

1. *GQM*: Goal Question Metric
2. *HBCP*: History-Based Change Prediction
3. *RCS*: Revision Control System
4. *RQ*: Research Question
5. *RQCP*: Research Question: Change Prediction
6. *RQFS*: Research Question: File Selection
7. *RQTA*: Research Question: Thematic Analysis
8. *SEG*: Software Engineering Group

Chapter 1

Introduction

1.1 Background

As a process from beginning to end, from specifying requirements to ongoing maintenance, software engineering activities require a high degree of understanding and the construction of complex mental models (Paul et al., 1991). For some software projects, the necessary level of comprehension can be maintained by expertise and experience alone, but this quickly becomes untenable as a project grows and evolves. As a project grows larger, a developer's model of it will become less and less complete; as more people work on the project their contributions must be constantly melded into the developer's model. Even more challengingly, a new developer must construct this model, and gain this understanding of the project completely from scratch (Anvik and Murphy, 2007). With a high quality project – one with proper documentation, structured, well-commented code and access to the experience or "group memory" of the current developers – this is a tractable problem. In many cases the project is not so well documented, the original developers are no longer involved with the project, the new developer lacks experience or resources are constrained and comprehension becomes unachievable (Cubranic and Murphy, 2003).

The activity of comprehension has a number of applications and support-

ing concepts in software development, such as code comprehension, project comprehension, software visualization, impact analysis, change prediction or unit testing. A number of techniques, methods, tools and systems have therefore been researched and implemented to support developers; the underlying concepts behind these, and examples of their implementation and evaluation are discussed in depth in the literature review in Chapter 2. Considerable research effort has been conducted on program comprehension – understanding the code which makes up a project, either at design-time or run-time – and such research tends to focus on analysing one or more (or all) code entities. With knowledge of the syntax and functionality of a programming language or paradigm, analysing source code can support program comprehension in a range of ways (Storey et al., 2000; Soloway and Ehrlich, 1989). At a simple level these methods mimic the strategies used by the programmers, but do so faster and at greater scale than the programmer can achieve unsupported, presenting the results at a sufficient granularity or abstraction to be most useful. An example of this is calculating dependencies – a human can maintain a model of how code entities are related, but algorithmically generating a dependency graph (Ferrante et al., 1987) can provide the same information more rapidly and across an entire project. At a more complex level, comprehension tools can use software metrics to calculate cohesion and coupling (Gall et al., 1998), make suggestions for refactoring or predict error-prone code (Graves et al., 2000).

A growing field of research is that of data mining, which has significant application in program comprehension, as discussed in Chapter 3. Revision control systems (RCS), such as SubVersion (Collins-Sussman, 2002) or CVS (Beck, 2003) contain a great deal of information about a project's history, transactional information which can be extracted and analysed to facilitate program comprehension activities. When a tool analyses a single version of a project it is restricted to data from that one snapshot of the project and can only extract information from the project's structure, existing documentation and syntactic knowledge of the programming platform; a tool analysing many

versions of a software project can expand on this in two ways – by being able to perform the same processes across a range of versions, and by being able to analyse the actual development processes themselves (Glassy, 2005). Such systems are of use not just to developers but to users across all stages of a project, as research has demonstrated that even related artefacts such as requirements documents can be analysed in this fashion (Dekhtyar et al., 2004). Project managers can use such analyses to make decisions on how the project should proceed or how to allocate resources; maintainers can use historical data to create and access an artificial "group memory" to make use of the expertise and experience of previous developers (Cubranic and Murphy, 2003).

1.2 Research Aims

The research presented in this dissertation seeks to address the question:

- RQ 1: How can data mining of revision control systems be applied to support project comprehension?

Specifically, this research aims to explore the methods by which data mining of historical project data from revision control system repositories, and demonstrate empirically that analysis of transactional data – the metadata associated with changes, such as comment logs, rather than the source code itself – can support a range of project comprehension activities in educational, industrial and academic contexts. As Chapters 2 and 3 describe, much of the research in these fields does not have a strong evidence-based foundation, and where empirical research exists there are often inconclusive or conflicting results.

In order to address this research question, four activities which require or support project comprehension have been identified – project profiling, change prediction, software visualization and file sampling. These activities are described in more detail in the following sections. For each of these four

activities a case study is designed and conducted to evaluate the degree to which data-mining RCS repositories can be used to support the activity under study. Each case study identifies one or more research questions, which will be used to feed back to the overall research question, *RQ 1*.

These studies will use a range of established empirical techniques, both qualitative and quantitative, with the dual goals of answering a series of research questions regarding data mining and the expansion of the empirical landscape of the field by designing methodical, structured and repeatable studies using clearly defined and representative subjects and data.

The subject of each case study is one or more sub-studies, which will directly evaluate the specific application of a technique based on RCS repository analysis; these sub-studies will identify their own research questions. The success or failure of the sub-studies to apply repository analysis to support the activity under study will form the basis of the evaluation of the parent case study. If a sub-study shows no benefit in using repository analysis to support the activity under study, then the case study will have no evidence to suggest that repository analysis is able to support project comprehension. Conversely, if a sub-study finds that the activity under study benefits from repository analysis then the parent case study will be able to show – for that particular activity – that repository analysis can be used to support project comprehension.

Figure 1.1 outlines the structure of the studies, showing how each case study evaluates the findings of a particular technique used to support a project comprehension activity, which in turn feed into the overall research goal.

The outline of the case studies, sub-studies and research questions are shown in more detail in Figure 1.2.

The DECIDE Framework will be used to structure the research aims, questions and studies (Sharp et al., 2002). Like the Goal Question Metric (GQM) approach (Basili et al., 1994) the DECIDE Framework outlines a process and structure for conducting research; however, the components of the framework are more fine-grained than that of GQM, and are as follows:

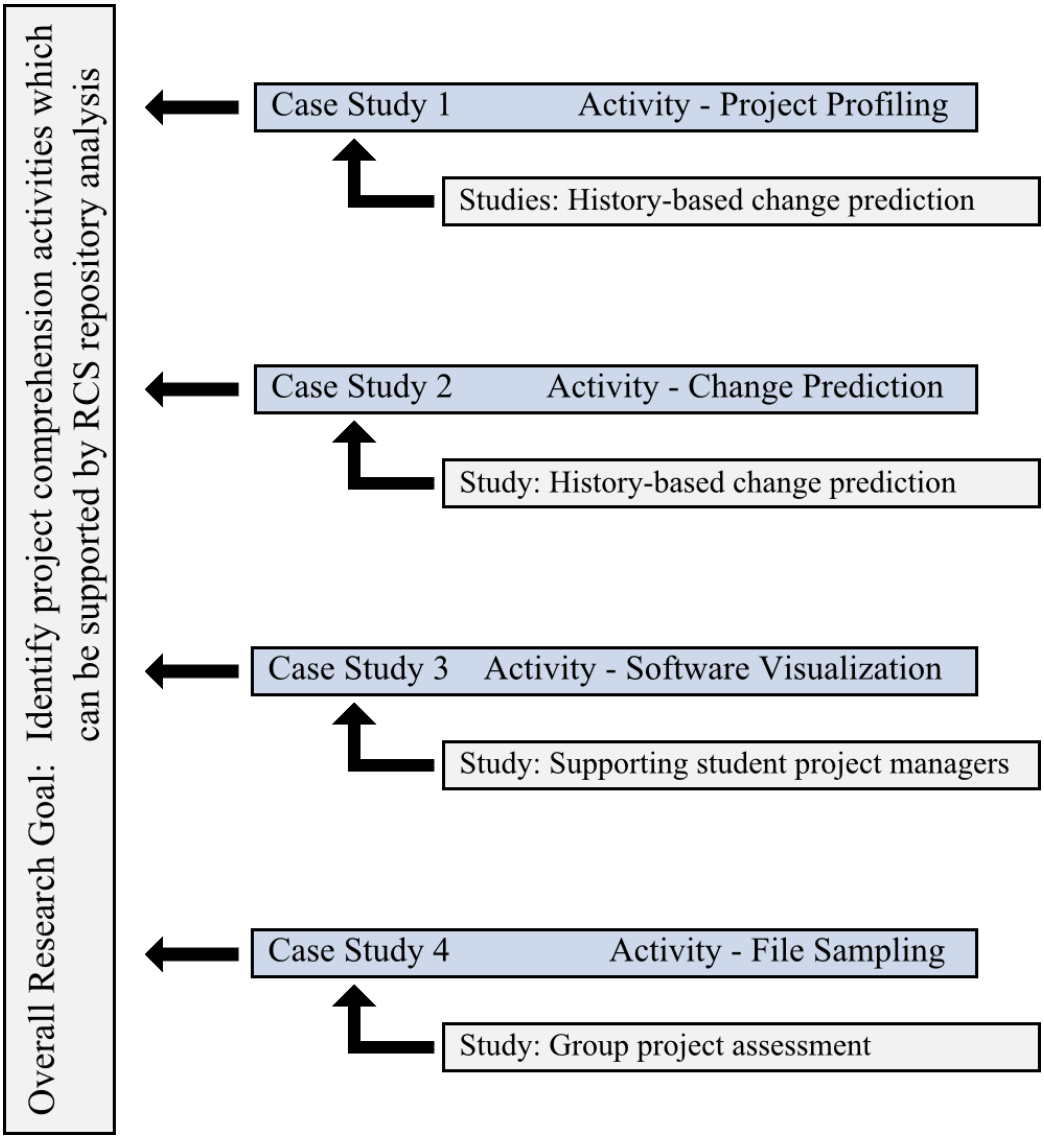


Figure 1.1: Each case study evaluates an RCS-based technique to conduct a project comprehension-related activity to assess whether repository analysis can successfully support project comprehension activities.

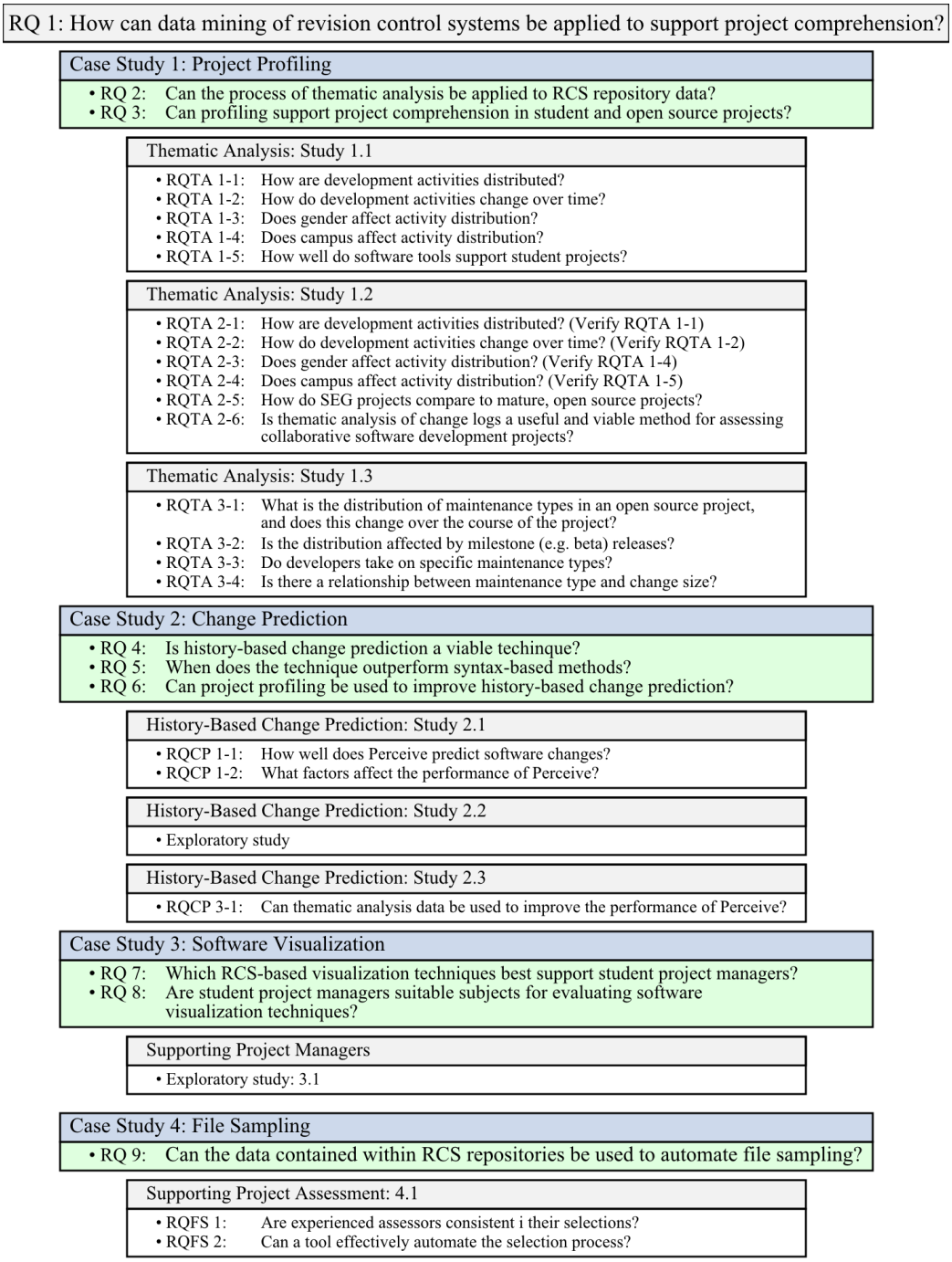


Figure 1.2: Structure of the research questions and studies

- Determine the goals the evaluation addresses;
- Explore the specific questions to be answered;
- Choose the evaluation paradigm and techniques to answer the questions;
- Identify the practical issues;
- Decide how to deal with the ethical issues;
- Evaluate, interpret and present the data.

The following sections provide an overview of the four case studies, their goals and the research questions they seek to address. The research methods, metrics and evaluation techniques are described in the relevant chapters.

1.2.1 Profiling Projects

Examining a project to profile or categorize it – or aspects of it – has several applications in project comprehension, either in terms of increased understanding of the subject project, or to learn more about the nature of software development. For example, Hindle et al. (Hindle et al., 2008) have used profiling to explore the nature of large revisions; while the results are directly applicable to the sample projects, they use the conclusions they draw to teach us about the wider role of large revisions.

The first case study has the goal of determining how analysis of repository change logs can be used to support project comprehension, in terms of effectiveness, cost and application scope. The case study described in Chapter 5 uses thematic analysis (Braun and Clarke, 2006) to profile a series of projects to examine the distribution of maintenance activities; the study seeks to investigate the development processes and management of student group projects, and how the technique can be applied to the assessment process, but it also aims to explore the application of thematic analysis to software projects in general.

Successfully using thematic analysis to profile software projects will demonstrate the technique as a powerful tool in project comprehension, both in terms of understanding individual projects and software development as a

whole. This study aims to answer the following questions:

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data?
- *RQ 3*: Can profiling support project comprehension in student and open source projects?

1.2.2 Change Prediction and Impact Analysis

As projects grow in size and complexity, developers and maintainers find it increasingly difficult to predict the impact of a change to the project, whether the change will require further changes, and whether the developer has missed a necessary change. Current techniques to support change prediction can use knowledge of a programming language's syntax to detect links between code entities, so that a change to one will imply a necessary change to another; an overview of these processes and techniques is given in Section 2.5. Research into the concept of using historical project data such as RCS transactions, detailed in Section 3.3.3, has shown that the technique is viable and can be made to perform sufficiently well to augment syntax-based methods.

The second case study is conducted with the goal of determining if transactional data from software repositories is sufficient to perform change prediction. Chapter 6 reports a series of benchmarking studies on a set of student and open source projects, followed by a detailed exploration of the technique on one student project and one open source project. The case study aims to confirm current research into history-based change prediction, while further exploring how the technique can be applied to different tasks and project types; it also seeks to identify areas in which it outperforms syntax-based techniques, and thus show how a hybrid technique might provide better change-prediction than a syntax- or history-based technique in isolation. Finally, the case study evaluates a history-based change prediction model augmented with data from thematic analyses to determine whether maintenance activity data can be used to improve prediction performance.

The case study reported in Chapter 6 seeks to answer the following research questions:

- *RQ 4*: Is history-based change prediction a viable technique?
- *RQ 5*: When does the technique outperform syntax-based methods?
- *RQ 6*: Can project profiling be used to improve history-based change prediction?

1.2.3 Software Visualization

Software visualization is the process of creating a mental model of software or of a particular aspect of that software (Price et al., 1998). Section 2.6 discusses software visualization and examples of software visualization tools, exploring the lack of an empirical foundation and the lack of evidence-based research in the field.

The third case study has the goal of determining if the data contained in RCS repositories can be visualized to support project managers. Chapter 7 describes a study in which a project comprehension tool including a suite of visualization tools is provided to student project managers to support their roles. The case study seeks to investigate which visualization techniques based on repository data mining are successful, both in terms of performance and adoption, and addresses the following research questions:

- *RQ 7*: Which RCS-based visualization techniques best support student project managers?
- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques?

1.2.4 Sampling

As projects increase in size, with thousands of code files, millions of lines of code and gigabytes of additional assets such as graphics, documentation and data, it becomes more and more necessary to be able to identify a

representative subset of a project. For example, calculating complex metrics on an entire project can be computationally intensive and time-consuming, while processing a smaller sample of the project can rapidly produce accurate results *if the sample is sufficiently representative of the project as a whole.*

The fourth case study has the goal of exploring how successfully an automated tool can select a representative subset of files from a software project. Chapter 7 details an experiment in which a model using transactional data from a project's history is compared to models used by experts in process of selecting files from student group projects for assessment. The performance of the models in automating a currently challenging aspect of assessment is measured to determine if an automated process can successfully identify a subset of a project which is representative of the whole. A successful model will have an impact in both education and industry, as there are a number of activities in which generating a representative sample of a project is either beneficial or necessary.

This study seeks to answer the following research question:

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling?

1.3 Summary

This chapter has laid out the context and aims of this research and described the case studies with which these aims shall be approached. Each study is designed to answer specific research questions in order to achieve a stated goal, but also to explore the viability of the research methods and data mining techniques used in them, and to provide support to future empirical work in this field.

Chapter 2 contains a review of the literature of the fields in which this research is grounded. It is followed in Chapter 3 by a structured literature review of research concerned with data mining and analysis of revision control system repositories.

Chapter 2

Literature Review

2.1 Introduction

Areas of research very rarely stand alone and project comprehension is no exception. It is related to a number of other fields – including software evolution and visualization – and this chapter provides a background for each of them, as well as describing the relationships between them. A great deal of research has been carried out in these fields and a number of tools, models and techniques exist to apply this research. This chapter explores this research in depth, identifying and evaluating the tools and methods used to support project comprehension.

This chapter is divided into four main sections – *program comprehension*, *software change*, *software visualization* and *impact analysis*. Program comprehension sets the stage for the fields that follow, and various models of how developers understand software are described. This leads into the software change section, which discusses the inevitability of evolution and maintenance in software systems. The various difficulties and problems this presents to project managers, developers and – especially – maintainers are described. Software change leads on to research into impact analysis, change propagation and change prediction, all concerned with the understanding of how a change to a project – whether in the requirements or in the code – will generate other

necessary changes. Finally, software visualization – in the context of program comprehension – is described, and research covering a number of tools and methods for supporting software change and change prediction is explored.

Chapter 3 follows this literature review with a structured literature review of the primary field of this research – mining revision control systems.

2.2 Overview

One of the immutable laws of software is that *it changes*. As it does so it almost invariably grows larger and more complex (Lehman, 1996). A typical software project can consist of tens or hundreds of thousands of lines of code, far beyond the capacity of even an experienced developer to maintain a complete understanding of. These factors give rise to several fields of software engineering, including:

- *Software Evolution*: "All programming activity that is intended to generate a new software version from an earlier operational version" (Lehman and Ramil, 2000).
- *Software Maintenance*: "The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" (IEEE Standard 610.12-1999).
- *Program Comprehension*: "The task of recapturing the abstract design of a system, in part or in full, from its source code" (Paul et al., 1991).
- *Impact Analysis*: "Estimating the potential consequences of carrying out a change" (Ajila, 1995).
- *Change Propagation*: "As developers modify software entities... they must ensure that other entities in the software system are updated to be consistent with these new changes" (Hassan and Holt, 2004).

- *Software Visualization*: "The use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software" (Price et al., 1998).

This research is centered around the concept of *project comprehension*, which is related to program comprehension, but is not focussed exclusively on capturing information solely from source code, but from other project data and artefacts.

2.3 Project Comprehension: Understanding Software

The human brain is an astounding thing, capable of feats that the most powerful computers and most elegant software in the world cannot hope to imitate, but it has its limits. These limits are unfortunately evident in the world of software development. Working on a small piece of software, in the order of thousands of lines of code, it is not hard for a developer to construct and maintain a perfect understanding or mental model of the code, how each of its elements work together, what each section does, and what the effects of changes or additions may be (Jørgensen and Sjøberg, 2002). This perfection can quickly decay however, in a number of ways. Simply leaving the code and returning to it weeks or months later can greatly reduce this level of comprehension. Adding another developer can lead to confusion and conflicts Cubranic and Murphy (2003), while increasing the size can exceed the brain's ability to maintain a good understanding of the whole. Unfortunately, in real life development all of these situations can and do happen (Jørgensen, 2006). A million lines of code, being worked on by a development team is beyond the human mind to fully understand.

Much research has been carried out to determine how developers and programmers understand software (Storey et al., 1999). Following observa-

tional studies of programmers various cognitive models have been proposed to describe their behaviour (Storey et al., 2000). These models include the following:

- *Bottom-up*: Programmers read the source code and mentally form higher-level understanding of the software by grouping code together.
- *Top-down*: Rather than working from the code up, programmers apply knowledge of the application domain to a mapping of the source code. It has been observed (Soloway and Ehrlich, 1989) that this model is applied more when the application domain is familiar to the developer.

Other models use a combination of these two approaches, acknowledging that developers are flexible and capable of altering their approach depending on factors such as the depth of understanding required, the resources available and existing familiarity.

A number of tools exist to aid developers in attaining program comprehension; software visualization tools, by definition, are to some extent program comprehension tools (see Section 2.6).

Program comprehension has special importance in the field of software maintenance (Section 2.4) – maintenance activities are frequently assigned to teams with no prior experience of the project at hand. They are required to gain a familiarity with the software sufficient to carry out non-trivial maintenance tasks on it without degrading the codebase. Given that a typical software project consists of many thousands – possibly millions – of lines of code (for example, Microsoft Windows Vista has in the region of 50 million lines of code (Lohr and Markoff, 2006)) it is impossible for a single person to have a complete understanding of the whole system, even a developer who has worked on it since the beginning. The situation is much worse when the developer in question has little to no experience with the software, as is frequently the case during software maintenance, which is usually carried out by a separate team (or an individual). Maintainers may have formal training, but this rarely makes up for their lack of experience with the software

itself (Jørgensen, 2006). This is typically compounded by frequent lack of access to the original developers, poor documentation, poor comments, badly formatted/styled code and previous maintenance efforts.

It is unsurprising therefore that a maintainer given a complex system and limited resources has difficulty in carrying out maintenance tasks without negatively impacting the system in unpredictable ways. Any tool or model which is capable of assisting a maintainer in developing a more complete and accurate understanding of the software is invaluable in reducing the need for future maintenance tasks.

2.4 Software Change – Evolution and Maintenance

"To most people, software is the code that is the end result of the software development process," (Reiss, 2001). This view of software is not limited to end users, but an unfortunate number of developers and managers see the actual implementation of software as being the most significant aspect of development, and therefore dedicate the bulk of the available time and resources to coding. This is counter-productive as better specification and design lead to greatly reduced implementation time, easier testing, and much better maintenance in the future (Bennett and Rajlich, 2000). Although the situation is improving thanks to software engineering maturing as a discipline the development of software, particularly of large systems, is still problematic and badly understood (Jørgensen, 2006).

2.4.1 Software Evolution

Software evolution lacks a standard definition. According to RISE (Research Institute for Software Evolution) it is defined as "the set of activities, both technical and managerial, that ensures that software continues to meet organizational and business objectives in a cost effective way" (Burd and Munro,

2003). Another description (Bennett and Rajlich, 2000) notes that evolution and maintenance are often used interchangeably, although evolution refers to a specific phase in development. A third definition (Girard et al., 2004) states that "evolution subsumes a range of activities from realizing new or changed requirements down to fixing small bugs in the code". Girard et al also make an interesting distinction – "The key difference between the development of a product from scratch and the evolution of a product is that a developer must ensure compatibility with former releases of the same product".

In recent years studies have shown that the changes software undergoes through its life cycle, both before and after delivery, conform to a number of laws (Lehman, 1996) (see Section 2.4.1.1 "Laws of Software Evolution"). This concept has given rise to the practice of viewing software not only in terms of its current state and its goals but in terms of the correlation between the software's development and its predicted development based on the laws of software evolution.

The idea of software evolution is, and will continue to be a key element in improving the state of software development. At the beginning of a project, there will likely be well-defined specifications, designs, constraints and test cases. As the project develops, the code will change and often the other aspects of the software have not changed to match. This means that further along the implementation phase there will be growing inconsistencies between the design, the specification, the implementation and the evaluation (this is obviously not true for all project models – ones that place a strong emphasis on prototyping, for example, will obviously have a different development process). New techniques and tools are being created that tie the evolution process to the rest of the project – a single framework for the whole project ensures that changes in any aspect of the system are automatically reflected in the other aspects. For example, a change in the requirements for the interface of a program will ensure that the developers update the design as well as the implementation. This increased consistency results in an overall improvement in the development process. While it may initially seem to require more effort

and time, the benefits later on greatly outweigh the early costs.

2.4.1.1 Laws of Software Evolution

Although software evolution is a process rather than a problem to be solved, it does generate some significant difficulties which research and tools can help to mitigate. Lehman's research (Lehman, 1996) has uncovered a number of laws and trends that are consistent in any sufficiently complex real-world software project, and as such opens up a whole realm of research into dealing with software evolution. After a series of studies, Lehman formulated the eight Laws of Software Evolution (Lehman, 1996) which apply to E-type programs (those that are "actively used and embedded in a real world domain") and these laws are as follows:

- *Continuing Change*:

  An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

- *Increasing Complexity*:

  As a program is evolved its complexity increases unless work is done to maintain or reduce it.

- *Self-Regulation*:

  The program evolution process is self-regulating with close to normal distribution of measures of product and process attributes.

- *Conservation of Organizational Stability*:

  The average effective global activity rate on an evolving system is invariant over the product lifetime.

- *Conservation of Familiarity*:

  During the active life of an evolving system, the content of successive releases is statistically invariant.

- *Continuing Growth*:

  Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

- *Declining Quality*:

  E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

- *Feedback System*:

  E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

These laws cover E-type systems and were arrived at following observations of a range of systems over a number of years and describe the pressures that dictate the changes that will take place in an evolving piece of software. Although the laws are not based on any statistically significant models the number of studies that have been carried out since, as well as on-going research, mean that "the laws represent an emerging theory of software process and software evolution based on many inputs including the reality of software development" (Lehman, 1996).

Lehman also formulated the Uncertainty Principle (Lehman and Ramil, 2001), which states that "the real world outcome of any E-type software execution is inherently uncertain with the precise area of uncertainty also not knowable". Considering the complexity of much software, and the fact that hardware is not infallible, this principle is incontrovertible. By accepting the truth of the Uncertainty Principle the risks and unpredictability of even the simplest software can be reduced, or at least acknowledged so that the issues may be addressed.

Despite Lehman's work, software evolution is still remarkably unquantifiable. However, there is an increasing amount of work being done on applying metrics to evolving software. Mens and Demeyer break evolution metrics into

two types, predictive and retrospective analysis (Mens and Demeyer, 2001). Predictive analysis uses metrics to assess which parts of a piece of software need to be evolved, which ones are likely to be evolved and which areas may suffer from evolution. Retrospective analysis falls into two categories. Firstly, metrics can be applied to deduce what changes were made, where and when. Secondly, metrics can be used to assess the effects of those changes, in terms of complexity or size for example.

Reiss (Reiss, 2001) discusses three approaches that may be employed to ensure consistent software evolution. The first method is to create a language that covers all aspects of software, from specification to implementation. Forays have already been made into this idea. Projects written in Java can have automatically generated documentation, while recent versions of Borland's Delphi IDE (Integrated Development Environment) are able to create diagrams based on the code being written. However, such an all-encompassing language is likely to fail as there is simply too much to cover and not everything can be represented by a programming language. The second method for achieving consistency is to develop a "semantic representation for software development that handles all the dimensions and maintains their consistency." Reiss argues that this approach is also likely to fail, as it requires re-implementing a range of tools so that they could be aware of all aspects of the software, regardless of methodology, language and notation. The third approach is to develop a mechanism that integrates tools appropriate to the different aspects of software.

This third approach is far more practical as it requires only a framework rather than an entire new system. The benefits of a framework are that new tools, methods, notations and languages can easily be integrated as needed without having to redevelop an entire system.

2.4.2 Software Maintenance

Software maintenance has been defined as "the process of modifying a software system or component after delivery to correct faults, improve performance or

other attributes, or adapt to a changed environment" (IEEE Standard 610.12-1999). Maintenance is one of the most costly and lengthy aspects of software development, often accounting for as much as two thirds of all development effort (Lientz et al., 1978), but maintenance is also one of the lowest-profile and badly supported aspects: "Many personnel believed that maintenance activities have low prestige, are poorly supported by management, and have a low priority at the corporate level" (Dart et al., 1993).

Maintenance is an integral part of software evolution – in fact the RISE definition of software evolution implies that maintenance is the set of activities that make up evolution – since in many instances the majority of work that changes a piece of software takes place in the maintenance phase (Economics, 1981). Not only that, but as maintenance is frequently badly performed, these changes often create the need for more changes.

The distinction between evolution and maintenance is not always clear, but at a very high level, evolution is the changing of software over its life while maintenance is the set of activities that comprise those changes. Even then, whether evolution and maintenance begin at conception or only after initial delivery is still debated. In this research, both maintenance and evolution are defined as beginning at the first point of release.

Software maintenance can be divided into four main types, namely perfective, corrective, adaptive and preventative (BS ISO/IEC 14764:2006), as follows:

- *Perfective maintenance* involves adding or improving functionality,
- *Corrective maintenance* is fixing errors or bugs in the software,
- *Adaptive maintenance* is updating the software to meet a changed environment, such as a new operating system,
- *Preventative maintenance* is altering the software to facilitate future maintenance efforts.

2.5 Impact Analysis and Change Prediction

When a developer makes a change to a piece of software, that change can have knock-on effects. Changing a variable from an `integer` to a `double` requires the developer to ensure that each reference to that variable is expecting a double, otherwise the program will fail. While modern compilers can catch simple cases sometimes the effects are more subtle, such as the problems introduced by inheritance and polymorphism. If the change is made and the program compiles it can be tempting to assume that the change had no unintended effects, but this is rarely the case, especially in larger, more complex systems. The more experience a developer has on the project being modified the more likely he is to anticipate the necessary changes, whereas a maintainer, new to the system and still trying to understand the code, would be unable to anticipate the effects. In this case he would either have to review the code by hand, carry out strict and thorough testing for each change, or use tools to analyse the code for him. Taking a more project-wide view, the impacted area may not even be in the code – for example, a change to a piece of code may require an alteration to the documentation or test cases.

Impact Analysis is the term for anticipating the effects that a change will have on the software. Developers carry out mental impact analysis every time they make a change, no matter how trivial, but there are limitations to their capacity for this. To this end, a number of tools and techniques exist that analyse code and perform a more algorithmic impact analysis.

Ajila (Ajila, 1995) cites a number of reasons for carrying out impact analysis:

- Estimating the cost of a change; it may become necessary to attempt a different, *safer* change.
- Identifying the parts of software that *must* be modified, and the parts of the software than *may* be modified for a given change.
- Recording the history of change information and evaluating the quality of changes.

- Determining the sections of the software that must be tested after a change has been made.

Even if an impact analysis is successfully carried out and every effect discovered, dealing with these effects will frequently require further changes, each of which may require more changes, and so on. This is an example of change propagation and is known as the ripple effect (Black, 2001). Efforts are usually made to reduce this effect during design time by compartmentalizing functionality, limiting the interactions between these blocks, a process for which object-oriented programming is well-suited. However, case studies have shown that up to a third of all requirements are only discovered during the development process (Rajlich, 2000). This means that, despite new programming paradigms and the best efforts of designers, change propagation is still a very real issue.

The typical approach to impact analysis is to look at the source code and apply syntactic knowledge of the programming language to determine which other areas of the code will be affected by a change to any given entity. For the remainder of this thesis such techniques shall be referred to as "syntax-based" techniques. Program slicing[1] is one such method, which aims to extract from a program statements which are relevant to a particular computation (Weiser, 1979). Another method is to create a dependency graph (a graph relating functions and methods) and use it to determine which entities have a connection to the entity/entities being changed. Several such methods and tools for carrying them out are described below.

2.5.1 Syntax-Based Impact Analysis

Program slicing was first introduced in 1979 by Weiser (Weiser, 1979); in the decades since then numerous new methods of program slicing have been developed (Xu et al., 2005). There are several types of slicing: executable, non-executable, forward, backward, static and dynamic to name a few. The

[1] A broad term, as there are a large number of slicing techniques available.

different types of slicing produce different sets of code. Some methods are more complex than others, some produce larger outputs, while some output more easily comprehensible code.

An example of a tool for program slicing is the Surgeon's Assistant (Gallagher, 1996), "a CASE[2] tool that uses decomposition slicing[3] to assist maintainers in limiting the scope of changes". It works on ANSI C programs, but the concept is easily extended to other languages. In itself, the Surgeon's Assistant allows the user to select one or more variables, and it will output a text-based display of the decomposition slices. The Decomposition Slice Display System is a component of the Surgeon's Assistant which creates a visual display of the results, in graph form. It deals with the issues of scaling by allowing the user to collapse regions of the graph. However, the visualization was seen as often being too complex, so a new version was designed (Hutchins and Gallagher, 1998) which offered a new type of visualization. This one was grid-based, with an emphasis on ordering the components. This ordering allowed a general rule of thumb to be applied – "When making a change to a variable, *look right* to see what variables will interfere with the change, and *look up* to see what variables will be affected by the change".

Ajila (Ajila, 1995) has proposed an "approach to impact analysis of object change", which can differentiate between a variety of change types (e.g. deletion, variable redefinition, merging, etc...), and operates over four separate life-cycle phases – requirements, specification, design and implementation, with views for each phase. For example, the requirements phase deals with changes to segments of the requirements documentation, rather than code. It allows various types of query to be made of the system, the most important of which being the 'WHAT-IF' query, which assesses the impact of a change type being applied to a code entity, *regardless of the actual change*. Query results are given textually, in sufficient detail to allow the user to see what effects there will be, where they will occur and why.

[2]Computer Aided Software Engineering

[3]A decomposition slice captures all computation on a variable.

The creators of the WHAT-IF model admit that their prototype has performance issues, but suggest methods for improving on it (including a relational database and a better graphical interface). Another significant issue with the prototype (only the tool, not the model itself) is the lack of a good visualization. The textual output of queries does not lend itself well to quickly assessing the impact of a change, nor for exploring this impact to follow its 'knock-on' effects.

Chapter 3 describes a number of impact analysis and change-prediction techniques which make use of project development histories to perform impact analyses less dependent on syntax and programming language.

2.6 Software Visualization

In Section 2.3 it was stated that there exist a number of tools and techniques for aiding program comprehension. Many of these fall into the category of software visualization.

Software visualization is the process of creating a mental model of software or an aspect of it (Price et al., 1998). Although the name implies that visualization tools use graphics this is not necessarily the case. Price et al argue that software visualization refers to mental models, as opposed to external pictures – looking at the Oxford English Dictionary definitions of visual, they note that: "The seventh definition suggests the formation of a *mental* image which is not necessarily related to something in one's visual field."

This process can have a number of aims: aiding program comprehension, facilitating code writing or project navigation, to name just a few. From the most basic diagram scribbled on the back of an envelope to a full worldwide, multi-user project workflow visualization system, tools can cover a vast range of complexity, functionality and application.

To further make the case that software visualization may not necessarily require graphics there are a number of text-based tools which many people

(and studies) claim are at least as effective – perhaps even more so – as graphical tools. Section 2.6.3 discusses empirical research into the relative benefits of textual versus graphical systems.

To a certain extent, software visualization is a naturally occurring concept – it is the nature of people to use diagrams, sketches and drawings to aid in understanding, to complement a description or to help solve a problem (Blackwell et al., 2001). Software visualization is merely the formalization of this concept which has in turn led to the growth of an entire field within software engineering. Experts will sometimes create their own representation of software (even if it is nothing more than a sketch on a scrap of paper) to solve a particular problem, moving the argument away from "are visualization tools necessary?" and towards "what visualization tools or techniques are most useful?" The issue is one of user skill level and experience; tools move experience into an automated form and can result in a cognitive reallocation of skills and functions.

2.6.1 Types of Visualization

Visualizations can be primarily divided into algorithm visualizations (AV) and program visualizations (PV) (Mulholland, 1993; Lanza, 2003). PVs represent the code or the architecture itself, while AVs use a higher level of abstraction to display the workings of an algorithm. Lanza notes that AVs have become less common and less important in recent years due to the increase in shared, reusable code libraries which has "shifted the focus away from the implementation of such algorithms". Therefore, the term software visualization will be used here to mean PV.

Both PVs and AVs break down into two types, static and run-time (Anslow et al., 2004). Static visualizations are derived from the source or compiled code and typically centre on the structure and hierarchy of the code, the variables, functions, inheritance and dependencies. Run-time visualizations are created by analysing the software as it runs and gathering data from the way the program behaves. This can include such things as execution time,

resources used, variable assignments and function calls. PVs typically use data extracted from the source code and are therefore usually static, while AVs tend to follow an algorithm as it executes and are usually dynamic.

Both types of model have extensive uses. For the most part, static representations are more useful for program comprehension, while run-time visualizations are well suited to debugging and optimizing – "statically extracted visualizations are wide but shallow, while dynamically extracted visualizations are narrow but deep" (Pacione et al., 2003). That is not to say that there is no cross-over between the two – for example, a tree of function calls derived at run-time can be a very powerful method for understanding the way the software works, while a dependency graph created from source code can be just as effective at uncovering ways of streamlining the code as a run-time visualization.

Mulholland (Mulholland, 1993) explains that there are four ways in which a visualization tool can affect a developer. They are as follows:

- Firstly, the nature of the tool affects the way the developer uses the tool itself – is it a graph, a chart, a 3D model? The developer may also decide that the tool would perform better if it presented its data in a different manner.

- Secondly, the visualization may lead to changes in the source code of the software – the tool may highlight a particularly inefficient piece of code, or a dependency graph may prompt a change of the structure of the whole system.

- Thirdly, the tool may alter the nature of the task, change the problem that the software is to solve, or change the way the software solves it. This is particularly true if the aim of the visualization tool is program comprehension, as the developer may then take a different approach to finishing the task.

- Finally, the tool can affect the developer's knowledge, both in the immediate case of the software and the task, and in a more long-term

> change in the developer's understanding of the language being used, or their field of work.

Mulholland also notes that the effects can work both ways – the code, the task and the developer's knowledge can all affect the visualization tool as well – for example, the developer may be using a standard tool that is applied to all of their projects, but then discovers that the tool may be altered to better suit the nature of the current project.

This leads to the discussion as to what visualization tools are appropriate to a task, how many should be used, and how much difference familiarity can make. Some believe that different visualizations are useful at different stages of development or for different tasks, and therefore advocate familiarity with a range of tools (Petre et al., 1998). On the other hand, Mulholland notes that while different representations may be more appropriate for different tasks, it may be preferable that "a 'best bet' representation be used throughout in order to remove the necessity to translate between a large range of representations".

While use of a "best bet" tool reduces the usefulness of the visualization – either a generic one is chosen which will do many things but not brilliantly, or one specific to a task that will do it well, but leave other tasks without suitable tools – it reduces the time necessary to familiarize oneself with new tools. This is not simply a case of learning how the tool works, but also of learning how it applies to the software being analysed. With common use a developer gains a much deeper, more intuitive understanding not only of the software but of how the visualization tool reflects that software.

2.6.2 Software Visualization Tools

There are a number of methods for graphically modelling a software application. Dependency graphs (Horwitz and Reps, 1992) (for example, Figure 2.1) are a common methods of software visualization and highlight both the immense advantages and severe shortcomings of software visualization. They work by representing the source code as a collection of nodes and edges,

where nodes are divisions of the code (files, objects, functions or classes, for example) and edges represent links between these divisions (Ferrante et al., 1987). Such a graph is useful in that it shows the structure of the software on one diagram and can easily show bottlenecks, uncover problem areas of code or simply highlight bad design (Balmas, 2003).

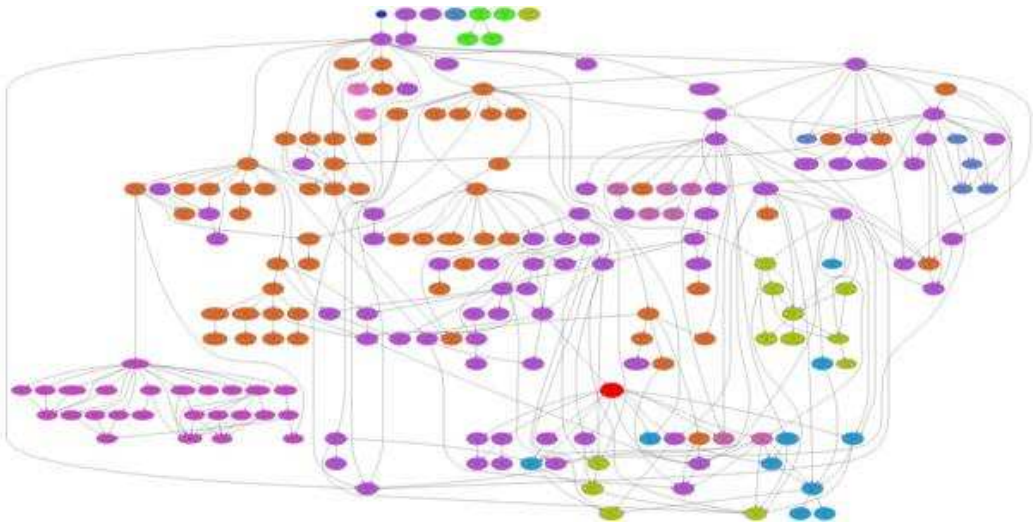


Figure 2.1: Dependency Graph

However, dependency graphs also suffer from drawbacks. For example, while they make it possible to effectively analyse a variety of aspects of a piece of software, they quickly become too big and complex to draw, let alone understand. Balmas (Balmas, 2003) describes an experience where he was unable to use a popular and powerful graph drawing tool, dot (Gansner et al., 2002), to "draw the dependence graph for a 3000 LOC program with 1700 control dependencies and 2000 data dependencies" on a reasonably powerful computer. To solve this problem a number of solutions have been presented, including splitting the model into separate views (Knight and Munro, 1999) or by varying the level of detail presented to the user (Balmas, 2003). Marcus et al (Marcus et al., 2003) explore the idea of using 3D visualizations instead of 2D and discuss the advantages of such a system.

Other tools which seek to address this problem are SeeSoft (Eick et al.,

1992) (see Figure 2.2), which visualizes individual lines of code as small lines of pixels whose lengths reflect the length of the line of code, using colour to represent information for that line, and sv3D (Marcus et al., 2003) (see Figure 2.3), which uses the SeeSoft metaphor but replaces graphs with colour and pixel maps in 3D.

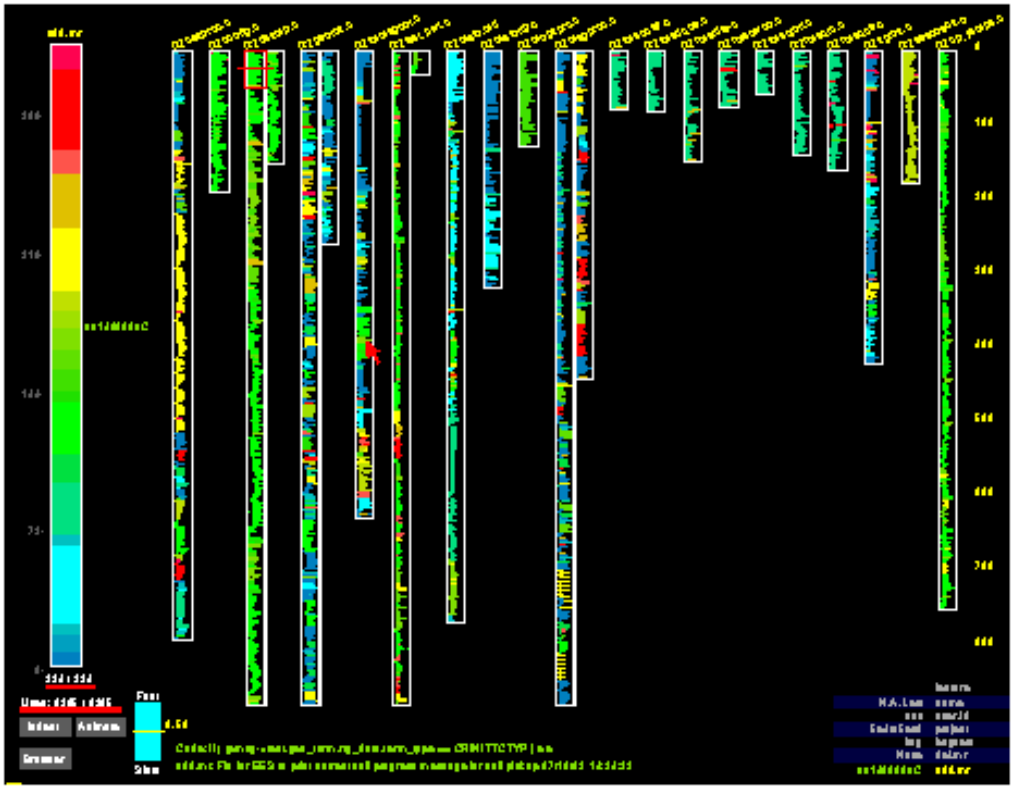


Figure 2.2: SeeSoft

2.6.3 Empirical Studies on Software Visualization

Empirical research into software visualization takes two forms. Firstly, there is fundamental research into whether or not visualization systems are superior to textual systems, and secondly there is research into specific techniques or tools.

There is little empirical research into the fundamental application and

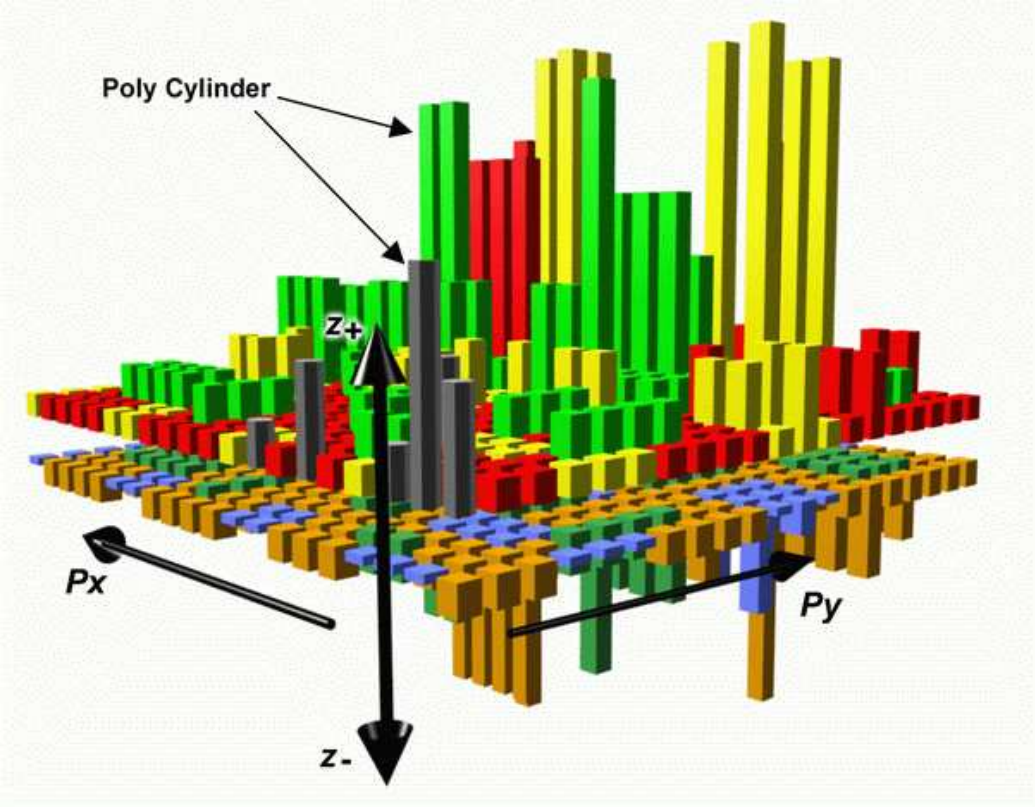


Figure 2.3: sv3D

adoption of software visualization. In his 2002 meta-study of software visualization effectiveness (Hundhausen et al., 2002), Hundhausen discusses the problems of "orphan systems" and "system roulette" – citing early research from Price et al. (Price et al., 1993), he notes that while more than a hundred visualization prototypes had been built by that time, only a particularly small number were in production use. He ascribes the orphaned systems to "system roulette", a development strategy by which technical challenges and a desire for innovation, rather than serving genuine needs, are the motivating force behind system design. In 1993 he conducted a study (Hundhausen, 1993) into user perceptions of a visual debugging system, and found that "given the choice between (a) textual debugging ... and (b) LENS (Mukherjea and Stasko, 1994) debugging ... programmers will choose the former." The LENS system "occupies a unique niche" (Mukherjea and Stasko, 1994) – the desire for novelty was the driving force behind the development rather than the desire to address a problem or support a task.

In the early 80s Shneiderman (Shneiderman, 1982) conducted an experiment to determine if graphical documentation of the source code improved understanding of that code. Groups of subjects were given the same source code and a variety of documentation types (including pseudocode, data structure diagrams, control flow information and macro flowcharts) and asked a series of questions on the code. For the test code, the subjects found data structure more useful than control flow data, but the format (textual or graphical) made no significant difference. This reinforced the results of an earlier experiment which again showed no performance benefit from a graphical format over a textual one (Shneiderman et al., 1977).

There is a marked tendency in research and systems development to assume that visualization techniques are inherently useful, that "accepted wisdom" is correct. For example, a systematic literature review of the UML (Budgen et al., 2010) found that the overwhelming majority of empirical research addressed extensions, variations, metrics and adoption, without examining if the use of UML was innately appropriate or beneficial, to any

class of user or task.

A systematic literature review of the software visualization (Burn et al., 2009) empirical landscape has shown that, in comparison to other software engineering fields, software visualization has received very little empirical research. From the initial mapping study based on titles and abstracts, 37 papers were included and classified. Of these, the 15 papers related to structure visualization were examined in detail; all but five were rejected as they did not report sufficiently empirical research. The remaining five studies had no replication or reinforcement, making the conclusions difficult to accept with any confidence. Only one of these studies examined the use of visualization compared to the absence of visualization, and found that it was indeed beneficial; the other studies began with the assumption that visualization was beneficial and studied the use of tools. One other paper compared actual visualization techniques, rather than the tools themselves, while the remainder were testing tools and implementations.

The visualization systematic literature review demonstrates that the findings of early research conducted in the 80s and 90s has not been contradicted since, and there is little supporting evidence to show that visualization techniques should be simply accepted as a solution without consideration of the task, user or requirements. More recent research has in fact supported Shneiderman's early work, such as Krinke's conclusion that as visualization techniques often "suffer from the sheer amount of data to be visualised", a text-based approach is often preferred as programmers are more accustomed to extracting information from large amounts of text (Krinke, 2004).

In conclusion, software visualization tools are commonly used to support project comprehension; however, research has shown that software visualization systems, especially academia-driven systems, tend to be innovation-driven rather than problem-driven, and the majority of systems will never see production use. Even the scarce evidence-based research into the appropriateness of visualization systems has shown that they are not automatically the best approach.

2.7 Summary

Research surrounding project comprehension and related fields has been discussed and evaluated, providing the context of this dissertation. Chapter 3 continues the literature review with a structured analysis of research into the field of data-mining revision control systems and subsequent analysis.

Chapter 3

Mining Revision Control Repositories

3.1 Introduction

Chapter 2 discussed the research covering program comprehension, software change and software visualization, setting the context for this research. This chapter provides a structured analysis of the research surrounding the mining and analysis of revision control systems.

The research presented in this thesis is primarily concerned with the data-mining and analysis of historical data contained within revision control repositories to support traditional methods of project comprehension, with an additional focus on application in an educational context. Section 3.3.2 discusses current techniques for mining historical data for a variety of tasks; Section 3.3.3 examines research in using historical data to perform change prediction and impact analysis and Section 3.3.5 explores research using RCS data mining in educational contexts.

3.2 Revision Control

An integral element of a software development project beyond a certain level of complexity – especially in team-based or distributed environments – is that of source control. Examples of such tool include CVS (Beck, 2003), SubVersion (Collins-Sussman, 2002), BitKeeper (Henson and Garzik, 2002), Mercurial (Mackall, 2006) and Git (Swicegood, 2008). CVS and SubVersion, are centralized repositories, while BitKeeper, Git and Mercurial are decentralized. Requirements for revision control systems have changed over time, as more modern systems are placing more emphasis on scalability and decentralization. However, the central functionality remains the same – to allow multiple developers to work concurrently on the same project, to provide version management, and to allow code to be reverted to previous versions.

The filesystems underpinning revision control systems typically use file deltas, storing only the differences from one version of a file to the next; this works well on textual files, but less effectively on binary files such as compiled executables, images or word-processing documents. At the same time that modern systems are incorporating additional functionality to better process these formats, other developments such as XML-based documents are helping to mitigate these drawbacks (Rönnau et al., 2005).

3.3 Structured Review

Chapter 2 contained a review of research and literature in the fields surrounding this research; this chapter consists of a more structured review. While it is not a complete systematic literature review (SLR) as described by Kitchenham et al. (Dybå et al., 2005), a more thorough data extraction process is applied than an informal process, which allows for a deeper analysis of trends and patterns in the research. The data extracted includes the following:

- *Year*: Software changes very quickly, which has an effect on the research taking place, such as the systems being examined and the nature of the

objects under test.

- *Topic*: There are a number of sub-topics in this field; the data extraction categorizes each paper by the theme or area of research.
- *System*: As previously described, there are a number of revision control systems in use, some proprietary, some open source, and each has advantages and disadvantages in comparison to others.
- *Domain*: Whether the research covers proprietary systems, FOSS (Free, Open Source Software) or educational systems or environments.
- *Test Objects*: In a more human-centred field this item would be "subjects", but here the subjects tend to be software projects.
- *Description*: A summary of the aims and method of the research
- *Conclusions*: The outcome of the research, the evaluation of the results and any challenges or further questions proposed.

3.3.1 Overview

Using a combination of automatic and manual searches, existing domain expertise and iteratively examining references, 48 studies were identified for inclusion (a duplicated study was later removed from this list, leaving 47 references). Appendix A shows the complete list of the results from this review. In the "System" column, *Multiple Sources* refers to revision control systems, bug tracking software, mailing lists, forums and documents. This is not necessarily a complete collection of the literature, but represents a comprehensive view of the field. For example, some of the included studies are not concerned with RCS repositories but with version snapshots; while not directly relevant to this review these studies provide useful comparisons with other techniques and frequently draw highly relevant conclusions.

A preliminary analysis shows some basic trends in the field:

- CVS is the most common system for data mining, largely due to its early and rapid rise to ubiquity. Although newer systems have begun to replace it in many environments recently, it remains a common basis for research due to the large amount of freely available data.
- Studies using industrial projects tend to use either version snapshots or proprietary version control systems.
- The most common domain is open source software, due to the large amount of freely available data for use. A sample of open source projects occur repeatedly in different studies – such as Apache, Eclipse, GCC and Linux – which serves to provide a useful context for future research.
- There is significant research in the use of supplementary information channels such as bug tracking software, mailing lists and forums.
- Visualization is often used to support mining and comprehension of repository data.
- There are supplementary research topics, such as the value and drawbacks of the focus on open source software (German, 2004), and supporting acknowledged limitations of repository systems, such as the name identity problem (Van Rysselberghe et al., 2006) or lack of user experience (Thomson and Holcombe, 2008).
- Some of the studies use snapshots of projects rather than mining version control systems, but they are included as their findings and applications can equally be used with version control systems.

The following sections will discuss specific findings and topics within the field of RCS data mining.

3.3.2 Data Mining

A 1997 paper by Ball et al. (Ball et al., 1997) is an early example of mining a revision control system to learn more about a project using the contextual

information it provides. The paper set out a series of proposals for future research, including visualization, time-series analysis, development process investigation and static program analysis, all using historical data. In 2004 Godfrey et al. (Godfrey et al., 2004) highlighted a set of challenges facing the field – scale, automation and syntactic understanding. The recent trend towards using open source software as the basis for research has given rise to a number of papers discussing the advantages, challenges and needs of open source projects in the field of repository mining. Gasser et al. (Gasser et al., 2004) make a series of recommendations to both researchers and the open source community which they claim would be of benefit to both. These recommendations include the development of standards for metadata and the creation of additional instrumentation for current tools. Howison and Crowston (Howison and Crowston, 2004) specifically addressed the problems they faced in mining the open source platform SourceForge, finding that even a development community as open as SourceForge had obstacles which had to be overcome by researchers wishing to access and analyse data.

Exploratory research has been conducted to determine to what extent existing techniques and models can be applied to historical version data. For example, Dekhtyar et al. (Dekhtyar et al., 2004) hypothesized that while most work on mining historical data focussed on analysing software, due to its structured and parsable nature, other text stored in repositories, such as requirements documents, could also be analysed using existing natural language processing techniques. The research showed that such analysis was "not too difficult" and therefore repositories should be augmented with all available natural language text used in development.

Hindle et al. (Hindle et al., 2008) described a study in which large commits – revisions in which a large, anomalous number of changes are made – were manually classified (as shown in Figure 3.1) to investigate how they could affect a project. They discovered that large commits were much more likely to consist of perfective changes than are small commits, which tend to be more corrective in nature.

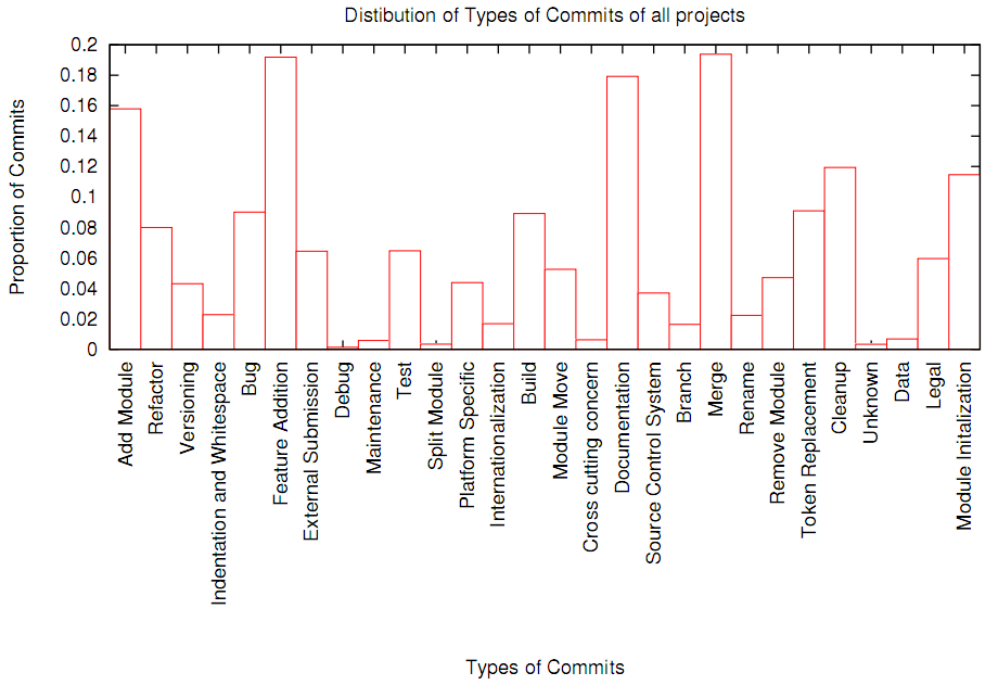


Figure 3.1: A manual classification of large commits

Research by Bachmann and Bernstein (Bachmann and Bernstein, 2010) investigated the relationship between the quality of process data, such as repository comment logs, with the quality of the actual product. Among other findings, they reported that there was a correlation between empty commit messages and both bug report quality and product quality. This evidence of links between human-generated meta-data and the actual project code demonstrate the potential for repository data mining to provide information regarding software development, to reflect on existing work, to support current work and to guide future work.

However, the outcomes of version analysis are limited by the correctness of the data stored. Thomson and Holcombe (Thomson and Holcombe, 2008) conducted a case study in which the code repositories of 17 student teams were analysed to determine what errors – human or technical – occurred. They classified the errors into "type one errors which relate to the non-use of the system; type two errors that emerged from the direct manipulation of the

repository; and type three errors from the limitation of CVS not to record file name changes". The type one errors were relatively common, and is reflected in other research (Glassy, 2005; Reid and Wilson, 2005); both type one and type two errors are more common in student or novice users, and do not appear to occur in more experienced environments. Type three errors reflect a well-known problem, that of the name identity problem (Van Rysselberghe et al., 2006; Gorg and Weisgerber, 2005), which was common with CVS, but has been partly addressed by more modern revision control systems.

Several researchers have discussed further potential problems with the field of version control analysis. As early as 1997 Gall et al. (Gall et al., 1997) discovered that a high level overview of system behaviour could mask significantly different behaviour at lower levels, a finding which must be carefully considered in a field where the large amounts of data generated tend to lead to a compromise between detail and coverage. Despite the common perception that repositories, with the scope of data they store, must allow for models to be developed which can predict a range of factors related to the project, this has not always proven to be the case. Mierle et al. (Mierle et al., 2005) performed an analysis of over 200 repositories of student work, calculating 166 features and metrics, and could find no effective predictor of student performance. Figure 3.2 shows an example correlation of final grade with total lines of code written – a correlation as strong as any other found in the study.

Finally, as with software visualization (see Section 2.6.3) there is little empirical research in the field of RCS mining. Most studies tend to consist of an informal case study of one or more projects to which the tool or technique is applied. Many of the empirical results which are available tend to come from interviews and observations of the target user, such as students or developers. Again, as with visualization, the performance of RCS mining – especially in supporting project comprehension – can be difficult to quantify and to compare to other techniques.

As repository analysis studies are concerned primarily with automated

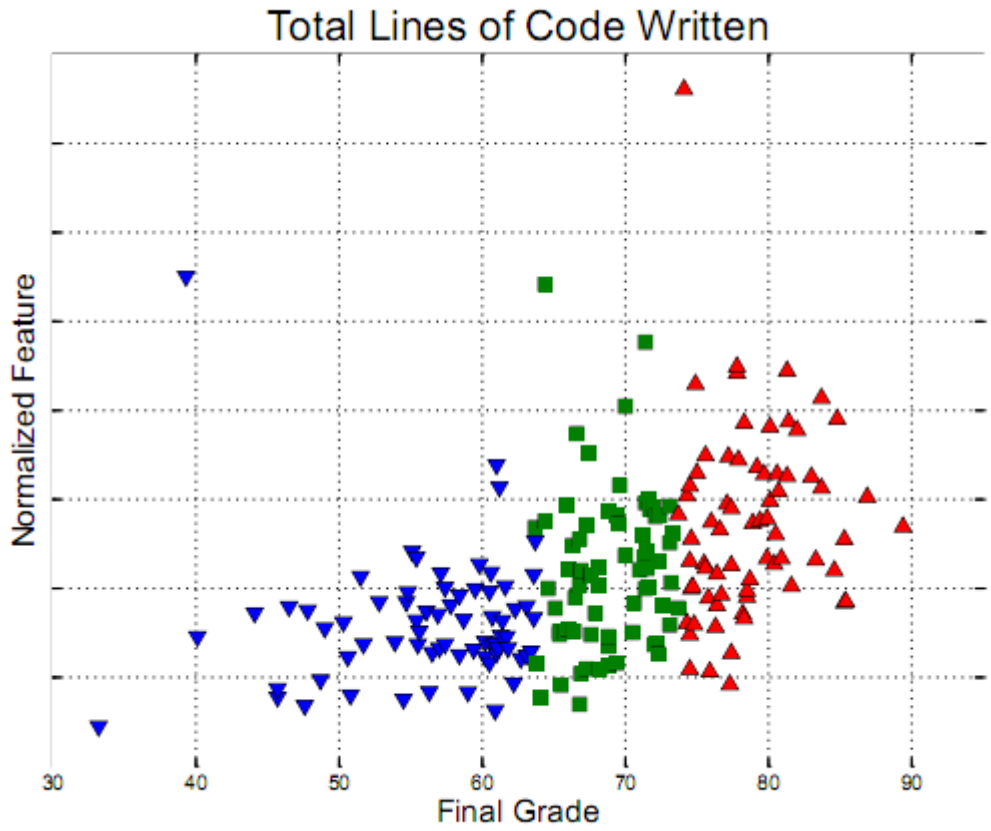


Figure 3.2: Mierle et al. could find no better correlation between a calculated metric and student performance than a simple "lines of code" count

systems applied to digital data, one important aspect of empirical research, that of replicability, should be well supported in this field. However, research by Robles (Robles, 2010) investigated the potential for replicating the studies of repository analysis and found that the replicability of the studies examined – covering six years – is currently very low, proposing a series of good practices to enhance the state of research. For example, the replicability of a study in this field depends on the publication of the raw data used, and the disclosure of any processing performed on this data.

The lack of replicability in a field which is inherently well-suited to it is a further example of the lack of depth and rigour that is typical of the current state of research in many fields of Computer Science.

3.3.3 Impact Analysis and Change Prediction Using Historical Data

The methods of impact analysis and change prediction discussed in the previous chapter (see Section 2.5) used a static analysis of the source code as their basis. Some methods allow the user to create their own rules to reflect their knowledge of the system or the language, while others can do a more dynamic analysis of code at run-time. However, these all rely on analysis of a single version of the code, and learn nothing from past experience. When a developer has been working on code for a long time the level of familiarity and understanding she has attained allow her to perform an almost unconscious analysis – she *knows* what the effects will be. This can be supported by tools such as those described in this literature review, a combination which can provide a very powerful impact analysis.

However, a maintainer new to the system has no such prior knowledge, and has little or no access to the original developers. Instead he has to rely on very rapid comprehension of the code, heavily supplemented by the tools available. While these tools can be powerful and effective, they do have their drawbacks. They can give false-negatives (missing an impact) or false-positives (declaring an impact where none exists). The latter is more common, and while it may

seem harmless can result in wasted time and even needless changes. Studies of various change propagation heuristics have shown that methods based on code structure are not as reliable as once thought (Hassan and Holt, 2004; Bieman et al., 2003).

Another limitation of code-based analysis tools is that they are limited to analysing code and predicting effects solely within that code. However, a change may require a non-code-based entity to be modified, such as documentation. Further to this, code-analysis by definition requires the tool to have some knowledge of the structure and syntax of the programming language in use, which ties the tool to a single language or a set of languages.

To overcome these problems, a second type of analysis is available – mining historical data for patterns. This type of analysis gives results familiar to anyone who has used online shopping sites: "Users who searched for 'x' also searched for `'y'`" or "users who bought 'a' also bought `'b'`". By analysing change records or version histories, a tool can tap into the habits and patterns of the experienced developers and use this data to conduct the analysis. This also overcomes the problem of language dependence – in the case of the tool being applied to a language it has no knowledge of it can fall back to the entities used by the version control software. In this way, the level of detail drops from "Users who changed function 'x' also changed these functions..." to "Users who changed file 'x' also changed files...".

The concept of using historical developer data has been touched upon in past research (Gall et al., 1998), where revision information was used to discover common change behaviour of modules and to identify possible structural changes. Such methods have been shown to be generally superior to code structure analysis (Hassan and Holt, 2004) and yet the concept remains relatively unexplored. Ball et al. (Ball et al., 1997) used modification records to cluster files based on developer behaviour, as shown in Figure 3.3.

One tool that performs impact analysis using version histories is ROSE (Zimmermann et al., 2005), a plug-in for the popular development platform Eclipse. It uses "full-fledged data mining techniques to obtain association

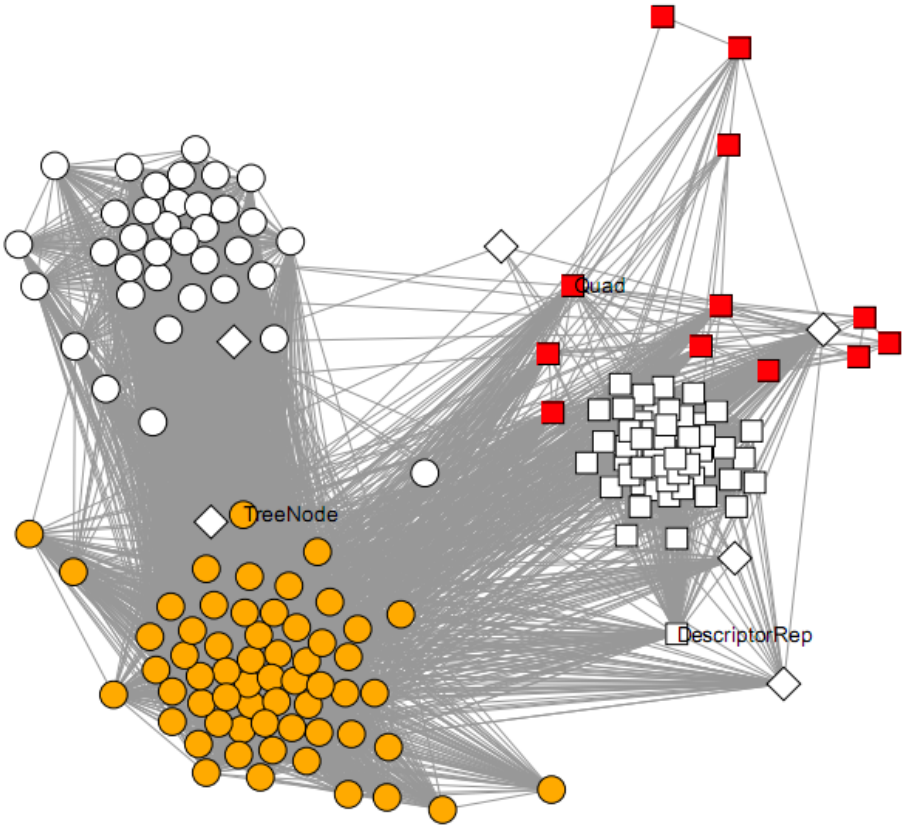


Figure 3.3: Cluster analysis based on modification request relationships

rules from version histories", and produces a confidence level for each impact it finds. ROSE creates sets of transactions where two subsequent changes by the same author are part of one transaction where they are at most 200 seconds apart (Zimmermann and Weissgerber, 2004). Each rule is given a confidence (the proportion of times that changing 'a' requires changing 'b') and a support (the number of times changing 'a' required changing 'b') level which helps to determine how likely the suggested impact is.

Evaluation of ROSE (Zimmermann et al., 2005) has shown that for stable systems (such as developing a mature product, or maintenance) ROSE is useful at a detailed level, whereas for more immature systems (such as those in early development) it is only useful at the file level, as otherwise it would have to predict new functions and entities. However, ROSE is limited in part by the nature of CVS, which does only allows one change per transaction, requiring ROSE to perform preprocessing to make groups of transactions using a "sliding window" algorithm (Zimmermann and Weissgerber, 2004). This grouping process is necessarily imperfect; two transactions made within a short period of time are assumed to be in the same transaction, which is not always the case – they could be part of two separate jobs being carried out back to back. Also, it can fail to make a link between two transactions if they take place too far apart – such as resuming a job after a break or the next day. This is less of a problem with more modern revision control systems, especially when integrated bug-tracking or job-control is employed.

An advantage of ROSE is also a drawback – the speed of rule computation can allow transactions to be added between two situations (a set of changed entities) can cause a false-positive to be acted upon and give it an even higher confidence level in the future (Zimmermann et al., 2005). A more flexible system allowing rules to be computed using restricted sets of data (such as transactions from specific users or data from specific versions) would allow this effect to be minimized. For example, by using only version data from a time when the software was structurally similar to the current version, but before maintenance started may provide better, and less 'corrupted' data.

A recent study (Adams et al., 2010) used clustering techniques to perform "concern mining", the process of automatically identifying concerns where a concern is a "conceptual unit" of code, such as logging or email. Concern mining is an innate process in development, performed almost continuously by developers on a manual basis as they work. The authors developed a new technique called COMMIT to reduce the manual effort involved in identifying concerns using historical code changes; it saw an 87.5% improvement over existing techniques, although the concerns identified did not always overlap, leading to the conclusion that the techniques actually complement one another.

Empirical, quantitative assessment of history-based impact analysis or change prediction is as difficult as it is for existing, syntactic techniques, as a definitive oracle or source of accurate results to use as a baseline is difficult to come by. Experts can manually create such data, but it is a lengthy, resource-intensive process which limits the breadth and depth of the data available. However, as research continues and the corpus of evidence grows, confidence in the veracity of the results will increase, and history-based analysis is growing as a credible technique for change-prediction, able to support and augment current methods. In this context, such techniques and algorithms can be compared with one another and empirical conclusions can be drawn.

3.3.4 Supplementary Data Sources

Revision control systems are not the only source of historical data for a project. Much research has been conducted which combines RCS analysis with other channels of data, such as bug reports, mailing lists, forums and project documentation. Each of these sources contains large amounts of information about a project's history, and analysis of these channels can be used to support results from other channels. Anvik and Murphy (Anvik and Murphy, 2007) developed a system to suggest a set of developers with expertise relevant to a bug report. They performed a study in which they compared the results of their technique with those of experts from the project. Using

information retrieval metrics they found that two source repository-based methods (SR-change and SR-package) had a precision (the proportion of suggested items which are correct) of .59 and .39 and recall (the proportion of the correct answers which are suggested) of .71 and .91 respectively. Analysis of the bug network resulted in a precision of .56 and a recall of .79. According to Hassan and Holt (Hassan and Holt, 2004), such systems should aim to achieve at least "typical information retrieval practical boundaries where precision usually lies in the 35-40% range and recall is around 60%". A tool developed by Cubranic and Murphy, Hipikat (Cubranic and Murphy, 2003) was designed to suggest pertinent artefacts to a new user based on a generated "group memory"; results showed it to be useful for newcomers to a project, successfully suggesting "entry points" to the project based on their tasks.

Gousios et al. (Gousios et al., 2008) use a range of information channels to measure developer contributions as part of a quality evaluation tool, with aims to empirically assess its performance using questionnaires. A more thorough description of the tool (Gousios and Spinellis, 2009) describes an informal case study in which an increase in code submissions are noted following intense discussion.

As already stated, RCS mining techniques are difficult to objectively assess; incorporating data from additional channels provides an avenue for validating the results from repository analysis.

3.3.5 Revision Control in Education

Much of the research into repository analysis is in the domain of industrial or open source software. This is understandable, as FOSS (Free and Open Source Software) makes an expansive quantity of projects freely available to study, while industrial case studies will tend to have a greater impact for practitioners, considering the differences between open source software and commercial software development in terms of motivation, organization and economics (Lerner and Tirole, 2002). However, there is a subset of research taking place in an educational context, where project comprehension is an

important aspect for a range of users – students, teachers, assistants and assessors. As discussed in Section 3.3.2 Thomson and Holcombe (Thomson and Holcombe, 2008) used a number of student CVS repositories to identify and classify errors in CVS data. Much of the problems came from student inexperience with the tools, resulting either in them not using them or corrupting the repositories. They discuss the implications of these findings for future research involving student repositories – "In order to maintain the goodwill of the students it may not be possible to require them to hand in their projects on a more frequent basis, thus type one errors may be hard to avoid." Similarly, in earlier research, Glassy (Glassy, 2005) discovered that it was necessary to "structure the assigned work in terms of concrete milestones, and attach consequences (grade points) to the meeting of those milestones." Glassy also discovered that while the use of RCS in student assignments did create an administrative overhead for the instructor, the benefits – which included teaching students the use of industry standard "best practice", enabling a deeper insight into student work processes, providing additional protection against plagiarism and encouraging students to manage their progress more effectively – potentially outweigh the aforementioned overhead.

Reid and Wilson (Reid and Wilson, 2005) compared a CVS-based submission system to an existing electronic assignment submission system, and found that overall CVS is superior to the previous system for accepting submissions and recommend that it should be used in the future. "It forces students to adopt good working practice ... it makes it feasible for us to assign team projects much earlier ... and it gives the instructors a powerful tool to manage interactions with students, TAs, and each other".

It is tempting to assume that with the amount of data available in student project histories it should be possible to use metrics to predict performance, and incorporate RCS use into the assessment. However, as described in Section 3.3.2, Mierle et al. (Mierle et al., 2005) have shown using a large project set, no effective predictor could be determined. Liu et al. (Liu et al.,

2004) also failed to uncover any measurable predictor of performance, despite anecdotal correlation between some results and students' work. As such, instructors must be careful when incorporating version control into a project in any summative manner – while it does indeed teach good industry practice, there is no evidence that it in itself affects students' performance in other aspects of software development, or that it can be effectively assessed.

3.4 Summary

There has been extensive research carried out in the field of mining revision control systems, and historical data has been shown to provide new approaches to existing problems, with comparable performance and results. Change prediction is one such application, where an approach using historical data can augment – without replacing – traditional, syntax-based techniques. Further research must be done, however, to explore the effects of the algorithms and parameters of such techniques and to investigate how the approach can be applied in different use-cases and to different classes of project.

Similarly, manual examination of historical data, such as that used by Hindle et al. (Hindle et al., 2008), is a promising approach with applications in a variety of contexts. By exploring the viability of such analysis it should be possible to determine if manual classification of change data could be used in an educational setting to allow assessors to better understand and assess a collaborative assignment or project.

This chapter explored in depth the field of mining revision control systems and highlighted some avenues for further research. Chapter 4 describes the design and implementation of the tool which supports the case studies of which this research consists.

Chapter 4

Design and Implementation

4.1 Introduction

Chapter 3 explored past and current work in the field of RCS data mining, providing a context for this research. As described in Chapter 1 a series of case studies are conducted to explore the use of revision control system repositories in supporting various aspects of project comprehension. These case studies require a range of software tools to extract, analyse and present the repository data. This chapter describes in detail the design and implementation of a tool, `Perceive`, which is used in the conduct of these case studies. The tool was designed iteratively, incorporating feedback from users and adapting to changing requirements and environments, and this chapter describes the development process. As well as `Perceive` a number of supplementary tools were developed to perform the data mining and preprocessing; these tools are also described in this chapter.

Revision control systems contain a great deal of data which can be extracted and analysed by appropriate tools. As discussed in Chapter 3 these tools vary depending on the system being used, as does the amount of processing the data will need once extracting. For example, CVS treats each changed file individually, so committing a set of changed files actually results in a series of transactions, rather than one, whereas SubVersion allows each

revision to contain a number of actions.

The projects used in this research all use SubVersion as their repository system and so the tools developed are all implemented to access SubVersion repositories. However, this is simply an implementation decision based on the subject projects, and does not affect the nature of the research in any way.

4.2 Requirements

The research questions stated in Chapter 1 are addressed by the case studies reported in Chapters 5, 6 and 7. This section discusses the requirements that each of these case studies have in the context of tools and software.

4.2.1 Thematic Analysis

The thematic analyses conducted in Chapter 5 require that the author, timestamp and comment for each revision in a project be extracted from the project's repository and presented to the user in a spreadsheet-like format which the user can filter, sort and organize as necessary. For performance and convenience, this will require the data from a repository to be previously extracted and converted into a format which can be more readily presented to the user. Section 4.3 describes the process by which a repository is analysed and the data extracted. Section 4.4 details the implementation of a graphical application which loads the extracted data and provides the user with the revision data and features with which to conduct the thematic analysis.

4.2.2 History-Based Change Prediction

Chapter 6 reports a series of studies in which transactional repository data is processed to create a network of file relationships within a project. This study requires that the user must be able to select a project (or a set of projects), select the model being used, define a set of parameters for that model and begin the process. As with the thematic analyses, this requires the data to be

extracted and stored before processing. However, no user interface is required as the algorithm is non-interactive and simply processes the transactional data to produce tabular output.

4.2.3 File Sampling

The file-selection experiment reported in Chapter 7 required no additional features or data beyond that used in the change prediction case study. Section 7.3.2.4 describes the algorithms which were incorporated into the software; again, these require no user interaction and simply generate tabular results from an input project.

4.2.4 Software Visualization

The software visualization case study described in Chapter 7 requires a more fully-featured application with a user-friendly interface. Section 7.2.2.2 describes how the initial spreadsheet interface was extended to create a project management tool and visualization suite used by student project managers. The software provided to the students contains the revision spreadsheet used by the thematic analyses, but not the algorithms used in the change prediction or file sampling case studies.

The following sections describe the tools and processes in use at each stage – data mining, preprocessing, storage and presentation; the final section describes the complete system and workflow.

4.3 Data Extraction and Storage

The first stage in the process is to extract the data from the RCS filesystem. SubVersion offers three methods for accessing the data:

- *Accessing the Database*: When configuring a SubVersion database, administrators can choose between FSFS (Fast Secure File System) or

Berkeley DB. Using correct tools and knowledge of the systems, data can be extracted directly from the repository.

- `svnlook`: SubVersion is supported by a utility called svnlook which allows users to query repositories for information from the command line, such as the number of the latest revision, the author of a particular revision or the list of changes in a particular revision.

- *Exporting a Log*: SubVersion allows users to export a log of changes, either in a native SubVersion format or in XML.

It was decided that `svnlook` would be the primary method for performing the data mining, as the use of an included tool assures reliability and compatibility. Using the generated log was also considered, but the log does not include as much information as is provided by `svnlook`.

A script was created which would, given a repository and an identifier (typically the project name), extract the required information and store it in a MySQL database. The database stage is useful as it allows highly flexible and rapid querying of the data to acquire metrics and information about projects, and is a common feature in repository mining research.

If the project has previously been analysed (such as in an ongoing student project), the script first determines the last revision processed, and resumes from that point. Each revision from that point to the most recent is then queried using `svnlook` for a list of changes. This produces output in the following style:

```
U   trunk/Project/IO/XMLLoader.cs
A   trunk/Project/Login.Designer.cs
A   trunk/Project/Login.cs
A   trunk/Project/Login.resx
U   trunk/Project/Main.cs
U   trunk/Project/Perceive.cs
U   trunk/Project/Perceive.csproj
```

The leading character shows the change type (in this case either 'updated file' or 'added file') and the associated file. If the revision included a copy operation, it appears as follows:

```
A + tags/2008-07-10/
    (from trunk/:r199)
```

Which indicates a file (or directory in this case) was created as a copy from the indicated file or directory, as it appeared in the given revision (e.g. 199). Finally, a move operation is stored simply as a delete and a copy:

```
D   trunk/IO/
A + trunk/Project/IO/
    (from trunk/IO/:r9)
```

The script processing these revisions makes no attempt to keep track of the move and copy operations – that function is deferred to a later stage.

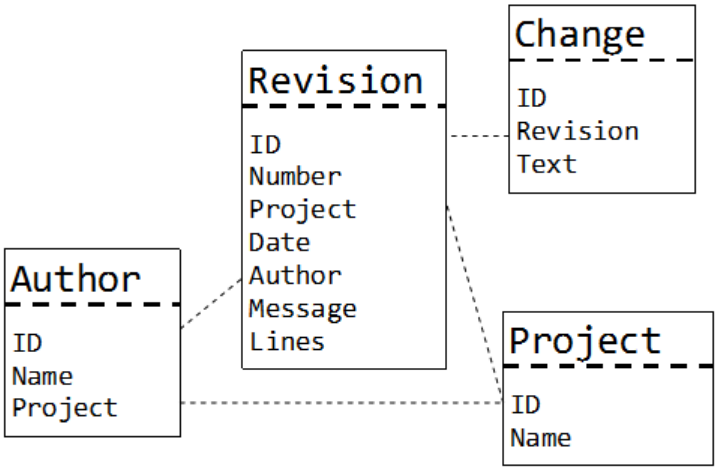


Figure 4.1: The MySQL database design used to store the data extracted from SubVersion repositories

The structure of the database is shown in Figure 4.1. The `project` table is a simple table of names mapped to identifiers, and the `author` table maps users to projects. The `revision` table stores the date, author, number and

project of each revision, as well as the accompanying log message. Finally, the `change` table maps individual changes to revisions. The `text` field of the `change` table stores the output of the `svnlook` commands. The output is rewritten into a mark-up format to simplify future operations, and copy operations are rewritten onto one line. The above example would become:

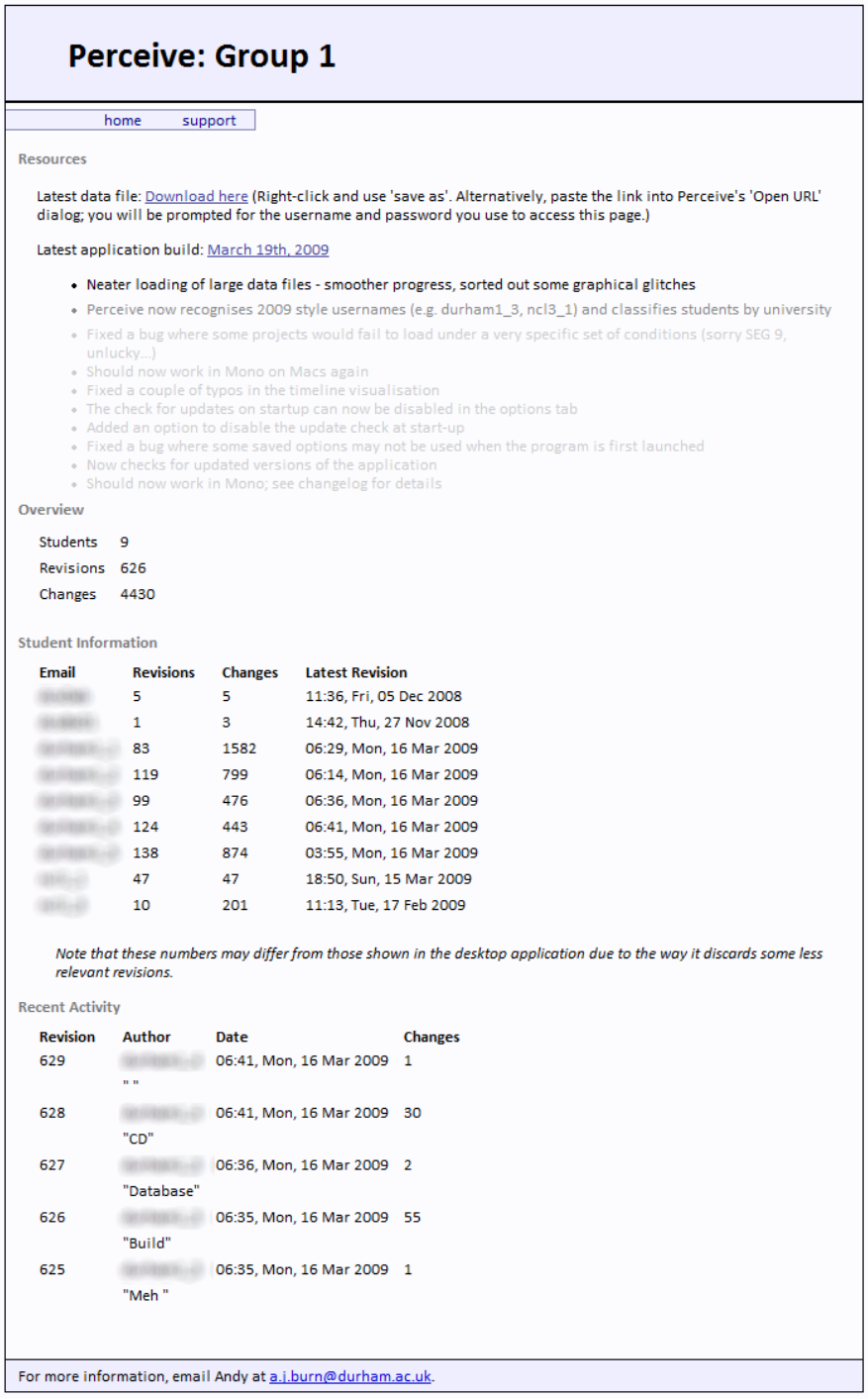
```
change type='Deleted' entity='trunk/IO/'
copy dest='trunk/Project/IO/' source='(from trunk/IO/:r9)'
```

The `lines` field in the `revision` table stores the "diff size" of the revision – the number of lines in the difference between the project as it was before the revision and afterwards. This metric has limited use, as a number of activities can produce extremely numerous or zero lines of difference, but is included as it can be a useful measure of activity when taken in aggregate over the course of a project's life.

With the basic data in the database, simple queries and metrics can be made. An example of this is a web interface which students can use to get an overview of their project and check for updates to their data, such as that shown in Figure 4.2.

The next stage in the workflow is to export the data for a single project to a compact, portable file which can be used by a front-end application. XML was chosen as a format for this data file, due to the readily available tools, APIs and software libraries, reducing the scope for introducing errors into the process. The previous example would output an XML-formatted revision as follows:

```
<revision number='10' author='dcs0ab'
  date='2008-04-27 11:26:42 +0100 (Sun, 27 Apr 2008)'
  changesize='0'>
    <message>Moved remotely</message>
    <change type='Deleted' entity='trunk/IO/' />
    <copy dest='trunk/Project/IO/'
      source='(from trunk/IO/:r9)' />
</revision>
```



Perceive: Group 1

home support

Resources

Latest data file: Download here (Right-click and use 'save as'. Alternatively, paste the link into Perceive's 'Open URL' dialog; you will be prompted for the username and password you use to access this page.)

Latest application build: March 19th, 2009

- Neater loading of large data files - smoother progress, sorted out some graphical glitches
- Perceive now recognises 2009 style usernames (e.g. durham1_3, ncl3_1) and classifies students by university
- Fixed a bug where some projects would fail to load under a very specific set of conditions (sorry SEG 9, unlucky...)
- Should now work in Mono on Macs again
- Fixed a couple of typos in the timeline visualisation
- The check for updates on startup can now be disabled in the options tab
- Added an option to disable the update check at start-up
- Fixed a bug where some saved options may not be used when the program is first launched
- Now checks for updated versions of the application
- Should now work in Mono; see changelog for details

Overview

Students	9
Revisions	626
Changes	4430

Student Information

Email	Revisions	Changes	Latest Revision
[illegible]	5	5	11:36, Fri, 05 Dec 2008
[illegible]	1	3	14:42, Thu, 27 Nov 2008
[illegible]	83	1582	06:29, Mon, 16 Mar 2009
[illegible]	119	799	06:14, Mon, 16 Mar 2009
[illegible]	99	476	06:36, Mon, 16 Mar 2009
[illegible]	124	443	06:41, Mon, 16 Mar 2009
[illegible]	138	874	03:55, Mon, 16 Mar 2009
[illegible]	47	47	18:50, Sun, 15 Mar 2009
[illegible]	10	201	11:13, Tue, 17 Feb 2009

Note that these numbers may differ from those shown in the desktop application due to the way it discards some less relevant revisions.

Recent Activity

Revision	Author	Date	Changes
629	[illegible]	06:41, Mon, 16 Mar 2009	1
	" "		
628	[illegible]	06:41, Mon, 16 Mar 2009	30
	"CD"		
627	[illegible]	06:36, Mon, 16 Mar 2009	2
	"Database"		
626	[illegible]	06:35, Mon, 16 Mar 2009	55
	"Build"		
625	[illegible]	06:35, Mon, 16 Mar 2009	1
	"Meh "		

For more information, email Andy at a.j.burn@durham.ac.uk.

Figure 4.2: A simple web interface for project management, demonstrating information available from the database before significant processing

This file is not dissimilar to the output from the log function of `svn` command, as shown here:

```
<logentry
   revision="10">
<author>dcs0ab</author>
<date>2008-04-27T10:26:42.581030Z</date>
<paths>
<path
   copyfrom-path="/trunk/IO"
   copyfrom-rev="9"
   action="A">/trunk/Project/IO</path>
<path
   action="D">/trunk/IO</path>
</paths>
<msg>Moved remotely</msg>
</logentry>
```

The main differences are the absence of the `changesize` metric and the lack of trailing slashes on the directory names (e.g. `trunk/IO/` versus `/trunk/IO`), which makes it hard to distinguish between directories and files and loses some semantic information.

4.4 Perceive

Once a suitable data file is created, it can be loaded into the main front-end application for full processing, analysis and presentation. This application is called `Perceive` and is designed for a range of purposes. The majority of the features of the system are used in Chapters 5, 6 and 7, and are described more fully where appropriate. This section focuses on design and implementation details of the core of the application and how it affects the data analyses which follow.

4.4.1 Preprocessing

By the XML stage, the data has undergone little processing, and the structure of the project at each stage must be reconstructed. In a project consisting of add, update and delete operations, this is a simple process – for each revision the tool simply needs to take the list of current files from the previous revision, add any newly created files and remove any deleted files. The operation becomes more difficult when move and copy operations are used, as the list of files copied must be inferred and retrieved from the given revision, a complex process made more difficult by the fact that entire directories can be copied in one command.

Because the research is focused on files, `Perceive` does not include directories in its processing; considering that many revisions consist of actions using only directories, this results in "empty" revisions, which are ignored. However, a copy or delete operation which appears to affect only a directory cannot be ignored as it must also affect the files contained within that directory.

After early testing, `Perceive` was modified to allow users to specify a list of exclusion criteria used to disregard certain files or types of files. An example of this might be to ignore images, branches or code libraries. This decision was taken primarily due to the numerous files in some projects – one student group project contained over 9,000 files when a local workspace and its local versioning system were accidentally checked in to the group's repository. As some of the processing functions described later involve file networks – with unavoidable $O(n^2)$ complexity – a way to reduce the number of files analysed became necessary. Later revisions of the software somewhat mitigated the complexity, but the filtering system was retained due to its ability to simplify project views.

To test whether or not the tool was able to maintain an accurate list of the files at each stage, a final set of files for a sample of revisions and projects was compared with the output of the `svnlook tree --full-paths` command, which shows the file structure for a project. When the outputs matched, the tool was determined to be functioning correctly.

4.4.2 Class Design and Data Structures

`Perceive` is an object-oriented application written in C# using Microsoft Visual Studio 2005 (and later, Microsoft Visual Studio 2008). The central, eponymous class is `Perceive`, which manages a collection of projects, including loading data files and performing operations which encompass all loaded projects. The tool is capable of reading data files which contain more than one project – for example, an entire year of student projects can be loaded at once and switched between, enabling rapid and easy comparisons and batch analysis. Each `Project` object contains collections of `Revision`, `Change`, `File` and `User` objects, in a structure matching that of the database.

Each `Change` object is classified as one of the following:

1. Added
2. Updated
3. Deleted
4. PropertyChanged
5. Copied
6. Moved
7. Unknown

The first five change types are the SubVersion classifications, while *Moved* is inferred from a combined *Delete* and *Copy* operation. *Unknown* is included to handle changes to the SubVersion environment or corrupted data files.

4.4.3 File Information

`Perceive` is language agnostic in that it operates at a file level independent of implementation or platform. This has the advantage of not limiting the software to any specific language or paradigm, but limits the semantic information it can access. To provide additional functionality, `Perceive` can classify files into one of the following:

1. Code

2. Document
3. Archive
4. Media
5. Data
6. Misc

This classification is done based on the file extension – by default `Perceive` classifies 74 different file extensions, and this set is entirely user configurable, allowing the system to deal with new languages, formats or user choices.

4.4.4 Limitations and Future Development

As discussed in Section 4.4.1 `Perceive` contains data structures which require O(n $\log_n$) or O(n^2) space, and computations with O(n^2) complexity. As large projects can have extremely large numbers of files, these operations can become infeasible. The example given was of a configuration error, but a mature project can easily contain many thousands of files, especially with branching (making a new copy of a module for concurrent development) and tagging (marking specific versions, such as releases or milestones). Currently `Perceive` fails gracefully when a project is too large, disabling certain features and continuing with a recommendation that the user filter some files. Future development, including improved data structures and algorithms, will increase the scale of projects which `Perceive` can fully process.

4.5 Workflow

In summary, there are two workflow avenues which can be used, as shown in Figures 4.3 and 4.4. The first, more complex workflow has the advantage of retaining some semantic data lost by the `svn` log command, and contains the diff size for each revision. The database step is also more suitable to a

web-based interface and the additional features this permits, such as RSS feeds.

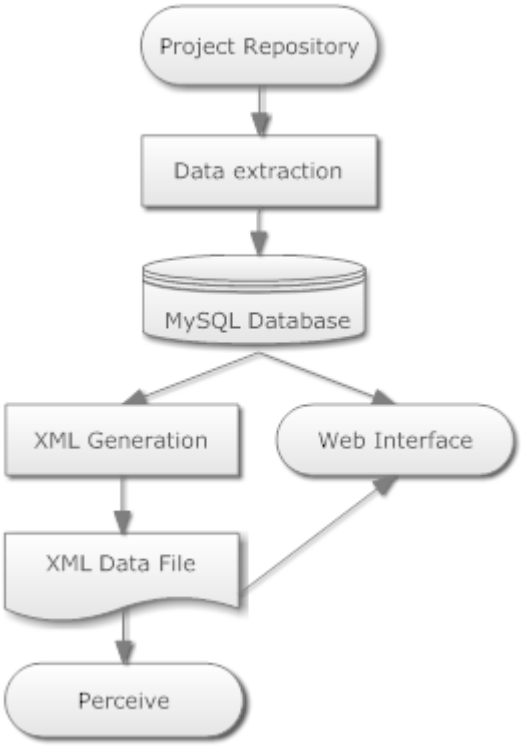


Figure 4.3: The complete workflow process, from repository to database to XML to Perceive

Figure 4.4: The simplified workflow process, generating XML directly from the repository

4.6 Summary

This chapter detailed the nature and development of the tools used at each step in this research, as necessitated by the requirements of each case study.

Specific functionality, such as project management features or visualizations are discussed in the Project Management case study in Chapter 7.

Chapter 5 contains the first of the case studies, a thematic analysis of the revision logs of a series of projects from both educational and open source domains.

Chapter 5

Case Study: Thematic Analysis

5.1 Introduction

As described in Chapter 1, RCS repositories contain much data which can be extracted and analysed to support project comprehension. One source of repository data is the comment log. These comments, which accompany each revision in a SubVersion repository, contain a large amount of information which can be used to gain an understanding of a project. The primary use of these logs is to help determine a change history and to associate intention and effects of a revision with the actual code changes, providing both a record of and a guide to a project's development. In many cases specific formatting and vocabulary is mandated in the logs, which facilitates data mining, but it is typical that the logs consist of free-form, unstructured comments, which require natural-language processing. This makes automated analysis difficult – keyword extraction for tag clouds are an example – and manual analysis of a large collection of text can be a time consuming process.

In an educational setting, the assessment and reporting of collaboration and progress in group projects often involves blogs, reports, diaries, interviews and reports (Burd and Drummond, 2006; Drummond and Devlin, 2006). However, even a cursory examination of the comment logs reveals a remarkably open, honest and direct insight into the activities and dynamics of a group;

given this finding a full analysis of a series of project logs was planned and conducted.

The case study consists of three sub-studies; the first stage (Study 1.1, see Section 5.4) performed an analysis of the projects of a single cohort of students (Burn, 2008), and was expanded in the second stage (Study 1.2, see Section 5.5) to include a second year of projects and subsections of four open source projects, with the aim of replicating the first stage and investigating further research questions (Burn, 2009). Finally, the third stage (Study 1.3, see Section 5.6) performed an analysis of a complete open source project to improve the quality of the comparisons with student projects.

5.2 Case Study Design

This section details the goals, techniques and evaluation of the case study using the DECIDE framework. The case study uses as its subjects the three sub-studies described above, and draws its conclusions from the outcomes of those studies.

Figure 5.1 shows the structure of this case study, how the sub-studies feed into the case study and how the case study feeds back to the overall research question. While the three sub-studies can stand alone as a single piece of research, the results and experiences are used to evaluate the use of repository analysis in project profiling; the case study will use that evaluation to assess whether or not repository analysis can make a useful contribution to project contribution in this context.

5.2.1 Research Goals

The case study is conducted with the goal of determining how the repository comment logs can be used to profile projects to support project comprehension. This profiling will use the process of thematic analysis. More specifically, issues such as how successfully thematic analysis can be applied to comment logs, how this is affected by the domain or type of project and how complex

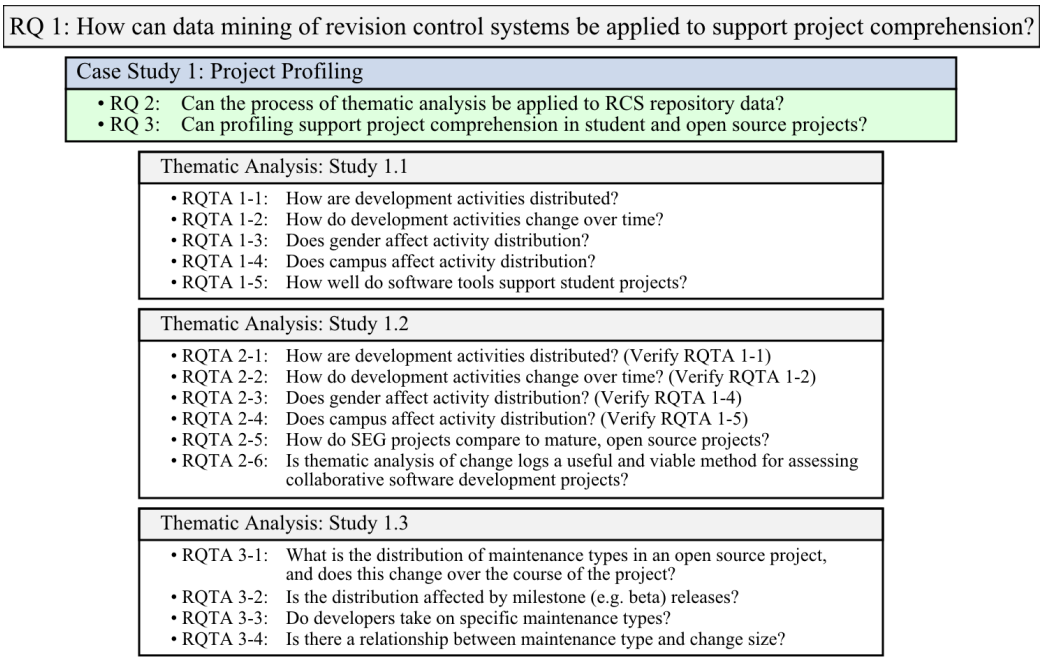


Figure 5.1: Structure of the research questions and studies of the project profiling case study

and resource-intensive the process is.

If the study determines that thematic analysis can be successfully and usefully applied to comment logs, then this research will have an impact in both educational and industrial contexts, as the range of applications is potentially very broad.

5.2.2 Research Questions

The overall goal of this research is to evaluate techniques by which data mining of project repositories can be used to support project comprehension. Therefore, the following research questions are identified:

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data?
- *RQ 3*: Can profiling support project comprehension in student and open source projects?

These questions are intentionally high level, exploring the *application* of thematic analysis itself. The actual conduct of the technique is evaluated using a series of thematic analyses each of which address more specific research questions based on educational and industrial projects.

By addressing these research questions, thematic analysis can be evaluated in the overall context of techniques which can support project comprehension, and either be accepted or rejected as such a technique.

5.2.3 Evaluation Paradigm and Techniques

To address the stated research questions and evaluate whether or not thematic analysis is a useful and viable technique, a series of studies were conducted, using thematic analysis to categorize maintenance activities in student and open source projects. The outcomes of these sub-studies are used to evaluate the effectiveness of thematic analysis as a process in the context of project comprehension.

5.2.4 Practical Issues

There are two primary practical issues involved in this case study: the selection of the projects, and the application of the thematic analysis. Team-based student projects were made available for analysis, and a number of open source projects are freely available on the internet. For this reason, selections of both student and open source projects were used for analysis, which allows not only for analysis within project groups, but enables comparisons between student and open source projects.

Secondly, there exists no research describing a formal thematic analysis of RCS repository data, and it was not certain that the technique could successfully applied (as reflected by the first research question - *RQ 2*: Can the process of thematic analysis be applied to RCS repository data?). The technique, as described in detail in Section 5.3 involves iterative design, review and revision stages, followed by a careful and lengthy manual categorization

process. Therefore, this case study is also concerned with the time and effort required to prepare and conduct a thematic analysis.

5.2.5 Ethical Issues

The open source projects used in this case study are freely available on the internet, whereas the copyright for the student projects is owned by the universities involved. Therefore, ethical clearance was sought and granted to use the student projects, following sufficient anonymisation. As no human subjects were involved with this case study, no further ethical clearance was necessary.

5.2.6 Evaluation and Discussion of Results

After each of the sub-studies are conducted, the findings will be evaluated in the context of the overall research questions and goals, to relate each thematic analysis to the overarching aim of using thematic analysis to support project comprehension.

5.3 Thematic Analysis

Thematic analysis (Braun and Clarke, 2006) is a qualitative analytic method which aims to uncover patterns or "stories" in data. It is conducted over several stages; firstly a set of codes is defined and each data item is labelled with one of these codes. This code scheme is checked by a reviewer to determine if it is balanced, repeatable and unambiguous. The codes are refined and reviewed until the researcher and the reviewer reach a pre-determined agreement rate. In this analysis, given the vagueness and ambiguity of the comments being analysed, it was decided that an agreement rate of 80% would be desirable. The data being coded in this study are the individual comments in the revision logs.

While thematic analysis is not commonly utilized in the field of computer

science, there are examples of research based on similar processes. In a taxonomical study of large revisions (Hindle et al., 2008) Hindle et al. classified the largest revisions from a series of repositories into the maintenance activities they represented; this allowed the researchers to investigate the causes and effects of the large revisions which are frequently excluded from revision control system mining research projects. Similarly, in a study of open source change logs (release summaries, as opposed to revision comments – the term *release notes* will be used here to avoid ambiguity) (Chen et al., 2004) changes were classified by type, e.g. corrective, enhancement, rearrangement or comment. This classification was based on both the release notes and a direct analysis of the source code, and the discrepancies found led the researchers to conclude that additional data sources such as revision logs should be used to verify the completeness and correctness of release notes. This research is indirectly related to the work of Chen et al. in that the classification process is performed using the revision messages, but the study lends credence to the assumption that revision messages are an accurate measure of development activity.

5.4 Study 1.1: 12 Group Projects

Each year, second year computer science students from Durham and Newcastle universities carry out cross-site, collaborative software engineering group (SEG) projects (Drummond and Devlin, 2006). Groups consist of a team from Durham and a team from Newcastle. Each group has a similar requirements specification, and implementation is partitioned between the teams. For example, in the academic year of 2006/07 the project consisted of a desktop application and a corresponding mobile application; the Durham teams worked on the desktop aspect, while the Newcastle teams developed the mobile portion. Each Durham team is managed by third year students from a Project Management module, and these managers attempt to facilitate groupwork, collaboration and communication, as well as guiding the development process

(Burd and Drummond, 2006).

The implementation phase for each group was supported by a SubVersion repository (Collins-Sussman, 2002) and every change to a project was reflected in the group's repository. Each time a revision is submitted, the student is prompted for a message or comment to describe the changes. It was these comments that formed the data for this analysis. A thematic analysis of the revision logs of the 12 SEG projects from the 2006/07 academic year was conducted. The campus, gender and group of each student were recorded, as were the time and date of each revision. In this study, the campuses are referred to as *C1* and *C2*, assigned randomly.

5.4.1 Research Questions

The main aim of the first stage of this analysis was to determine how development activity changed over the lifetime of a project. Sub-questions include whether or not these activities are affected by campus, and if they are consistent throughout the project or change over time. A secondary aim was to determine how well students made use of software tools to facilitate their projects, and whether this could be improved in future years.

- *RQTA 1-1*: How are development activities distributed?
- *RQTA 1-2*: How do development activities change over time?
- *RQTA 1-3*: Does gender affect activity distribution?
- *RQTA 1-4*: Does campus affect activity distribution?
- *RQTA 1-5*: How well do software tools support student projects?

This study addresses these questions in the context of the SEG projects, but the findings are equally applicable to any academic, group-based projects, especially cross-site or cross-campus projects.

5.4.2 Design

The first step of the analysis was to devise a set of codes which could result in useful data and which were repeatable and clearly-defined. In a similar – but independent – approach to the research of Hindle et al. (Hindle et al., 2008) the initial codes were derived from the types of software maintenance: perfective, preventative, adaptive and corrective (IEEE Standard 610.12-1999). This set of codes did not fit the data however, and a new set was defined:

- *Additive*: For new features added to the project
- *Progressive*: For improvements or expansion to existing elements of the project
- *Preventative*: Cleaning, testing or documenting of the project
- *Corrective*: Fixing bugs and errors in the project
- *Misc*: Anything which did not fit in the other codes[1]

An initial application of this scheme resulted in a distribution of codes shown in Table 5.1.

Additive	18.8%
Progressive	44.7%
Preventative	13.2%
Corrective	13.7%
Misc	9.6%

Table 5.1: Distribution of maintenance activities using the second set of codes

In the SEG projects bug fixes tend to be carried out at the same time as other changes and as such are not reported, which explains why corrective

[1]It should be noted that "Misc" codings are acceptable and expected, but should typically form the smallest group – if it is too broad then this indicates a problem with the codes.

actions are so low. Progressive changes are dominant, partly because the threshold for addition being counted as an additive change was quite high. "Added a new constructor" or "Added drop-shadows" could both be counted as additive, but because they alter existing functionality or features they are counted as progressive. This vagueness was expected to lead to a lower than acceptable agreement rate in code validation.

5.4.2.1 Code Validation

An independent reviewer was sent samples of the data – 10% from each coding, 103 in all – and asked to code them based on the codes and definitions provided.

The review gave a 67% agreement rate, below the acceptable level. Some of the changes were simple mistakes in the initial application of the codes (e.g. automatically assuming a comment beginning with "updated" would be progressive), and some were results of a comment genuinely having two possible codings (e.g. "Added a splash screen, and fixed the database bug"). A small proportion came from misunderstandings of the code definitions, especially in use of the *Misc* code. When the definitions were improved the agreement rate rose to slightly over 70%. There were also some occasions where contextual information such as comments on adjacent revisions suggested a coding that differed if the comment was taken in isolation.

5.4.2.2 Revise Codes

Based on the initial review, it was clear that the codes needed revising. To begin with, *Misc* was expanded to include "multiple possible codings", to cover situations where a large change cannot be slotted into one code. This code may equally have been fixed by allowing multiple codings, but that is beyond this analysis.

More importantly, *Progressive* was renamed *Incremental* and the definition strengthened to emphatically include additions made to existing features. This repaired the largest difference in the initial coding and the review.

The alterations and corrections (prior to the big *Progressive* change) brought the agreement rate up to 77%, nearly acceptable. If the alteration of the *Progressive* change were successful, the agreement rate would be acceptable in the second review.

5.4.2.3 Second Review

The second review using the revised codes actually had a slightly worse agreement rate than the first – around 63%. This was caused by two issues. Firstly, the reviewer was more likely to apply the *Misc* code where the data was slightly ambiguous; often this ambiguity was removed when the data was placed in context. Secondly, the problem with differentiating the *Incremental* and *Additive* codes remained.

5.4.2.4 Second Code Revision

Preventative was renamed to *Perfective*. It was initially called preventative to map to the well-documented maintenance activity, but the code expanded to include other forms of maintenance and so the name changed to compensate. *Perfective* includes testing, cleaning, refactoring, deleting, restructuring, commenting and JavaDoc. *Incremental* and *Additive* were merged to form *Developmental*. Due to the inability to reliably separate the two codes, it was deemed sensible to merge them. *Misc* was split into *Misc* and *Ambiguous*. *Ambiguous* is used when progress or changes have clearly been made and are being reported, but it is not clear which activity type was carried out. It is also applicable when there are clearly two or more codes applicable (e.g. *Corrective* and *Developmental*). *Ambiguous* is kept separate from *Misc* because even though it is not known what activity type it describes, the presence of ambiguous messages and how frequently they occur is interesting in itself, and so is considered for analysis. *Misc* is now solely for irrelevant or out-of-scope comments. It also includes early instances of "test" and "initial import" which are obviously SubVersion related and not directly connected to the project itself.

Vague comments such as "change", "working now", "Adam's changes" or "updates" are coded as *Developmental* – although they may seem ambiguous, investigation of the source data reveals that the majority of these revisions are developmental in nature.

5.4.2.5 Third Review

The third review, using the new codes and definitions, had a higher than 90% agreement rate – well above the 80% minimum acceptable rate. Therefore the codes were considered repeatable, balanced and unambiguous, and therefore final.

5.4.3 Limitations and Threats to Validity

Although the data set is large it is only a quarter of the total set of activities. Therefore a large amount of potential data is missing, which could theoretically impact the results. In some cases, the comments were primarily from one student within the group, and in others the comments came from a larger body of students who commented less frequently. If the distribution of comments is random or arbitrary then this would not be a problem – each activity would be impacted equally. On the other hand, if people were systematically not commenting minor bug fixes (for example) then that activity would be under-reported. Looking at the data more closely, there seems to be no systematic bias or selection occurring with comments – in some cases it is random and in other cases it is determined by the individual student. Extending the study with data from subsequent SEG projects will help to mitigate any unseen problems.

5.4.4 Evaluation

5.4.4.1 RQTA 1-1: How are Development Activities Distributed?

The overall spread of activity types is described in Table 5.2. It was noted previously that a *Misc* code should be the smallest group, but in this case

it was not possible – one group had a disproportionate amount of revisions concerning documentation unrelated to the implementation itself, and so they were all classified as *Misc*. It would have been possible to provide those with a separate coding, such as *Documentation* and ignored them for the purposes of analysis, but as part of *Misc* it maintains their relevance to this analysis in terms of "irrelevant comments".

Developmental	53.8%
Perfective	14.3%
Corrective	13.2%
Misc	10.5%
Ambiguous	8.3%

Table 5.2: Distribution of maintenance activities

Figure 5.2 shows how the spread of activity types differed between the groups of students, revealing how some groups were almost entirely focussed on developmental activities, while other were much more balanced. It is probably not a coincidence that the highest scoring group had the highest proportion of corrective maintenance and fewer Misc and Ambiguous codes, although there is little correlation between any particular activity and final group score, supporting existing research (Mierle et al., 2005). Any correlations which do exist are just as easily explained by better developers as opposed to better practices. There was great variance of these categories within groups (standard deviation ranged from 6.1% to 13.2%) reflecting the varied developmental and commenting practices adopted by each group.

If the *Misc* and *Ambiguous* codes are ignored, then *Developmental* activities account for two thirds of the revisions, while *Corrective* and *Perfective* each account for one-sixth.

5.4.4.2 RQTA 1-2: How do Development Activities Change Over the Course of a Project?

Figure 5.3 shows how the various activity types changed across the course of the projects' lifetimes.

Corrective is, as expected, low for the first 20% of each project, around 3-4%. As development continues, *Corrective* rises to 15-20%, where it remains for the life of the projects.

Developmental varies quite widely, between 42% and 67%, with a low of 20% (this anomaly coincides with the Christmas holidays). Overall, developmental activities – adding, expanding and improving features – form the majority of the work for the entire life of the project.

Perfective activities hover between 8% and 13%, with a very low variance. In other words, students consistently appear to spend 10% of their effort testing, documenting, commenting, cleaning and refactoring their code – low compared to the ideal proportion, but expected in the context of inexperienced developers working to a strict deadline with no scope for their code to be maintained subsequently.

Ambiguous activities range between 1% and 13%, with no apparent pattern – this is also to be expected as *Ambiguous* is not an activity in itself but the inability to classify an activity based on the comment. Only by requiring better commenting practices or by time-consuming investigation can this group be reduced.

Misc begins very dominant, 42-64% in the first fifth of the project cycle, then dropping off to much lower values of 0-14% for the remainder. This is caused by several factors – students are still learning to use SubVersion and the commenting system, and students are still working on other aspects of the project, which crosses over into SubVersion when things like documents are committed. Combined with the lower amount of revisions at that stage of the projects, this makes *Misc* more pronounced before being overshadowed by other activities.

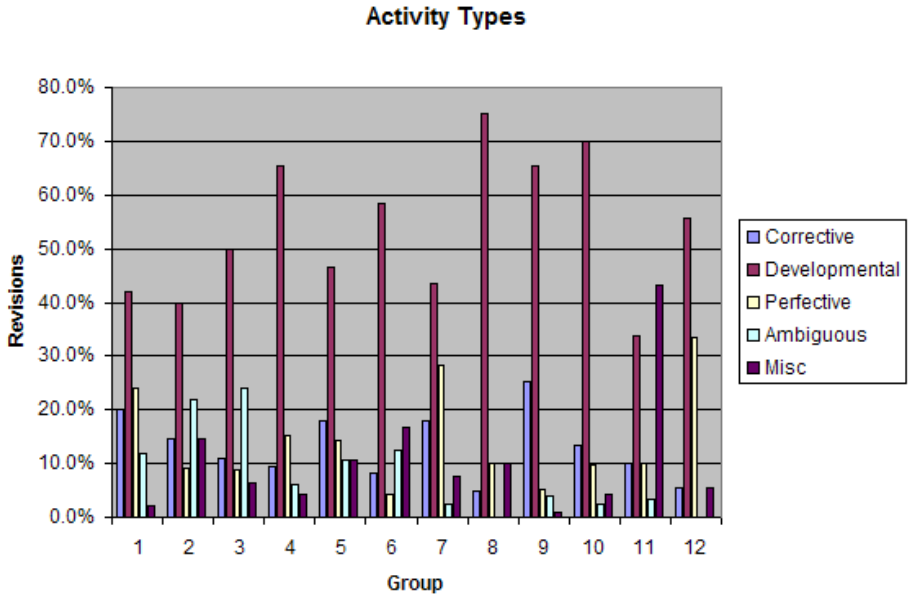


Figure 5.2: Activity types broken down by group

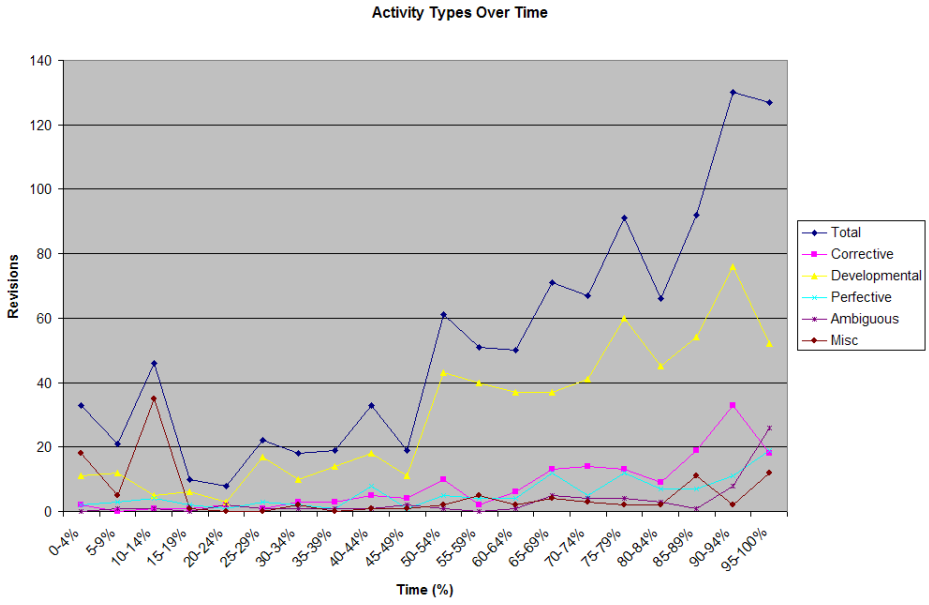


Figure 5.3: Activity over time

5.4.4.3 RQTA 1-3: Does Gender Affect Activity Distribution?

Despite having gender data available, it was not possible to address this question, as there were so few revisions and comments from females, due to the gender disparity in the cohort. The low proportion of revisions committed by females (43 out of 4,111, i.e. 1%) was compounded by the low proportion of comments in total. The only relevant result was that women also accounted for about 1% of the comments, and so were not over- or under-represented. A total of 9 comments from females meant that no further analysis could be conducted.

5.4.4.4 RQTA 1-4: Does Campus Affect Activity Distribution?

The SEG projects were carried out by teams consisting of students on two campuses – C1 and C2. The teams were mixed so that development was inevitably a cross-site process. In total students from C1 committed 63% of the revisions – considering that the project was worth twice as much to C1 students as C2 students, this is a fair proportion.

In terms of the revision data, C1 accounted for 78% of the comments – significantly above the expected proportion.

Figure 5.4 shows the breakdown of activity types differs between campuses. Most categories were evenly matched between campuses, but C1 appeared to do significantly more developmental work than C2, while C2 performed twice as much perfective work as C1. This could be due either to differing software engineering practices fostered by the respective universities, or if it is a result of the nature of the different aspects of the project each team was working on. Therefore, when talking about the difference between campuses, it is also possible that we are talking about the difference between project domains.

Figure 5.5 shows an overview of how the work levels of the two campuses changed over time. Both campuses increased their work rate towards the end of the projects, but C1 hit a peak much earlier on and maintained it, whereas C2 spiked much closer to the end. This had the result that for the middle third of the project (40% – 70%) C1 were doing the majority of the

work, even above the two-thirds ratio expected. Lastly, C1 began work much earlier, and then dropped off again around the Christmas holidays.

Figures 5.6 and 5.7 show a more detailed breakdown of how the activity breakdown of the campuses changed over time.

5.4.4.5 RQTA 1-5: How Well Do Software Tools Support Student Projects?

The fact that only a quarter of all revisions were accompanied by a comment suggests that students did not, on the whole, make full use of the tools provided to aid them in their project. Equally, the prevalence of *Misc* and *Ambiguous* activities shows that even when comments were supplied, SubVersion was not being used properly. *Ambiguous* codes occur in two situations; either the comment was ambiguous, or the revision itself consisted of more than one maintenance activity. *Misc* codes occur when the comment is irrelevant to the project implementation (e.g. flippant remarks, or work relating to other project phases such as requirements). The fact that nearly a fifth of all comments were *Ambiguous* or *Misc* suggests that students require further training in the use of SubVersion, and a deeper education of the benefits of proper software development practices.

5.4.5 Conclusions

Thematic analysis, by manually attaching additional information to a data set, allows us to see patterns in that data that quantitative analysis itself could not uncover. In the case of the SEG projects, by analysing activity types it is possible to gain a better understanding of the development processes.

In comparison to ideal development practices, where feature development is stopped prior to release to allow for bug fixing and "polish" to take place, SEG projects actually saw an increase in developmental activity as the projects drew to a close and while corrective activities did increase too, it was not as significant as it should have been. There was also a marked increase in

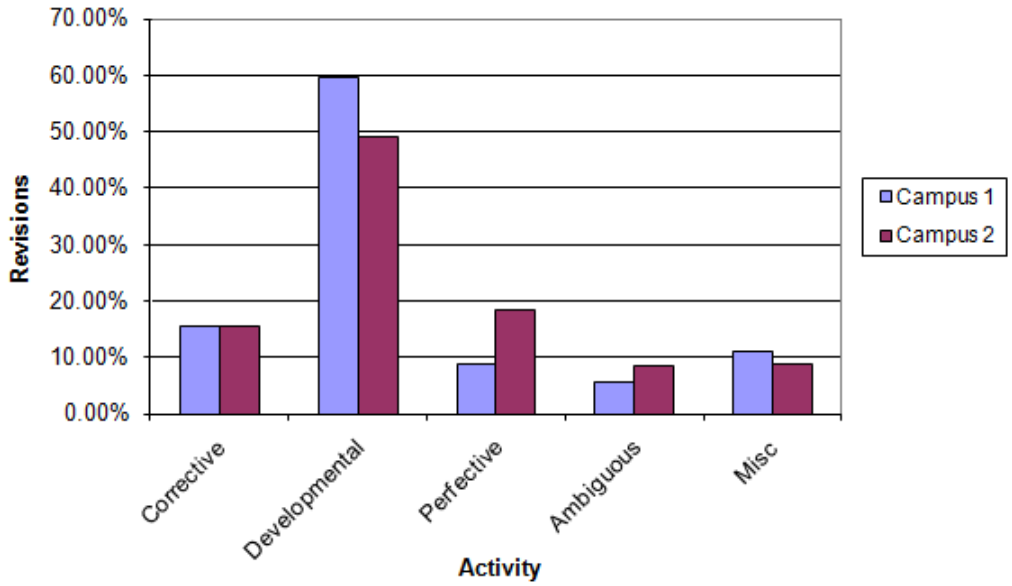


Figure 5.4: Comparison of activity types across campuses

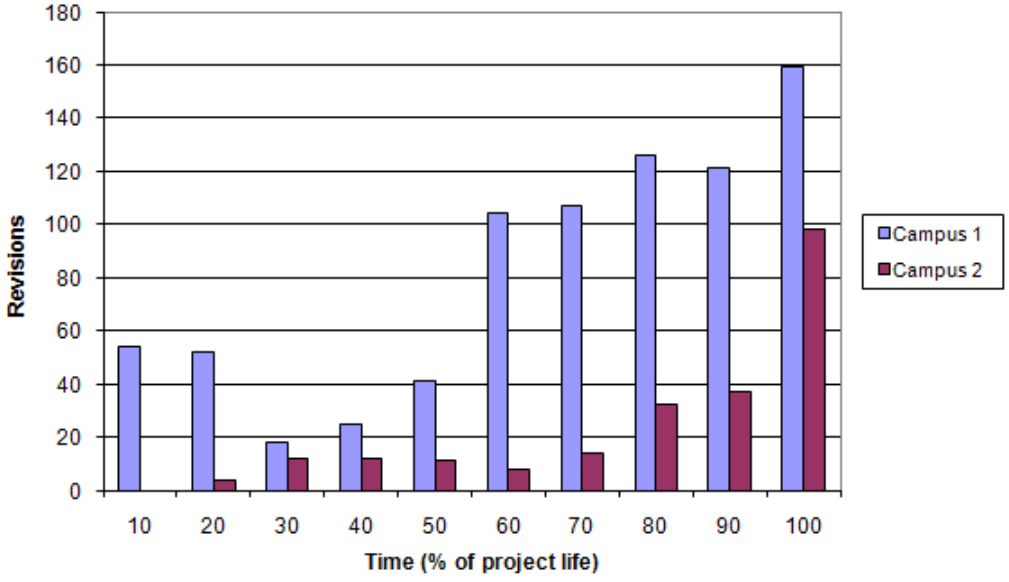


Figure 5.5: A broad comparison of work levels on each campus over time

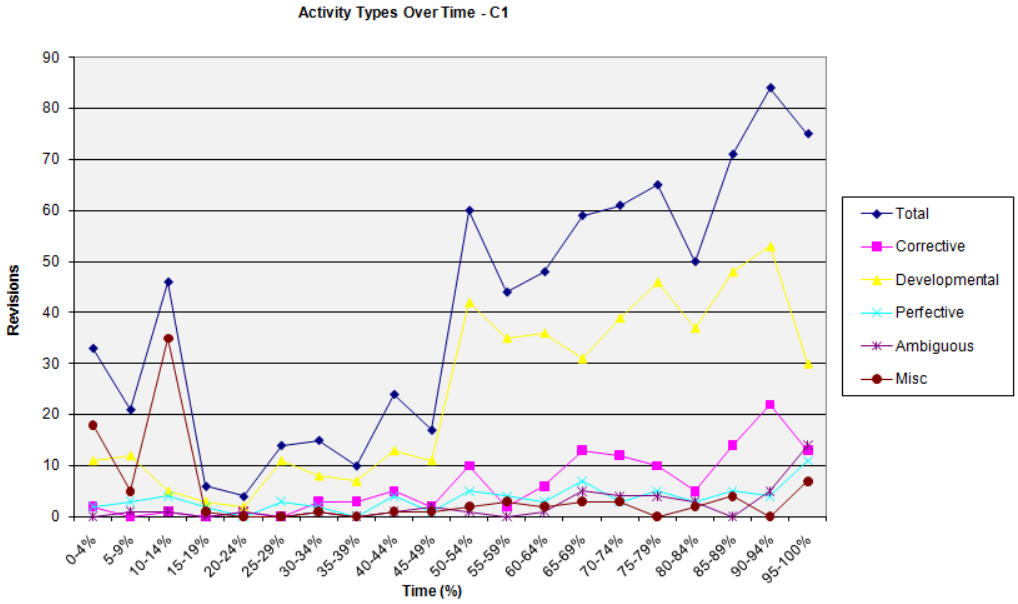


Figure 5.6: C1 students – Breakdown of the various activity types

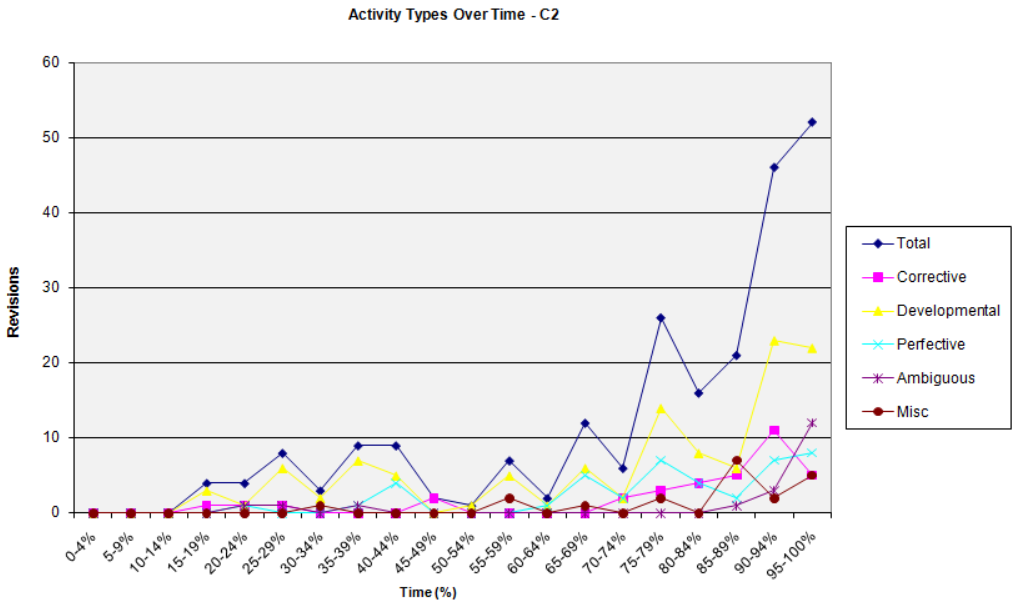


Figure 5.7: C2 students – Breakdown of the various activity types

ambiguous comments towards the end of the project, possibly caused by increased frustrations, or deadline pressure.

Overall, it is clear that students' use of SubVersion is not optimal. In an environment where there is no scope or requirement for future development or maintenance of their projects there is little incentive to make proper use of the software provided. Unfortunately communication is often cited as one of the groups' main problems in collaborative work. During the implementation phase, when communication and cohesion are highly important to the development of a high quality, well-integrated software application, students largely ignore the facilities provided by SubVersion which could aid communication within groups. This is likely due to a lack of students' awareness of the features offered by SubVersion, and how they could use them to their advantage. As one student said in a SubVersion comment, "WHERE DOES THIS MESSAGE SHOW UP?"

This has led to a tendency for students to be much less formal and rigorous in their use of SubVersion comments. While this makes analysis in terms of development activities difficult, it is a useful mine of information with regard to social dynamics. There is scope for further thematic analysis using different code groups to uncover more patterns. An example of this might be to code based on the tone or mood of a comment, and investigate how flippancy, competitiveness, aggression and frustration vary over time and between demographic groups.

5.5 Study 1.2: 22 Group Projects and Four Open Source Projects

Study 1.1 determined that thematic analysis of project logs could be successfully used to examine the behaviour and patterns of academic group projects. However, due to the fact that the 12 projects analysed were all taken from one cohort, an expanded study was conducted with the aim of investigating whether the findings were consistent with those of other years'

projects. The expanded study also included four open source projects to provide a comparison between real-world projects and projects from an educational environment. A second year's data added to the original data set and used in a repeat of the same analysis was expected to be of great help in minimizing the problems caused by low comment ratios, especially among women, and also help to uncover whether campus differences are caused by the environment or by the project domain.

5.5.1 The Projects

14 additional projects were analysed: the 10 2007/08 SEG projects and four open source projects.

5.5.1.1 2007/08 Projects

In the original study, there were potential threats to validity from the nature of the projects – all 12 were from the same year and covered the same basic specifications. By analysing an additional year's projects it was possible to reinforce the results from the original study.

5.5.1.2 Open Source Projects

Due to the nature of open source software, it is possible for members of the public to access the SubVersion repositories and examine the source code and history. In several cases it is possible to download and create a copy of the entire repository, which allows much deeper analysis. Four such projects were used: PuTTY (the popular SSH client), CapiSuite (a discontinued Linux ISDN telecommunication suite), Parrot (a virtual machine for dynamic languages) and GNUstep (part of the GNU project). While SEG projects are developed over the course of weeks, these projects are developed over years and can consist of thousands of revisions. Rather than perform a complete analysis of each of them it was decided to analyse only the first 150 revisions from each of them. This would allow a much more practical comparison with

the SEG projects by covering a similar timeframe and stage of each project. While the initial set of SEG projects averaged over 300 revisions per project, when revisions with no comments were removed this number dropped to 85 revisions per project.

The open source projects are included to provide a real-world comparison with the SEG projects – mature projects developed by experienced programmers in an environment demanding good collaboration provide an excellent point of comparison to academic projects performed by students new to collaborative development.

5.5.2 Research Questions

The original study aimed to explore how development activity changes over the lifetime of a project, and what factors this might be affected by. This study seeks to verify the first study with an expanded set of projects, and then to expand it with a deeper analysis and comparison with projects in other contexts. When the findings of Study 1.1 were published (Burn, 2008), feedback suggested that exploring the viability of thematic analysis as an educational tool would be a worthwhile avenue of study. The research questions are therefore:

- Verify the findings of the initial study:
  - *RQTA 2-1*: How are development activities distributed? (*RQTA 1-1*)
  - *RQTA 2-2*: How do development activities change over time? (*RQTA 1-2*)
  - *RQTA 2-3*: Does campus affect activity distribution? (*RQTA 1-4*)
  - *RQTA 2-4*: How well do software tools support student projects? (*RQTA 1-5*)
- *RQTA 2-5*: How do SEG projects compare to mature, open source projects?

- *RQTA 2-6*: Is thematic analysis of change-logs a useful and viable method for assessing collaborative software development projects?

5.5.3 Thematic Analysis

Experience from the first study led to two slight revisions of the set of codes being used. Firstly, to better reflect the activities it describes, "Perfective" was renamed to "Administrative". This is a minor change, and has no effect on the results from the initial study. Secondly, while "Ambiguous" previously referred to two meanings, in this study it was split into two codes:

1. *Ambiguous*: A change has clearly been made, and partially documented, but the type is not clear

2. *Multiple*: Multiple activities – (e.g. corrective and developmental in the same revision)

This is also a minor change, intended to help differentiate between desirable and undesirable codes. Previously an "ambiguous" comment could be either good (e.g. a clear, informative comment which refers to two types of activity) or bad (an unclear comment), so it was decided to make this change.

The final set of codes used in this study are:

1. Developmental
2. Corrective
3. Administrative
4. Multiple
5. Ambiguous
6. Misc

The results from Study 1.1 were updated to use this new scheme for the scope of this second study.

5.5.4 Limitations and Threats to Validity

One of the aims of this study was to mitigate some of the limitations of the first study – such as the homogeneous nature of the projects and the single year of students. While the addition of a wider range of projects has indeed addressed this, it has introduced a new set of limitations.

A direct comparison cannot be drawn between SEG projects and open source projects due to their relative lengths. As discussed in Section 5.5.1 SEG projects run for weeks, while the open source projects run for years. By taking only a slice of the revisions, significant results or behaviours could be missed. Study 1.3 in Section 5.6 describes a thematic analysis of a complete open source project, which addresses this issue.

Another limitation is that comments in the open source projects frequently require domain-specific knowledge to understand properly. This problem was addressed by coding a sample of revisions in each project outside of the set selected for final analysis; the experience from this training exercise resulted in a better knowledge of the project and the terminology used.

Finally, students were unaware that their change-logs would be analysed. It is possible that if they knew the analysis was being carried out, especially if it were to support assessment, that their behaviour would change. One of the benefits of the change-logs at the moment is that they provide an honest, open insight into student behaviour, which would likely change. Conversely, if students knew that their change-logs were being used in assessment, it would likely encourage them to apply the theories they have learnt. Future research is planned which will investigate how student change-logs change when the students are aware that the logs will be used to support assessment.

5.5.5 Evaluation

The 2006/07 SEG projects have a comment/revision ratio of around 25%, while the 2007/08 SEG projects are around 60%. Contrasted with this, the open source projects have a ratio of almost 100%, highlighting a crucial

shortfall of SEG projects. Both open source and SEG projects are distributed, collaborative environments with all the attendant difficulties this presents, especially in communication. Open source projects make good use of tools available to them – mailing lists, chatrooms and development tools such as SubVersion – to work together, whereas SEG students cite communication as a major difficulty and hindrance to development, despite not making good use of the tools available to them.

5.5.5.1 RQTA 2-1: How are Development Activities Distributed?

Overall, the spread of activity aggregated over all 26 projects are shown in Table 5.3:

Administrative	13.9%
Developmental	46.1%
Corrective	12.0%
Ambiguous	18.4%
Multiple	4.7%
Misc	5.0%

Table 5.3: Distribution of maintenance activities

As expected, developmental activity accounts for the largest amount of revisions. However, by themselves these figures do not provide much information. Figure 5.8 shows how the two years of SEG projects compare to each other.

The distribution is largely similar between years aside from the spike in ambiguous revisions in 2007/08, almost four times as high. Examination of the individual groups shows that this is not caused by one anomalous group but by a consistent increase in ambiguous comments. It is interesting to consider that the ratio of comments to revisions was also much higher in the 2007/08 projects – 60% compared with 25%. Whether this was due to teaching or a factor inherent in the cohort cannot be known. However, an

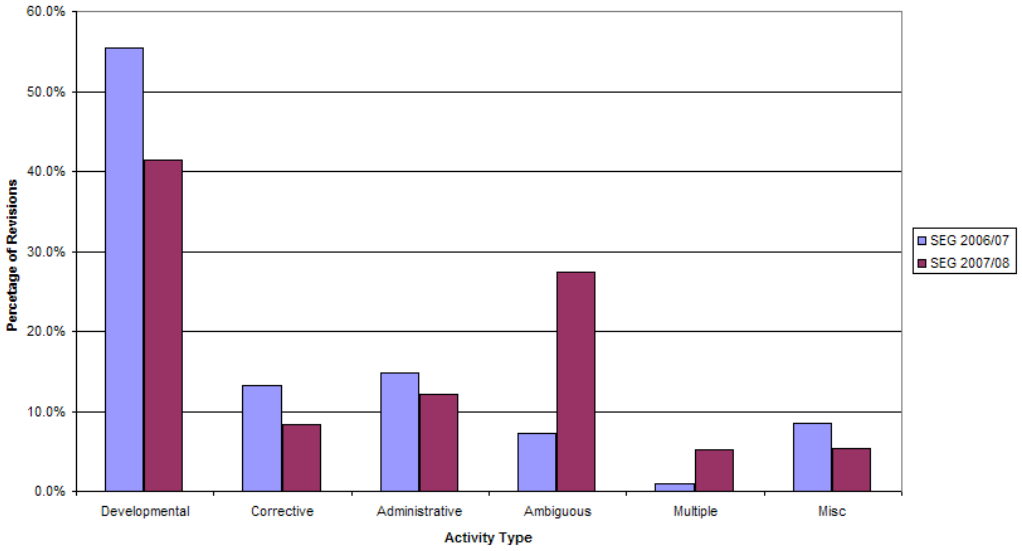


Figure 5.8: Activity Types Between SEG Project Years

ongoing question has been "how useful are automatically generated metrics in supplementing assessment?", with a specific sub-question being "is the comment/revision ratio a useful indicator of performance?" These results would suggest that this simple metric is not necessarily a useful measure – a higher comment ratio has simply led to a disproportionate increase in meaningless or unhelpful comments – "filler" in many cases.

5.5.5.2 RQTA 2-2: How Do Development Activities Change Over Time?

In the initial study it was shown that, overall, activity was skewed towards the deadline – more work was done at the end of the project than at the start. This was an entirely expected result considering the subjects, and is repeated in the 2007/08 SEG projects. A breakdown of the activity types over time showed few patterns in the 2006/07 data – there was a lot of variation between groups, leading to a confused overall view. In the 2007/08 data however, there are some definite patterns evident. There is a decrease in the proportion of developmental activities over time, while corrective and "multiple" (typically

a combination of developmental and corrective) activities increase as the deadlines approach. This is a much better trend than the previous year and is closer to how theory states, i.e. as a release (final deadline in this case) approaches, there should be less emphasis on adding new features and more on polishing and fixing the existing code.

No comparisons can be drawn between the open source and SEG projects in this respect due to the differences in the projects – SEG projects are closed and finished, while the others are ongoing.

5.5.5.3 RQTA 2-3: Does Campus Affect Activity Distribution?

As stated earlier, the SEG projects are carried out by teams consisting of a mix of students from Durham and Newcastle universities. One of the aims of the original study was to discover if there were differences in behaviour between students from these two campuses. It was found that there were differences in activity distribution between the two campuses, but it was not clear whether these were limited to that one year or showed consistent differences between the departments. As was reported for the 2006/07 SEG projects, the amount of work carried out by students from each campus was in line with the relative weightings of the courses – the project was a larger proportion of the year's summative work on one campus, and the distribution of work reflected this. On the other hand, C1 commented a significantly higher proportion of their revisions than C2 – 63% of the revisions but 78% of the comments. In contrast, while the distribution of activity types was similar across campuses, C1 was responsible for significantly more developmental work, while C2 performed twice as much administrative work as C1.

In the 2007/08 projects, the distribution of revisions was roughly the same, and again the students from C1 were much more consistent in commenting the revisions. On the other hand, as shown in Figure 5.9, maintenance types were far more evenly balanced in the 2007/08 projects, with no significant differences between the campuses. It is therefore still impossible to say what factors affect the practices and behaviour of the different departments – cohort,

training, experience, project domain, another factor entirely, or a combination of these. Repeated studies on future projects will be able to explore this further.

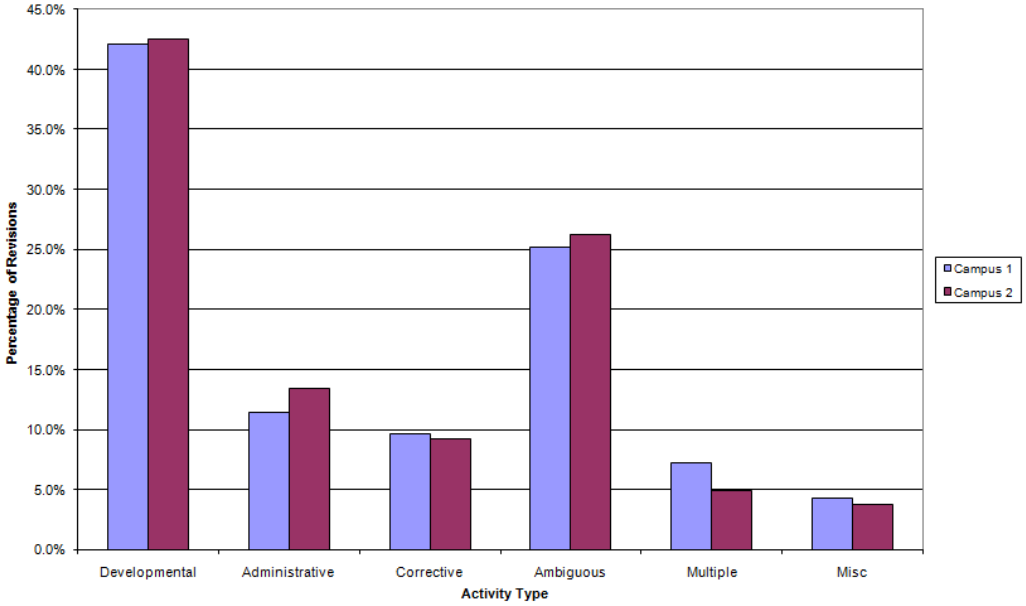


Figure 5.9: Breakdown of activity types between campuses

5.5.5.4 RQTA 2-4: How Well do Software Tools Support Student Projects?

In Section 5.5.5.1 the differences between the two cohorts were discussed – although there was a much higher comment-to-revision ratio in the second year there was a corresponding increase in ambiguous comments, suggesting that while students were perhaps more aware of the need for comments, they did not understand or accept the purpose of them.

5.5.5.5 RQTA 2-5: How do SEG Projects Compare to Mature, Open Source Projects?

Figure 5.10 shows how activity types are distributed between sets of projects – SEG and open source projects. Again, in both cases developmental activities

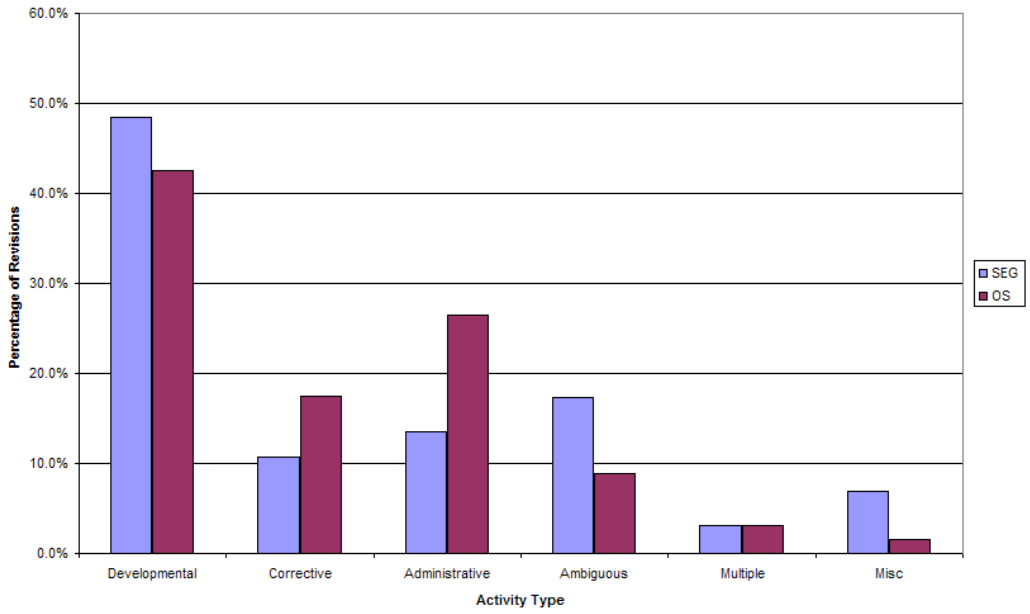


Figure 5.10: Activity Types Between Sets of Projects

are the largest group, while the open source projects have significantly more corrective and administrative revisions than the others and much lower proportion of ambiguous activities, coupled with consistently low "multiple" activities. Overall, the desirable activities (administrative, corrective and developmental) are more prominent and evenly divided in open source projects than in the other projects, whereas they have a significantly lower proportion of undesirable activities (ambiguous and misc). A breakdown of this is shown in Figure 5.11.

It is also worth reiterating that the revisions analysed for the open source projects are taken from early in their development, and so the higher incident of corrective activities is even more noteworthy – the typical development cycle for an open source project tends to include a "feature freeze" followed by a "code freeze", which restrict the development to existing features and bug-fixing, respectively (Love, 2003). If the analysis covered a time period later in the cycle where a release was being finished, the level of corrective activities would be higher still.

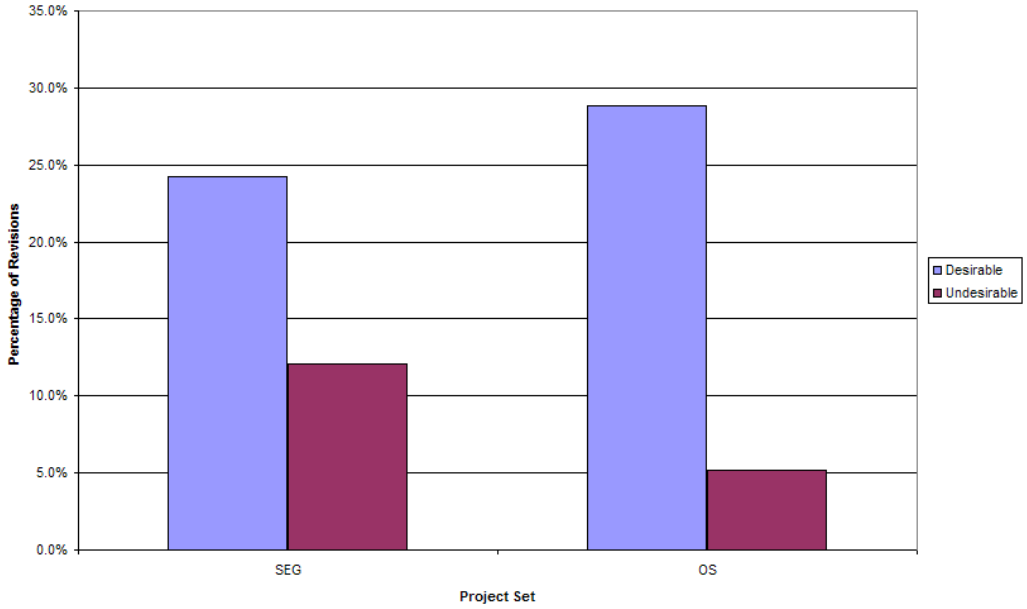


Figure 5.11: Breakdown of desirable and undesirable activities

5.5.5.6 RQTA 2-6: Is Thematic Analysis of Change-logs a Useful and Viable Method for Assessing Collaborative Software Development Projects?

The final purpose of this study is to explore whether or not an analysis of project change-logs can be a valuable resource when assessing a group project. It is hard to assess the development process, and attempts to do so include having students keep logs of their work as they go on, but this simply replicates functionality that already exists in provided support tools. The comparison with real-world projects shows that students should definitely be making better use of the software available as it would support communication and facilitate groupwork. Considering that using descriptive, unambiguous comments during development is demonstrably beneficial to the students and groups, there is no reason why assessment should not make use of these comments instead of an artificial, secondary progress log. At a minimum, change-logs provide a useful insight into the development process and the roles and behaviour of individual students that may not be clear from other

sources of information, and it is recommended that instructors at least read through change-logs when assessing group projects.

While a simple reading of change-logs is beneficial to assessment, a thematic analysis of those same logs is even more so. While requiring a greater commitment from the assessor, a thematic analysis can be performed sufficiently quickly as to be a viable tool. Once the coding scheme is understood by the assessor it is a relatively simple task to perform the analysis, and the results it produces allow for a deep, comprehensive outline of the development and history of the project; this study has looked no deeper than the campus level in the analysis, but it is entirely possible to examine the roles of individual students within a group, allowing for even deeper understanding of collaboration and contribution.

5.5.6 Conclusions

The results of Study 1.2 verify some of the findings of the original study, but fail to verify others. The first study found that students did not employ the practices they were being taught, a finding which has been borne out by this research. In a comparison of the behaviour of the two campuses, the overall differences are the same – the campuses both made contributions commensurate with their respective assessment weightings, while one made much better use of comments than the other. Conversely, the original study found significant differences in the activity distributions, which did not exist in the expanded analysis. In terms of exploring whether or not students make good use of tools to support collaboration, the initial study found that they did not – the aspects of SubVersion designed to support communication were underused and often not understood. In the following year, students made more use of the tools available, but the increased proportion of ambiguous comments lead to the conclusion that while they understood the need to *use* the mechanisms, the students did not know how to use them properly. This research therefore suggests that changes in training from year to year have been beneficial, and that improvements can be made to cover the remaining

problems.

In comparing the SEG projects to open source projects, it was shown that the open source projects were more structured and better commented than the SEG projects. The real-world success of projects such as PuTTY is an indication that the results of this research can be used to reinforce the training students receive regarding development and collaboration practices.

Thematic analysis requires more time and effort than simply reading the change logs, but the insight they can provide into a project – or a set of projects – has been shown to be useful. Because the results confirm many commonly-held beliefs about student work, such as the skewing of effort towards the deadline, this lends weight to the belief that other results found in this study are representative of student projects as well.

This study found that the differences between campuses for the 2007/08 SEG projects did not follow the same pattern as the 2006/07 projects; therefore a study of the 2008/09 projects will be used to investigate this further. Section 5.6 describes a complete analysis of an open source project which allows a more complete comparison between a real-world project and SEG projects. Future research will examine a smaller number of SEG projects in much greater detail to investigate the possibility of uncovering social roles that emerge within groups and can address the question: can the comment log discover facilitators, managers, hard workers, "fixers" and obstructers, or subgroups of students?

5.6 Study 1.3: A Complete Open Source Project Analysis

As noted in Study 1.2 a direct comparison between the SEG projects and the open source projects cannot be directly drawn, as only a subsection of each open source project was analysed. This study performs a thematic analysis of over 5,000 revisions of PuTTY. It should be noted that since PuTTY was, at the time of publication, still being maintained, this is a "complete" analysis

of PuTTY as it existed at the time of the study.

5.6.1 Research Questions

This study addresses the following research questions:

- *RQTA 3-1*: What is the distribution of maintenance types, and does this change over the course of the project?
- *RQTA 3-2*: Is the distribution of activities affected by milestone (e.g. beta) releases?
- *RQTA 3-3*: Do developers take on specific maintenance types?
- *RQTA 3-4*: Is there a relationship between maintenance type and change size?

Where appropriate, these questions will be addressed in comparison with the previously analysed SEG projects.

5.6.2 Thematic Analysis

Initially, the same codes and definitions were used as before (developmental, corrective, administrative, multiple, ambiguous and miscellaneous) but after a preliminary analysis an extra code was introduced - documentation - to reflect the fact that the PuTTY project includes documentation and a copy of the project's website in the repository. Rather than ignore these features, the coding system was extended to include them, as they are in continual development and are an integral part of the project. The revisions categorized as documentation can be disregarded when making comparisons to projects using the previous scheme.

5.6.3 Limitations and Threats to Validity

The primary threat to this study is the fact that only a single project was analysed, which limits the ability to generalize the findings. Further studies are planned which will expand the number of projects analysed.

5.6.4 Evaluation

5.6.4.1 RQTA 3-1: What is the distribution of maintenance types, and does this change over the course of a project?

Figure 5.12 shows the breakdown of activity types for the PuTTY project. Documentation is clearly the most active type, covering nearly 40% of all revisions, followed by corrective and then developmental. The low frequency of miscellanious, multiple and ambiguous revisions matches the results of the previous study. Likewise, the 100% ratio of comments to revisions is continued.

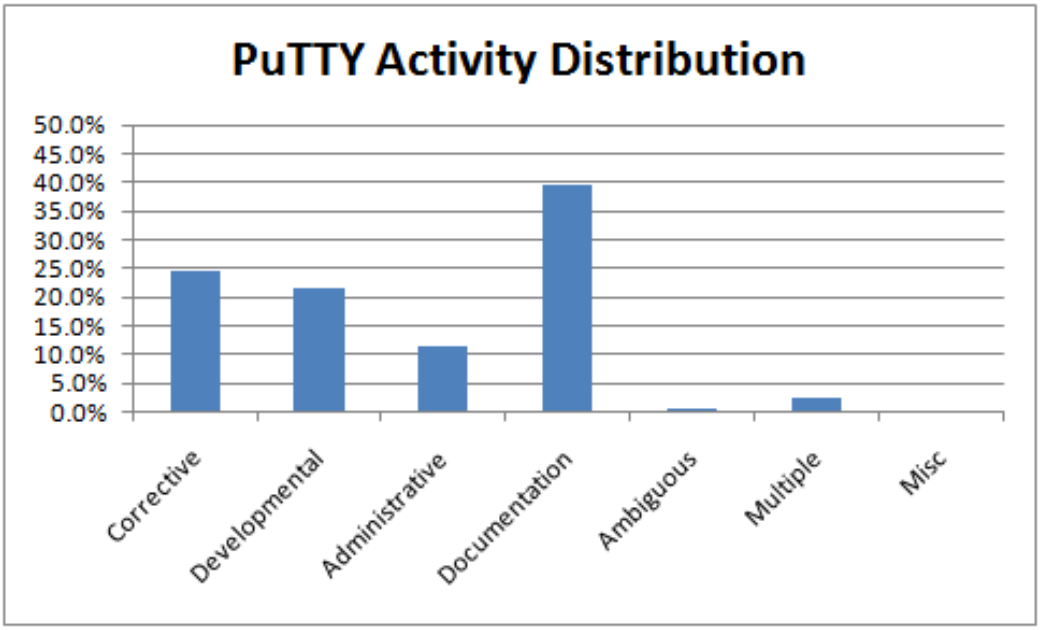


Figure 5.12: Distribution of activity types for PuTTY

Figure 5.13 shows how the distribution of activities changes over the course of the project. Only the main activities - developmental, corrective, administrative and documentation - are included in this chart, which displays the proportion of activities broken into segments of 200 revisions each. It shows that:

- Documentation becomes more and more dominant as the project con-

tinues

- Administrative activities remain relatively constant
- Corrective activities tend to correlate with developmental ones (Pearson correlation coefficient of 0.76)

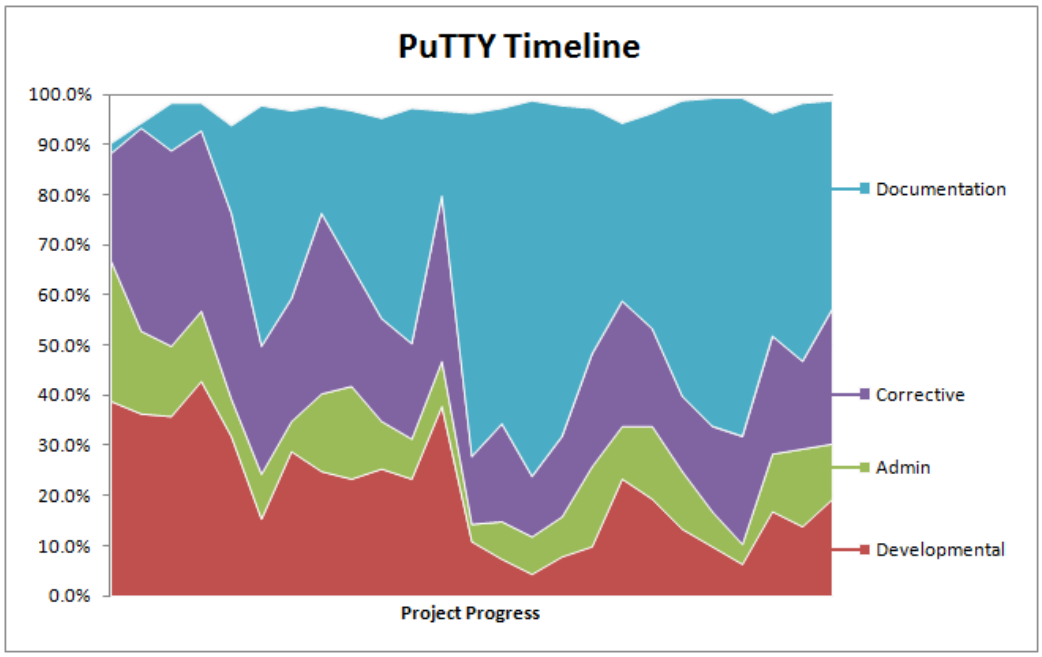


Figure 5.13: How the activity distribution of the primary activities changes over the course of the project

Figure 5.14 shows the cumulative amount of revisions for each activity type over the length of the PuTTY project. Each activity type grows linearly over time, with the exception of documentation which only becomes significant roughly 20% of the way through the project, and quickly dominates all other activity types, becoming almost 40% of the total activity. This documentation includes change-logs, wish-lists, the website, frequently asked questions and the software manual. It is also interesting to note that developmental and corrective activities are given roughly equivalent emphasis until midway through the project, at which point corrective becomes dominant over developmental,

reflecting a strong shift from feature implementation to security, stability and reliability maintenance.

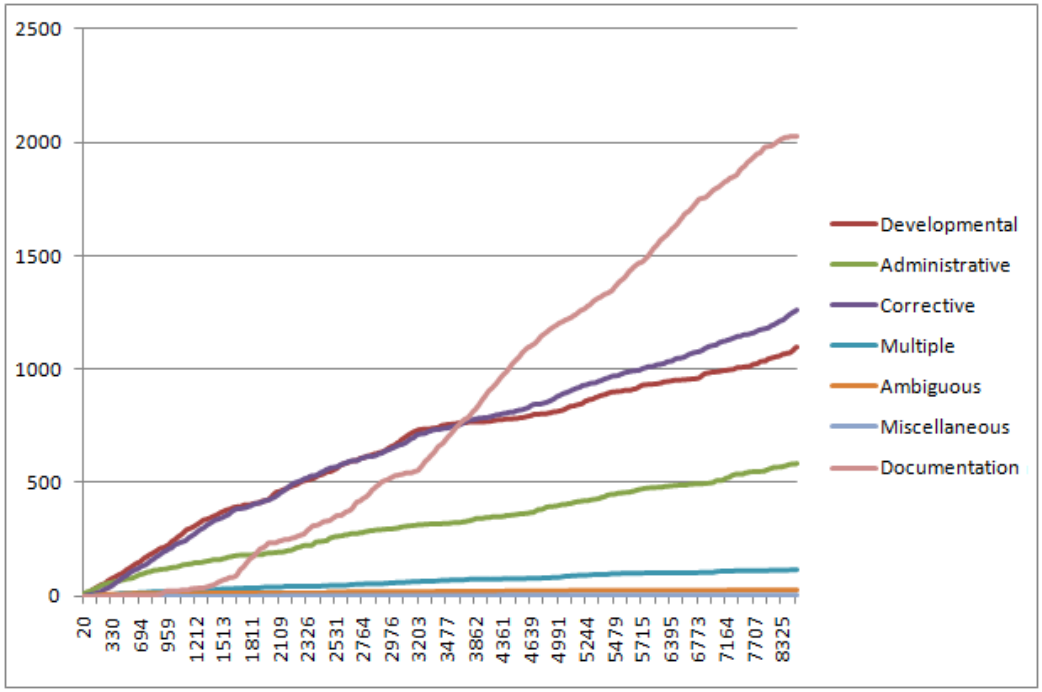


Figure 5.14: Cumulative activity breakdown over the development history of PuTTY

What is perhaps most interesting is the comparison of activity types between the previously reported SEG projects, the partial PuTTY analysis (137 revisions, referred to here as $\mathtt{P}_{137}$) and the complete PuTTY analysis. The SEG projects and $\mathtt{P}_{137}$ show a similar proportion of activity types, particular with respect to developmental (SEG 48%, $\mathtt{P}_{137}$ 49%) and - to a lesser extent - corrective (SEG 11%, $\mathtt{P}_{137}$ 17%). In comparison, PuTTY displays a much reduced proportion of developmental activity (36%) and increased corrective activity (41%). In summary, the initial series of revisions for the PuTTY project have a markedly different distribution to the whole project as developmental activities are replaced by corrective ones. Figure 5.15 compares the activity distributions of the three data sets.

The similarities between the SEG projects and the early revisions of

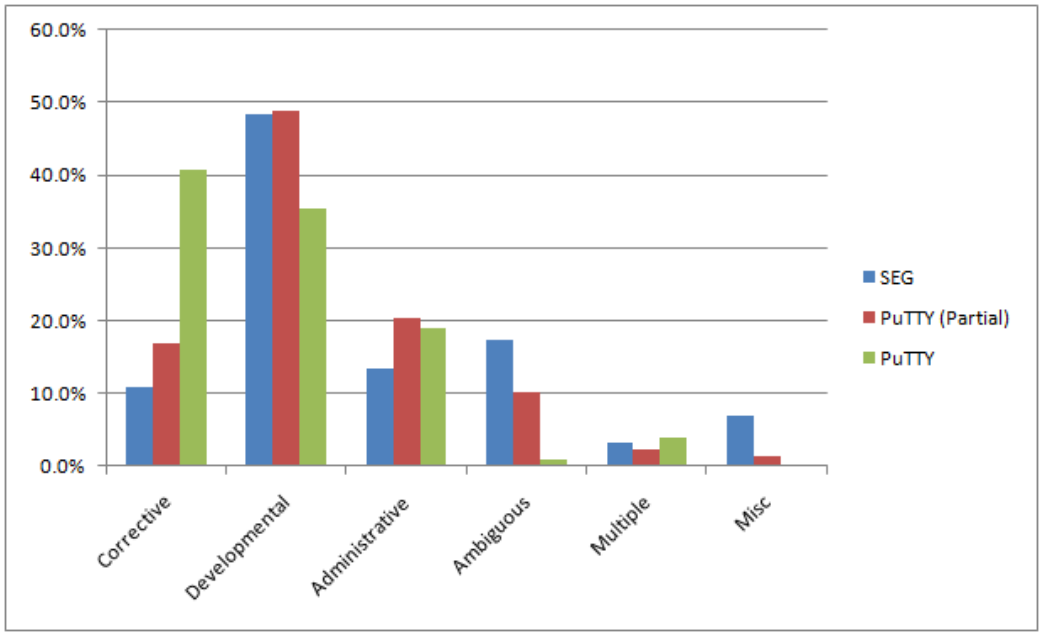


Figure 5.15: Comparison of activity distributions for SEG, PuTTY1 and PuTTY2

PuTTY indicate that the distribution of activities in SEG projects are perhaps simply representative of young projects rather than poorly managed projects. The fact that, over time, PuTTY sees a reduction in ambiguous and miscellaneous activities and an increase in corrective activities would suggest that student projects could follow the same pattern. SEG projects are extremely time-constrained; if they were to continue then we might see the same trends as shown by PuTTY.

Whether this is due to inexperience, or whether it is inherent to young projects cannot be known without further analyses of maintenance activities. There are two main factors to consider: team experience, and project lifespan. The PuTTY/SEG comparison shows the difference between projects of differing lengths, but cannot control for developer experience.

5.6.4.2 RQTA 3-2: Is the distribution of activities affected by milestone releases?

It is interesting to examine the types of activities which precede a milestone, when a tag is created to mark a new release (e.g. revision 180 by cvs2svn, "This commit was manufactured by cvs2svn to create tag 'beta-0-46'."). By examining the activities which occur in the 10 revisions prior to the tag being created, it is possible to see which actions commonly precede a release. The results for both SEG and PuTTY are shown in figure 5.16. Developmental, administrative, corrective and documentation activities remain dominant, but there is an increase in developmental and corrective activities, while documentation is reduced. Interestingly, administrative activities remain constant, and are unaffected by upcoming releases.

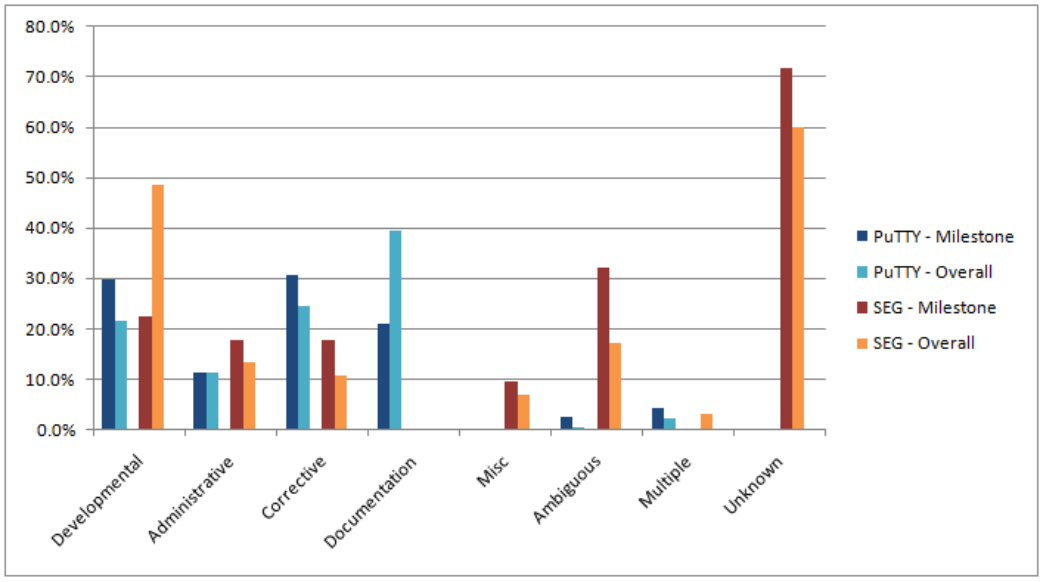


Figure 5.16: Activities preceding project milestones in PuTTY and SEG projects. Overall project values are included for comparison.

This can be compared to the SEG projects, where the milestone is considered to be the final revision, after which the project is submitted for assessment. The SEG projects see a reduction of development activities from 48% overall to 23% in the final 10 revisions, and a doubling of ambiguous

activities from 17% to 32%. Although they are not included in this analysis, it is worth noting that as a percentage of the entire SEG project set, unknown activites (i.e. revisions with no comment) also increase from 60% to 72%. This increase in uncommented and ambiguous comments towards the end of the projects is perhaps indicative of stress, tiredness, haste or a lowered perceived importance of good practice.

5.6.4.3 RQTA 3-3: Do developers take on specific maintenance types?

Figure 5.17 shows how the primary activity types are distributed between the main developers. There are two additional developers not shown, one who only has one revision, and another which is in fact a tool (cvs2svn) responsible for tagging and branching. Figure 5.18 shows how developers activities are broken down as a proportion of their total revisions.

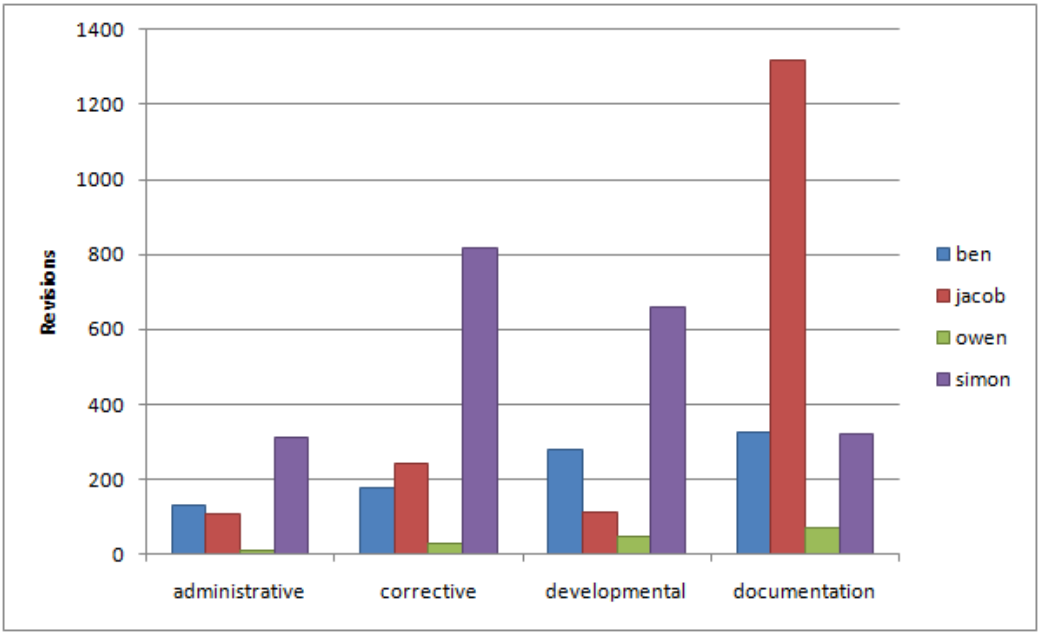


Figure 5.17: Breakdown of activity types by the PuTTY developers.

It can be seen that Simon and Jacob are the most prolific developers, committing 2,200 and 1,800 revisions respectively. Jacob is clearly responsible

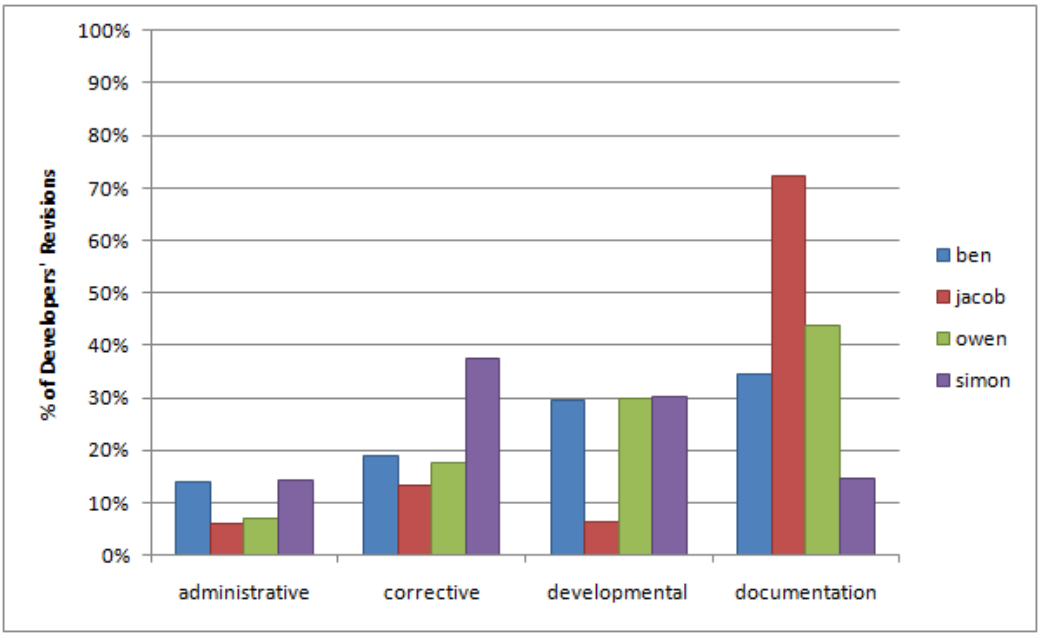


Figure 5.18: Breakdown of activity types by the PuTTY developers as a proportion of each developer's total revisions.

for the documentation, committing almost twice as many documentation revisions as the rest of the team combined, and devoting nearly three-quarters of his revisions to documentation. Similarly, Simon performs far more corrective and developmental maintenance than the rest of the team. However, while Simon commits more corrective revisions as a proportion of his total work, his developmental work is in line with Ben and Owen, each devoting nearly a third of their effort to development. All four developers devote a roughly similar amount of their work to administrative revisions.

There are, in places, definite divisions of labour to be seen in this analysis. This might suggest an equivalent degree of "file ownership", where developers tend to work on a set of files exclusively rather than sharing their development; however, applying a visualization tool[2] to the project (see Figure 5.19) shows that there is a great deal of cross-development of files between developers. This suggests that the developers tend to divide their work based more on

[2]This tool is and its use are described in more detail in Section 7.2.2.3.5.

the type of maintenance rather than the file under development.

5.6.4.4 RQTA 3-4: Is there a relationship between maintenance type and change size?

While there are some trends in the data, such as the fact that corrective changes tend to have a lower than average change/revision ratio, while administrative changes have a higher than average change/revision ratio, there are no significant relationships that allow activity type to be an effective predictor of change size, or visa versa. This is consistent with related workAlali et al. (2008) which attempted to use keyword analysis of revision comments to predict change size but failed to uncover any conclusive relationships.

5.6.5 Conclusions

The main findings of Study 1.3 are:

- The documentation process began some way into the development cycle of PuTTY, but rapidly became the most common activity. There is a correlation between corrective and developmental work; corrective activities are given much more emphasis than in the SEG projects. All maintenance activities have a linear growth, with no signs of change in that trend.
- The early stages of PuTTY's development are not dissimilar to those of the SEG projects. Further analyses of other projects are needed to determine if this is typical.
- As a milestone approaches, there is an increase in developmental and corrective work. This is in contrast to the SEG projects where there is a marked increase in ambiguous and uncommented revisions as the deadline approaches.
- There are indicators that some developers take on specific activity types; one PuTTY developer is responsible for more than half of the

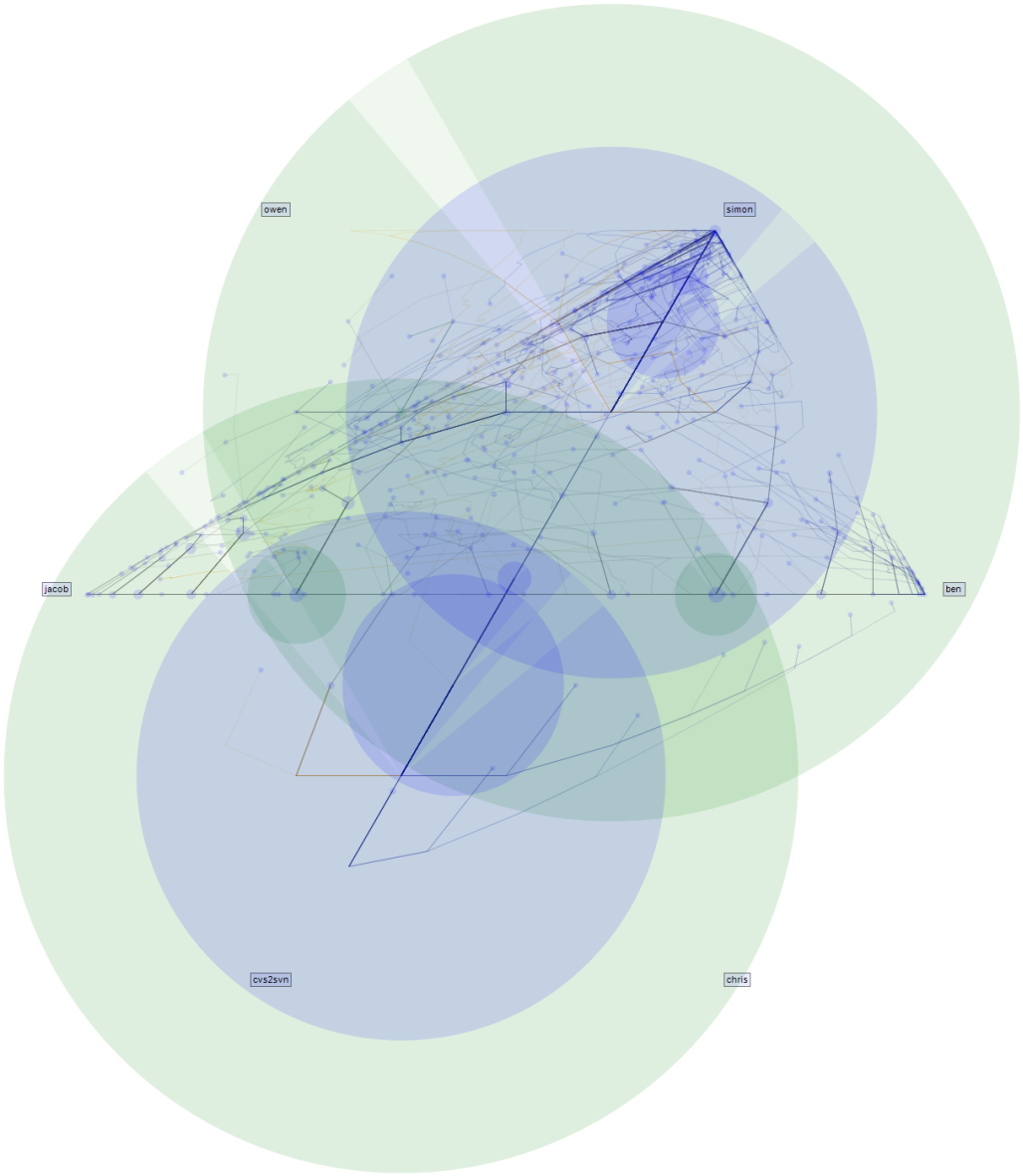


Figure 5.19: Owner Visualization of PuTTY: the large blue circles within green circles represent files which are created an never changed; the lines indicate the degree to which files change ownership. In this case, many files are worked on by more than one developer. See Section 7.2.2.3.5 for details of this visualization.

documentation activities, while another is responsible for half of the corrective and developmental activities. This is in contrast to the lack of file/module ownership in PuTTY - developers take on tasks based more on the nature of the task than the files involved.

- There is no correlation between maintenance activity types and change sizes.

In the context of the broader research question investigating whether or not thematic analysis of RCS repositories is a useful process, this follow-up study has demonstrated that a formal analysis of revision data can provide deeper insight into the nature of software projects and the developers behind them. A common concern regarding thematic analysis is whether or not the process is viable in terms of difficulty and time. As previously reported, the most time-consuming process is deciding on a repeatable and useful set of codes; once this is performed they can be reused with no additional overhead. As shown by this study, the data under analysis can require the codes to be modified, but this will occur naturally and requires little additional effort.

As previously stated, the SEG projects were relatively quick to analyse, with the small size of the data sets balancing the frequently informal nature of the comments. There was a concern that a full open source project would take much longer to analyse, due to the size of the data set and the more formal nature of the comments, which more often require domain specific knowledge. However, the formality of the comments proved beneficial to the analysis, as recurring comments could be classified en-masse, and a more common vocabulary left less room for ambiguity. Overall, PuTTY was more easily classified than the SEG projects, with a similar total number of revisions requiring less time to classify.

5.7 Case Study Discussion

This section aims to draw together the findings of the three thematic analysis sub-studies to address the overall research questions of this case study. These

research questions are restated here:

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data?
- *RQ 3*: Can profiling support project comprehension in student and open source projects?

5.7.1 *Research Question 2*

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data? ***Proven: Yes***

As the literature reviews reported in Chapters 2 and 3 revealed no studies reporting thematic analyses being performed on revision logs, it was not certain whether or not the process, designed primarily for conversational text, could be applied to this data. As has been demonstrated in each of the three studies, there are no practical barriers to performing such analyses. Whether the messages are in the informal, incomplete style of the student projects, or the formal, domain-specific language of the open source projects, the thematic analysis process can be used to extract patterns and information from the logs.

An early concern was the time, effort and training required to perform the analysis. As shown in Study 1.1 (Section 5.4) devising the initial codes can be a time-consuming process involving repeated reviews and revisions. However, once the base set of codes are created it is trivial to re-use it, or to modify it to suit the needs of other analyses. For some of the open source projects it was necessary to spend time gaining an understanding of the domain-specific vocabulary used in the comments before the analysis could begin; this is perhaps the most serious obstacle to using thematic analysis to aid project comprehension. However, learning the vocabulary particular to a project is a necessary step in understanding that project, and will be required of a maintainer regardless of the tasks they seek to perform.

The scale of the project being analysed is an issue: PuTTY, for example, required over 5,000 comments to be classified. However, many of the comments were sufficiently similar that they could be grouped and classified together, saving significant amounts of time and effort. While this technique could not be used on a less formal set of messages, such as those found in the student projects, those projects tended to be smaller and less completely commented, resulting in a smaller set to be classified.

As a direct consequence of the publication of Studies 1.1 and 1.2 (Burn, 2008, 2009), the thematic analysis process has been successfully used in a study to evaluate a new technique for providing students with feedback from programming courses (Cummins et al., 2010).

In summary, regardless of the type or scale of the project, it was possible to use apply thematic analysis to the change logs of a range of projects, following a rigorous, repeatable protocol through every step of the process.

5.7.2 *Research Question 3*

- *RQ 3*: Can profiling support project comprehension in student and open source projects? ***Proven: Yes***

Demonstrating that a process can be applied to a data set is not the same as showing that it is useful to do so. However, the data derived from the three studies has been used to gain an insight into the development processes and practices of both open source and student projects. For example, the first study demonstrated that students do not make good use of provided support tools or taught practices (Burn, 2008); these results were fed forward to the following cohort, which showed improved use of tools and a better (if not perfect) adherence to best practice (Burn, 2009).

Another application of thematic analysis is in the assessment of student projects; the revision messages left by students provide a window into their teams' development processes, collaboration and social cohesion. With thematic analysis shown to be a viable technique it would be possible to use it

to explore more completely students' teamwork and social roles, as well as their technical contributions.

5.8 Case Study Conclusions

The overall goal of this research is to identify and evaluate techniques which use analysis of transactional repository data to support project comprehension, using the following research question:

- RQ 1: How can data mining of revision control systems be applied to support project comprehension?

By proving that thematic analysis can be applied to the comment log of a revision control system, and that the outcomes of the analysis can provide information about the development of a project, thematic analysis has been demonstrated to be a technique which supports project comprehension; thus, one answer to the above research question is:

- Profiling projects using the technique of thematic analysis of historical project data supports project comprehension.

5.9 Summary

This chapter reported a case study in which thematic analysis of student and open source projects was conducted to assess the viability and usefulness of thematic analysis in supporting project comprehension. Thematic analysis was successfully performed on the comment logs of a number of projects, generating useful information in the context of assessment and management, proving that profiling projects using thematic analysis is a technique which can be used to support project comprehension.

The next chapter describes an empirical benchmarking study evaluating the use of revision data to perform history-based change prediction.

Chapter 6

Case Study: History-Based Change Prediction

6.1 Introduction

Chapter 5 explored the use of thematic analysis to profile projects and gain a deeper understanding of their nature, structure and development. This chapter investigates the use of a project's historical data to perform change prediction.

Predicting the effects of a change to a project's source code is a vital skill in software development. A developer familiar with the code will instinctively know what knock-on effects a modification may have. Tools exist (Weiser, 1979; Xu et al., 2005; Gallagher, 1996) to assist developers in performing impact analysis, which typically function using static code analysis, i.e. syntactically examining the source code and inferring programmatic links between software entities, or by examining the behaviour of the software at run-time.

Models based on code-snapshot analysis have a number of drawbacks. Firstly, they are language-dependent, in that they require knowledge of the syntax and structure of a programming language to function. Secondly, they cannot infer links that are not present in the code itself. As described in (Zimmermann et al., 2005), code-snapshot models cannot uncover links

between code and documentation. A study into "change-proneness" (Bieman et al., 2003) demonstrates that the change structure and code structure do not always match.

Another approach to change prediction is to use the history of a project to discover links between software entities. By mining the sequences, groups and patterns of changes over a project's development, relationships can be inferred between entities. As this analysis looks primarily at transactions and files rather than source code, it approaches the problem in a different way and can be used to support results from code-snapshot models.

Zimmermann et al (Zimmermann et al., 2005) have presented a model – ROSE – of performing impact analysis using a combination of code-parsing and history analysis to perform change prediction at the more fine-grained level of variables, functions and classes. The model was shown to be effective at predicting entities which would require changing based on a set of changes, while rarely producing false alarms. When the model was used at a coarser level – at the file level – the effectiveness improved significantly, although the actual results are perforce less useful.

Another tool, Chianti (Ren et al., 2004), analyses the difference between two versions to suggest a subset of regression tests affected by the changes, successfully reducing the number of test-cases to be run following a change. (Ball et al., 1997) demonstrated that the links discovered in history-based analysis identify partitions of classes evident in the project structure. (Weissgerber et al., 2005) visualizes the relationships between files based on their change-histories, demonstrating the benefits of the process to program comprehension. (Cubranic and Murphy, 2003) extends this model beyond change history and uses a range of archived information such as bug reports and forum discussions to generate an implicit group memory and suggest artefacts from that memory in response to a task; a case study identified advantages and disadvantages to the approach – while it provided good "entry points" to a task, newcomers were often confused or misled by the results.

A benchmarking study (Hassan and Holt, 2004) into various change

prediction techniques, including developer-based, entity-based co-change, entity-based code structure and hybrid heuristics demonstrated that a hybrid technique could achieve results "in par with typical information retrieval practical boundaries." These results support those of (Zimmermann et al., 2005), suggesting that history-based techniques are a viable avenue of research.

This chapter uses `Perceive` to measure the performance of history-based change prediction, and to seek ways to improve this performance. The case study is conducted using three sub-studies. The first measures the performance of `Perceive` in conducting change prediction on a series of projects, exploring what factors affect the success of the technique. The second is a more in-depth study of the application of `Perceive` to two individual projects to more accurately assess where history-based change prediction succeeds and fails. The final study seeks to improve the performance of `Perceive` by augmenting the algorithm using data created in Chapter 5, using maintenance activity types to supplement the transactional data used by `Perceive`.

6.2 Case Study Design

This section details the goals, techniques and evaluation of the case study using the DECIDE framework. As described above, this case study uses three sub-studies as its subjects, and draws its conclusions from the outcomes of those studies.

Figure 6.1 shows the structure of this case study, how the sub-studies feed into the case study and how the case study feeds back to the overall research question. The three sub-studies themselves form a piece of research that can stand alone; however, the results and experiences are used to evaluate the use of repository analysis in change prediction. The case study will use that evaluation to assess whether or not repository analysis can make a useful contribution to project contribution in this context.

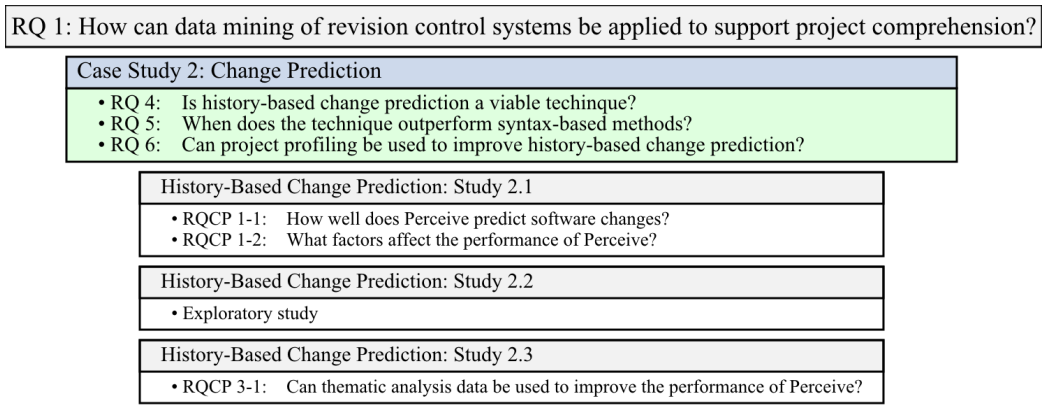


Figure 6.1: Structure of the research questions and studies of the change prediction case study

6.2.1 Research Goals

The goals of this case study are to measure the performance of history-based change prediction, to explore the factors which affect its success, and to seek to improve the model using data generated by the thematic analyses in Chapter 5.

The first study (*Study 2.1*, Section 6.4) will provide a baseline measure of performance, which can be compared with other, existing research, such as (Zimmermann et al., 2005), (Zhou et al., 2008) or (Kagdi et al., 2007b). By focussing closely on the application of `Perceive` to two projects, the second study (*Study 2.2*, Section 6.5) will explore what affects the performance of history-based change prediction, which leads into the third study (*Study 2.3*, Section 6.6) which will seek to improve on the performance baseline established in the first study by incorporating data generated in Chapter 5.

If the third study can improve the performance of history-based change prediction, then maintenance activity types can be added to the set of factors which can be used when performing history-based change prediction.

6.2.2 Research Questions

The overall goal of this research is to identify techniques based on repository analysis which support project comprehension. Therefore, history-based change prediction must be evaluated in this context. The following research questions have been identified to assess whether change prediction is a process whereby repository analysis can support project comprehension:

- *RQ 4*: Is history-based change prediction a viable technique?
- *RQ 5*: When does the technique outperform syntax-based methods?
- *RQ 6*: Can project profiling be used to improve history-based change prediction?

Table 6.1 lays out which research questions are addressed by which study.

Research Question	Addressed By...
RQ 4	2.1, 2.2
RQ 5	2.2
RQ 6	2.3

Table 6.1: Map of research questions to studies

RQ 4 and *RQ 5* are designed to reinforce or refute existing research which states that history-based change prediction can successfully support project comprehension; by evaluating aspects of the technique which have not been explicitly researched thus far the suitability of history-based change prediction can be assessed. *RQ 6* goes beyond existing research to evaluate whether or not thematic analysis can be used to *improve* the performance of history-based change prediction. If the answer to *RQ 6* is proven to be positive then history-based change prediction will be a proven technique by which repository analysis can be used to support project comprehension.

6.2.3 Evaluation Paradigm and Techniques

Study 2.1 is an empirical benchmarking study, which uses information retrieval metrics to measure the performance of a series of permutations of change prediction models. Due to differences in implementation and test projects, it is difficult to directly compare the performance of `Perceive` with existing studies, although the results will be compared with these. The performance of each model configuration and application will be measured relative to the other configurations, allowing for an investigation into which parameters and configurations perform most strongly in which situation.

Study 2.2 is more exploratory and qualitative in nature, with empirical measurements being a secondary outcome. The study primarily aims to assess what factors affect the application of history-based change prediction, highlighting use-cases and investigating where the performance of a traditional, syntax-based technique might differ from a history-based one.

Study 2.3 is another empirical benchmarking study, whose outcomes will be evaluated in a direct comparison with the baseline established in Study 2.1.

6.2.4 Practical Issues

The only practical issue with this case study is the selection of projects. As with the Thematic Analysis studies two sets of projects will be used: three open source projects and 35 student group projects. As the change-prediction techniques applied in this chapter are automated, there are no scale limitations on the projects, and so unlike Chapter 5 the entirety of each open source project will be used, rather than a partial sample.

Each of these projects was stored in a SubVersion repository. In such repositories, a change is a modification to a single file, including creating or deleting the file, changing its contents, copying it or changing its properties. A revision is one or more changes committed to the repository at once. In previous research (Zimmermann et al., 2005) it was necessary to create

transactions (revisions) algorithmically as the test objects were from CVS repositories which stored each change individually.

In this research, only changes involving modifications to files are used – creation, deletion and copying are not included, as change prediction is primarily concerned with changes to code as opposed to creation or deletion of files.

The test projects vary in size, maturity and duration – some of the academic projects consist of less than a hundred revisions, whereas one open-source project had over 13,000 revisions. It is expected that there will be differences in the results between these groups of projects – the academic ones are much shorter and as such have less scope for relationships to be inferred (conversely, they also have fewer files, which serves to mitigate this somewhat); more importantly, there are differences in programming styles and behaviour – the open source projects have a much smaller change-to-revision ratio than the academic projects. Such differences have been predicted to have an effect on the effectiveness of history-based approaches, as have the "quality" of revisions (Zimmermann et al., 2005).

6.2.4.1 Outcome Measures: Information Retrieval Metrics

As with previous research (Zimmermann et al., 2005), precision and recall will be used to assess the effectiveness of the change prediction. Precision is the accuracy of the suggestions, the proportion of suggestions that are accurate, while recall is the proportion of the correct results which were returned. It is trivial to construct a model with perfect recall – simply suggest every entity in the project. Likewise, perfect precision is attainable by making no suggestions at all. It is evident therefore that a good model will seek to optimize both precision and recall, potentially requiring that one be prioritized over the other. To this end, the F-Score measure is used, a weighted average of the precision and recall. The F-Score can be modified to prioritize precision over recall, or vice versa. Common variants on the F-Score are F_2, which weighs recall twice as much as precision, and $F_{0.5}$, which weighs precision twice as

much as recall.

A proposed benchmarking framework (Lessmann et al., 2008) discusses the difficulties of comparing classification models, especially ones with variable input parameters and threshold variables. Receiver operating characteristics (ROC) curves are proposed as an effective visual method for comparing classification models, with area under ROC curve (AUC) as a secondary method. These methods require the models to be tested across a complete range of threshold values to produce comparable curves. As the models in this case study do not have an upper bound on all of their input values, these methods can be applied. However, they highlight the reality that comparison of information retrieval based studies is a complex matter and care must be taken when making assertions based on the outcomes.

6.2.4.2 Measuring Success

To measure the precision and recall of a model, a set of known good results must be available. This is achieved by assuming that the actual set of files modified in each revision is correct – i.e. if a revision modifies files A, B and C, then a correct prediction will be those three files, no more or less. This assumption is problematic in that it may not actually be correct in many cases. However, there is no realistic process by which absolutely correct sets of files can be generated. Comparisons with results from other tools is viable, but as stated previously, different methods investigate different partitions of files. Using developer expertise is another method, but these developers already used their expertise to make the revisions in the first place. It can be argued that they would learn as they gain experience, in which case the relationships between files would change over time. Future research will attempt to address this problem using a combination of code analysis tools and developer feedback. However, this measure has been used in other research (Zimmermann et al., 2005), and so is considered suitable for use here.

6.2.5 Ethical Issues

The open source projects used in this case study are freely available on the internet, whereas the copyright for the student projects is owned by the universities involved. Therefore, ethical clearance was sought and granted to use the student projects, following sufficient anonymisation. As no human subjects were involved with this case study, no further ethical clearance was necessary.

6.2.6 Evaluation and Discussion of Results

After each of the three studies are conducted, the findings will be evaluated in the context of the overall research questions and goals, to relate each analysis to the overarching aim of assessing and improving the performance of history-based change prediction.

6.3 Perceive

This research uses the tool described in Chapter 4 to analyse project repositories and perform history-based change prediction using only the transactional data, as opposed to analysing the source code and using a syntax-based technique.

Change prediction involves a model, given one more changes as an input, calculating the changes which might occur as a result. In the case of history-based change prediction, the inputs are a set of code entities which have changed, and suggesting a set of code entities which might therefore also require changing. Depending on the model, the code entities might range from the fine-grained (e.g. variables or functions) to the coarse-grained (e.g. files). `Perceive` uses files as its code entities, as this allows the model to remain language- and platform-agnostic - no additional code is required to allow `Perceive` to function on different languages; this also has the benefit that non-source code files can also be included in the prediction process.

The costs and benefits of this approach have been explored in other research (Zimmermann et al., 2005), and the benefits of being language-agnostic are seen as outweighing the drawbacks of the coarse-grained results. In the context of this case study, all models work on a file-level, and so the results can be considered relative to one another.

Perceive functions by maintaining a network of files, with edge weights being the number of revisions in which the pair of files were both modified. For any given file, A, therefore, the edges from that file can be used to make suggestions. Two metrics are used to make these suggestions – *support* and *confidence*. For each linked file, B, support is the number of times A and B were modified together, and confidence is support divided by the total number of modifications of A. By modifying the required thresholds of support and confidence before a link can be inferred, the precision and recall can be affected – investigating the effects of changing these parameters is one of the goals of this research.

Because Perceive, by its nature, cannot make change predictions early in a project, the first half of each project is used to build a training set, while the analysis is conducted using the latter half.

6.4 Study 2.1: Empirical Benchmarking Study

This section describes the first stage of this case study, an empirical benchmarking study, in which `Perceive` is applied to a series of projects with a variety of parameters to measure the model's performance in terms of information retrieval. The study has the goals of confirming or refuting existing research which states that history-based change-prediction is a viable technique and exploring which factors affect performance.

6.4.1 Research Questions

The research seeks to address the following research questions and associated hypotheses:

- *RQCP 1-1*: How well does Perceive predict software changes?
  - *HCP 1-1*: Perceive is a better change-predictor then a random control model.
- *RQCP 1-2*: What factors affect the performance of Perceive?
  - *HCP 1-2*: Perceive performs better on projects with more revisions.
  - *HCP 1-3*: Perceive performs better on projects with more files.
  - *HCP 1-4*: Perceive performs better on open-source projects than academic projects.

This study also has an exploratory aspect, addressing the following issues:

- Perceive can be used with a variety of parameters, which will have an effect on the accuracy and completeness of the results. This research will explore the effects of these parameters, without making predictions as to the outcomes.
- It is expected that there will be situations where a history-based model cannot reliably predict changes. This research aims to explore the effects of these "blind spots".

6.4.2 The Models

A set of models were devised to provide a comparison with `Perceive`.

- `Random`: This model simply selects a set of files at random from the n available files, ranging from 0 to n. This model was predicted to be the worst, and was devised as a naive baseline comparison.
- $\texttt{Random}_2$: Like `Random` this model selects random files, but limits the number to the average change-per-revision thus far. This model was expected to be better than `Random` as it would have higher precision without much reduction of recall.

- `Frequency` (Frequency-based): This model selects a number of files from the available pool based on how much they have been modified thus far. By simply suggesting a number of the most active files, it was expected that the results would be no worse than $\texttt{Random}_2$, and potentially significantly better, depending on the project.

As these three models are non-deterministic, i.e. they each contain a random element, they are run multiple times and an aggregate of the results is reported. These models are intended to provide control groups against which to compare the performance of `Perceive`, and are not presented as viable change-prediction strategies in their own rights. This allows the performance of `Perceive` to be statistically compared to control models rather than simply stated as absolute metric values. Therefore, the statistical significance testing allows `Perceive` to be evaluated in terms of "Does the technique perform better than a control model?".

6.4.3 Phases

The experiment is conducted in two phases, which measure the performance of history-based change prediction in two different use-cases.

6.4.3.1 Phase 1

This phase investigates the ability of each model to correctly predict all files that will require changing based on a single file. For every revision, a set of suggestions is made from each change in that revision. In this way, the precision and recall can be measured in detail for every file in every revision in every project. This phase is simply investigating the general predictive ability of each model; as the aim is to predict an entire revision from a single file, the success rate is expected to be low. However, relative success of each model to the others is the important measure.

Phase 1 will be conducted using all four models: `Perceive`, `Random`, $\texttt{Random}_2$ and `Frequency`.

6.4.3.2 Phase 2

Phase 2 explores the ability of Perceive to identify a missing change from each revision. For every revision, change prediction will be performed once for each file, to try and predict that it is the missing file. This phase aims to measure the success of Perceive in a real-world use-case, that of suggesting a missed change to a developer. This is similar to an experiment conducted in (Zimmermann et al., 2005), and is intended to contribute to the existing set of results as opposed to providing a novel technique.

Phase 2 will be conducted using four predictive models – `Perceive`, `Frequency`, `Random` and a second variant of `Perceive`, named $\texttt{Perceive}_2$. $\texttt{Random}_2$ is omitted as the differences between it and `Random` are examined in sufficient detail in Phase 1. In this Phase `Random` and `Frequency` are designed to generate only a single result, while `Perceive` functions in the same manner as Phase 1, suggesting any file which fulfils the criteria. To account for the difference between `Perceive` and the control models in this phase, $\texttt{Perceive}_2$ is introduced.

- $\texttt{Perceive}_2$: This model functions in the same manner as `Perceive`, but only suggests the file with the highest confidence, using support to resolve ties. This variant is used to provide a closer comparison of performance to the control models in this phase, while still allowing the performance of the primary `Perceive` model to be evaluated.

Both phases will be conducted repeatedly, measuring the effects of the support and confidence parameters. In addition, each phase will be conducted twice for each project. The first run will select only source code files, while the second run will use code, documentation and data files. The aim is to investigate whether Perceive is more effective when focussed on code alone, or whether it is possible to broaden the application.

6.4.4 Implementation Details

During the implementation and prototyping of the models, a number of issues were encountered:

- If no files are suggested, precision is 1.
- If no files are expected, recall is 1.

As the parameters became stricter, fewer files were suggested and therefore the precision tended towards 1. This was combined with the fact that a large number of revisions consist of a single change, and therefore the recall was always 1 in a non-trivial number of cases, regardless of model or configuration. This led to the dilemma that there was a large portion of the data in which a perfect model was indistinguishable from a failed model; to circumvent this problem only revisions where more than one file was changed, and at least one suggestion was made were counted. The effects of this decision are monitored through the recording of the number of revisions skipped. This has led to the introduction of a new measure:

> *Coverage*: The percentage of revisions for which valid suggestions *can* be made in which suggestions *were* made.

This metric is used only in Phase 1, as Phase 2 approaches the problem in a different way. In Phase 2 only revisions which can have a suggestion made are measured, so if a model fails to make a suggestion then the model has failed in that revision. Therefore, the results will be measured in two ways, using an additional parameter – the *fail-weight* (FW_i), where i can be either 1 or 0. This parameter is used to determine the precision in the case that no suggestion is made:

- If no files are suggested, precision is FW.

FW_1 is correct in the definition of precision, whereas FW_0 is correct in that the model has failed to produce an accurate answer.

As stated in Section 6.4.2 the `Random`, $\texttt{Random}_2$ and `Frequency` are non-deterministic, and as such require running multiple times to produce aggregated results. In practice they were run 1,000 times, the most possible considering performance and time constraints. Analysis of two separate 1,000-run sets shows very little variation between the results – precision has a mean difference of 0.002 (standard deviation 0.003), while recall has a mean difference of 0.001 (standard deviation 0.001).

6.4.5 Limitations and Threats to Validity

The primary limitation of this research is the lack of comparison with syntax-based impact analysis or change prediction tools. While demonstrating the effectiveness of Perceive against a random control and against user data is useful from an academic viewpoint, comparisons with tools which are used in actual development environments are necessary to provide results likely to have value to industry practitioners. However, this research builds on existing work in the field of history-based change prediction and the findings contribute to that field.

As described in Section 6.4.4 a number of design decisions had to be made which directly affected the results. The rationale behind these decisions has been explained, but it must be borne in mind that the effects described mean that any history-based analysis model seeking to be complete can give perfect precision simply by failing, and that some projects will necessarily lend themselves towards high recall. It is vital that any investigation or model makes plain its handling of these cases, or no comparison with similar research can be made.

The aim was to draw on as many projects as possible from which to generate results. While there are a large amount of projects, the factors investigated in RQCP2 are somewhat intertwined (e.g. the open-source projects tend to have more revisions), and while efforts have been made to account for this, there are simply not enough projects to be able to do so with complete confidence.

6.4.6 Results

This section summarizes the results for each phase. Only a brief overview of the data is included – the full set of results is available in separate documents.

6.4.6.1 Phase 1

This phase consists of two sets of results, depending on the categories of file being included in the analyses. The first set are code files only, while the second set are code, document and data files.

Model	Coverage	Precision	Recall
`Perceive`	0.293	0.662	0.219
`Random`	1.000	0.463	0.620
$\texttt{Random}_2$	0.696	0.503	0.123
`Frequency`	0.632	0.500	0.140

Table 6.2: Phase 1: Overview

Model	Coverage	Precision	Recall
`Perceive`	0.293	0.656	0.220
`Random`	1.000	0.473	0.621
$\texttt{Random}_2$	0.684	0.498	0.127
`Frequency`	0.624	0.495	0.142

Table 6.3: Overall performance of all models: Phase 1, code files

Table 6.2 gives an overview of the results for Phase 1 broken down by model. Table 6.3 shows the overall results for each model, aggregated across all projects, when only code files are analysed. Table 6.4 shows the same aggregated results when code, data and document files are analysed. Tables 6.5 and 6.6 break the results down into the two sets of projects – academic and open source.

Model	Coverage	Precision	Recall
`Perceive`	0.293	0.669	0.218
`Random`	1.000	0.454	0.618
$\texttt{Random}_2$	0.707	0.507	0.120
`Frequency`	0.639	0.506	0.138

Table 6.4: Overall performance of all models: Phase 1, code document and data files

	Academic Projects			FOSS Projects		
Model	Coverage	Precision	Recall	Coverage	Precision	Recall
`Perceive`	0.281	0.644	0.216	0.433	0.802	0.268
`Random`	1.000	0.483	0.623	1.000	0.360	0.599
$\texttt{Random}_2$	0.671	0.490	0.128	0.837	0.595	0.104
`Frequency`	0.606	0.487	0.145	0.837	0.595	0.114

Table 6.5: Overall performance of all models, by project category: Phase 1, code files

	Academic Projects			FOSS Projects		
Model	Revisions	Precision	Recall	Revisions	Precision	Recall
`Perceive`	0.282	0.657	0.214	0.422	0.804	0.265
`Random`	1.000	0.461	0.619	1.000	0.373	0.608
$\texttt{Random}_2$	0.696	0.500	0.121	0.835	0.597	0.115
`Frequency`	0.622	0.498	0.139	0.835	0.599	0.122

Table 6.6: Overall performance of all models by project category: Phase 1, code document and data files

6.4.6.2 Phase 2

As with Phase 1, there are two sets of results, one for code files only, and one for code, document and data files.

Table 6.8 shows the overall mean precision and recall over every project and configuration for Phase 2. Table 6.9 breaks this data down by the file selection, giving precision and recall for each model depending on whether the file set was code or code, documents and data.

Table 6.10 shows the differing performance of each model aggregated across project categories. Table 6.11 shows how the FW parameter affects the results.

6.4.7 Evaluation

6.4.7.1 Phase 1

Phase 1 seeks to assess the predictive ability of any given file when using history-based analysis. The overall results, as shown in Tables 6.3 and 6.4, demonstrate that there is no model with highest performance across all metrics. Of the control models, `Random` has the worst precision by a narrow margin, but the best recall – over four times better than `Frequency`. `Perceive` has significantly better precision than `Random`, but poor recall – better than $\texttt{Random}_2$ and `Frequency` but far worse than `Random`. Statistical testing, using a significance value (α) of 0.005, shows these conclusions to be statistically significant.

The low precision and high recall of `Random` was predicted – by simply

	Code Files			Code, Documents and Data		
	Precision	Recall	Coverage	Precision	Recall	Coverage
Revisions	0.25	0.27	0.45	0.24	0.27	0.44
Files	0.24	0.16	0.20	0.27	0.22	0.16

Table 6.7: Correlation between project factors and `Perceive` performance

Model	Precision	Recall
`Perceive`	0.411	0.170
$\texttt{Perceive}_2$	0.435	0.093
`Random`	0.016	0.016
`Frequency`	0.099	0.099

Table 6.8: Phase 2: Overview

	Code		Code, Documents and Data	
Model	Precision	Recall	Precision	Recall
`Perceive`	0.418	0.166	0.404	0.173
$\texttt{Perceive}_2$	0.439	0.091	0.431	0.094
`Random`	0.020	0.020	0.012	0.012
`Frequency`	0.108	0.108	0.089	0.089

Table 6.9: Phase 2: Effects of file selection

	Academic		FOSS	
Model	Precision	Recall	Precision	Recall
`Perceive`	0.410	0.170	0.421	0.170
$\texttt{Perceive}_2$	0.434	0.092	0.446	0.099
`Random`	0.017	0.017	0.002	0.002
`Frequency`	0.102	0.102	0.058	0.058

Table 6.10: Phase 2: Effects of project category

Model	FW_1 Precision	FW_0 Precision
`Perceive`	0.754	0.068
$\texttt{Perceive}_2$	0.778	0.093
`Random`	0.016	0.016
`Frequency`	0.099	0.099

Table 6.11: Phase 2: Effects of fail-weight (FW)

		Code Files		Code, Documents and Data	
		Precision	Recall	Precision	Recall
`Perceive`	Revisions	-0.17	0.14	-0.09	0.11
	Files	0.05	-0.02	-0.09	0.16
$\texttt{Perceive}_2$	Revisions	-0.13	0.16	-0.07	0.14
	Files	0.06	0.08	-0.06	0.23

Table 6.12: Correlation between project factors and `Perceive` performance

choosing a large number of files it can easily produce high recall, at the cost of precision. Likewise, $\texttt{Random}_2$ and `Frequency` are designed to choose smaller amount of files with the intention of improving precision. The reduction in recall was expected, but surprisingly their precision was almost identical. `Frequency` was predicted to have a better precision than $\texttt{Random}_2$ simply by choosing the most active files, but in practice this was not the case.

In terms of file selection, the types and number of files selected for analysis had little effect on the control models, and resulted in a slight improvement to `Perceive`'s precision. From this it can be concluded that expanding the set of files for analysis in no way negatively impacts the performance of `Perceive` means that the benefits of content-agnostic analysis can be applied to an entire project, not just source code, regardless of language or format.

Breaking the results down by project categories (academic and open source) as shown in Tables 6.5 and 6.6 it can been seen that the open source projects, which are larger and maintained by more experienced developers, tend to receive better precision and recall from `Perceive` than do the academic projects. It can also be seen that `Random` performs worse on the larger projects, since it has more files to choose from and more scope to make errors, while $\texttt{Random}_2$ and `Frequency` actually perform better on the larger projects. This is because both models select a random number of files in each revision up to the average number of changes per revision by that stage in the project. Until a number of revisions have passed and the average number of changes per revision has increased, it is more likely that the model will select no files,

and thus the revision will not be counted.

It is worthwhile noting that in every case `Perceive` processes far fewer revisions than the other models. There are two reasons for this. Firstly, a number of revisions are used to create a training set and typically only the latter half of a project is analysed. This is to allow the model to develop the links between files, and reflects a scenario where is it being used during maintenance, or later in a project, rather than in its early days. Secondly, as discussed in Section 6.4.4, a decision was made to ignore revisions where no suggestions were made. The practical effects of this decision are discussed in Section 6.4.7.2.

6.4.7.1.1 Effects of Parameters on Perceive

Overall, it has been demonstrated that `Perceive` outperforms the control models in terms of Precision in every case, but is inferior to `Random` in terms of recall. This section explores how modifying the parameters of `Perceive` affects the precision and recall of the model. The following results use the expanded file set (code, documents and data files) as this has been shown to have, at worst, no impact on the results.

To describe a particular combination of support and confidence, the following notation will be used:

- $P_{s,c}$, where:
  - P is the $F_{0.5}$ score of `Perceive`
  - s is the support parameter
  - c is the confidence parameter

Figure 6.2 shows the aggregated precision, recall and coverage of the academic projects for different support values. It was expected that as support rose, precision would increase as recall fell, due to the increased number of revisions files would be required to have appeared in before `Perceive` would infer a link. However, as the recall does fall as predicted, the precision rises

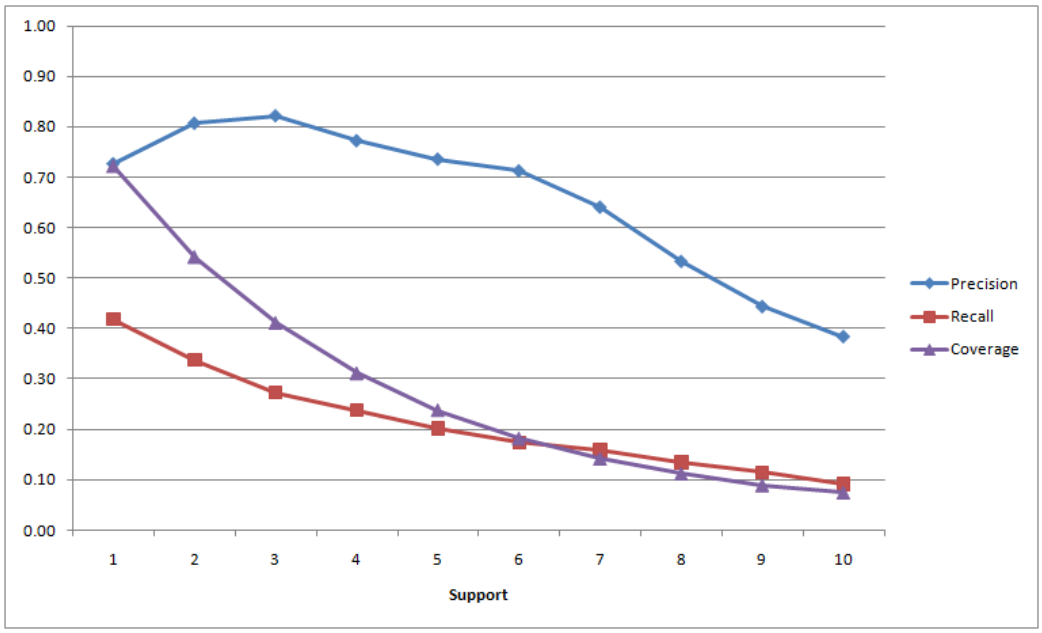


Figure 6.2: The effects on precision, recall and coverage of changing the support parameter on academic projects

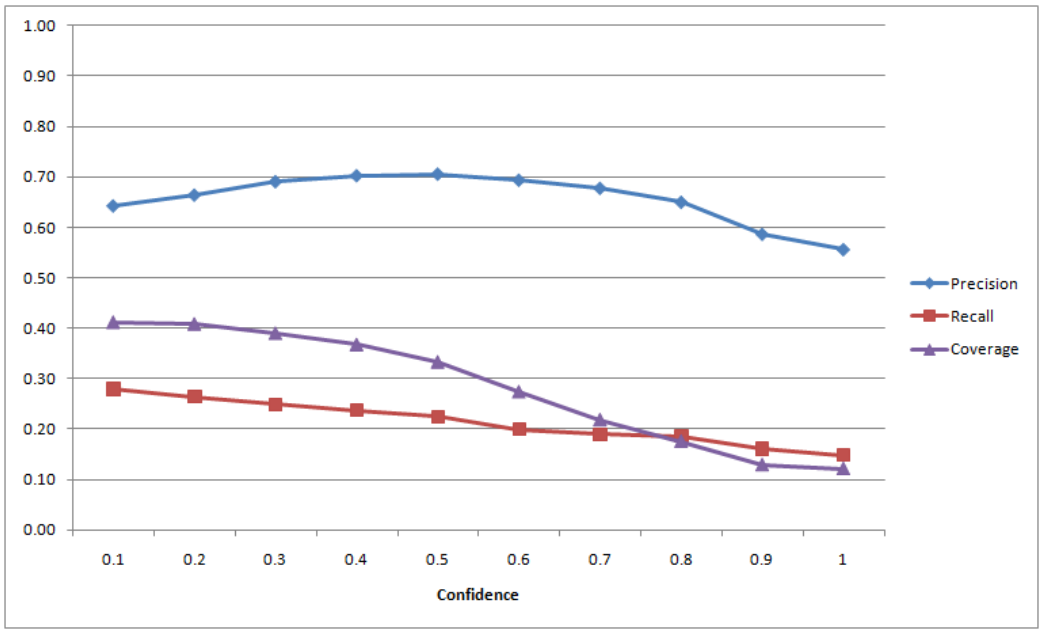


Figure 6.3: The effects on precision, recall and coverage of changing the confidence parameter on academic projects

briefly then too begins to fall. The revision coverage also falls, as predicted. From this set of results it can be inferred that, for smaller projects, a low support parameter is required to make good change predictions.

Figure 6.3 shows the effects of changing the confidence parameter. The trends are much the same as for Figure 6.2, although the magnitude of the effects is smaller.

By combining these results it can be proposed that a lower support parameter, around two or three, and a mid-range confidence, 0.5 to 0.6, might produce the best results. Examining the individual results for the support/confidence combinations, the absolute best (in terms of $F_{0.5}$-Score) combination for the academic projects is $P_{1,0.8}$=0.649. However, this value has a coverage of 0.56, meaning that almost 50% of the revisions are not analysed. In a real-world scenario this could negatively impact the usefulness of the model. If we aim to achieve a coverage of 0.75 – e.g. results are generated for three-quarters of the revisions – then the best combination is $P_{1,0.5}$, with an $F_{0.5}$-Score of 0.632. The absolute best coverage is, like recall, at $P_{1,0.1}$. If the user decides that precision is the most important metric, then $P_{2,0.8}$ provides the best precision, but relatively poor recall (0.29) and coverage (0.36). It is clear that depending on the application and the user, a trade-off between precision, recall and coverage will be required. From a practical perspective, a high recall is important as the model should suggest as many of the correct answers as possible. However, this recall comes at the expense of precision, meaning a lot of false-positives will be made, tending towards providing the user with nothing more useful than a complete list of files in the project. From a usability perspective avoiding excessive false-positives is important, and sacrificing recall for precision might in fact be the correct decision, as suggested in (Zimmermann et al., 2005).

Figures 6.4 and 6.5 show the effects of support and confidence on precision, recall and coverage in the FOSS projects. The precision and recall in both cases follow the same pattern as for the academic projects, although they are typically higher in this case. Precision rises slightly with support, but then

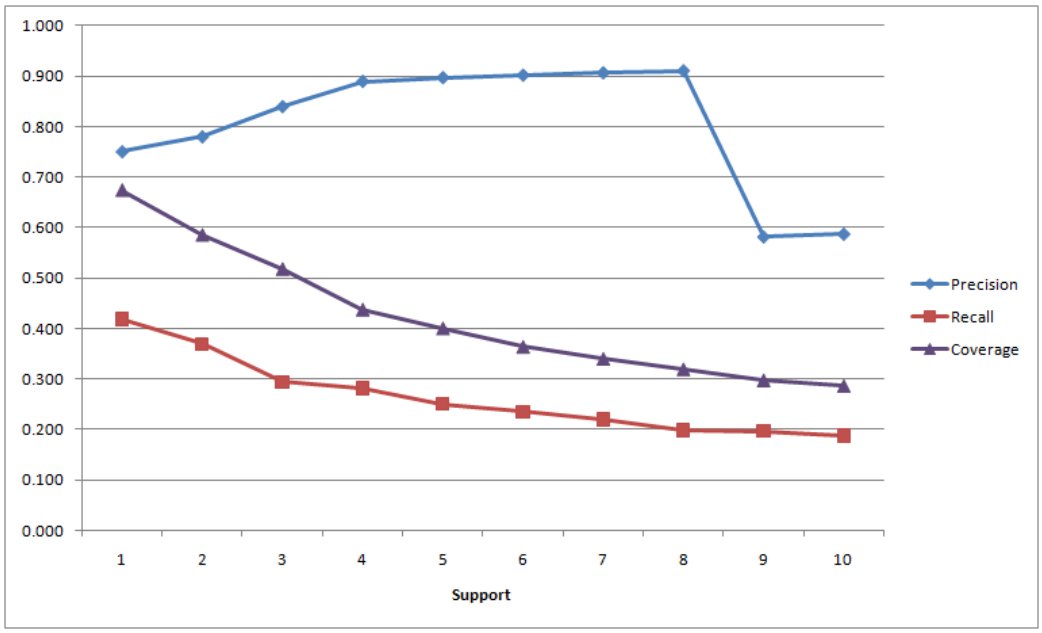


Figure 6.4: The effects on precision, recall and coverage of changing the support parameter on open source projects

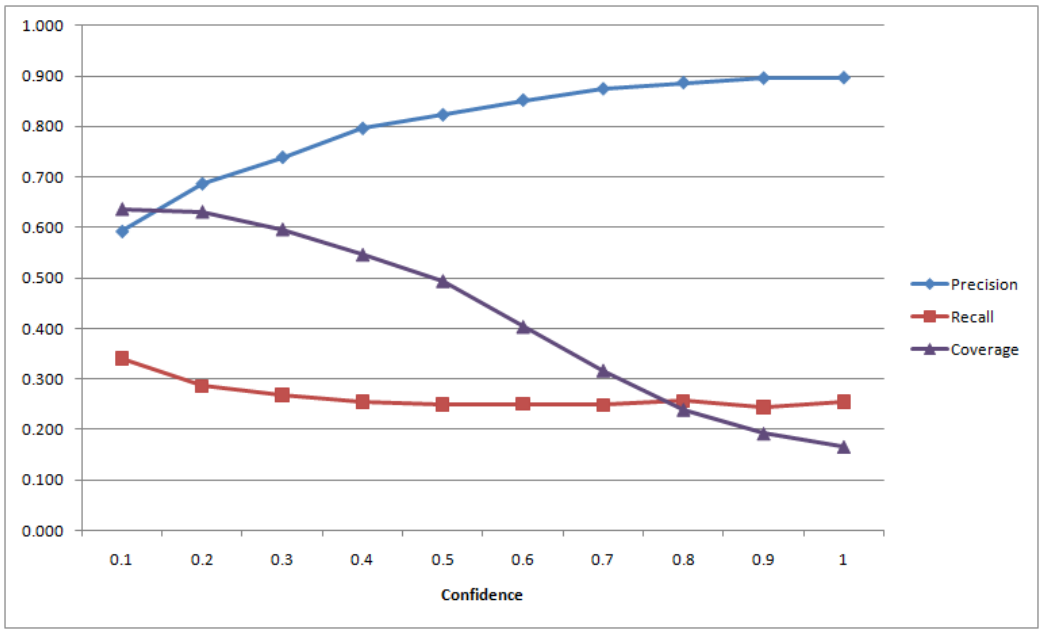


Figure 6.5: The effects on precision, recall and coverage of changing the confidence parameter on open source projects

plummets once support reaches 9 – this is due to one of the projects, shorter than the others, having zero coverage at that point, as there are simply too few revisions to infer links with a support parameter that high. Increasing the confidence, however, has no such effect and the precision increases smoothly as confidence is raised. As before, increasing the confidence parameter reduces coverage significantly, but recall remains fairly constant in comparison to the academic projects.

Looking at a deeper breakdown, there are some noticeable differences between the two categories of project. Whereas with the academic projects there seemed no reason to select a higher level of support, the open source projects have a much higher precision at a support of 8, so if precision is deemed more important than recall, $P_{8,1}$ is the best option. If recall is important, then a lower support is required, such as $P_{2,1}$. If coverage has a higher priority, then a selection such as $P_{1,0.3}$ gives better results.

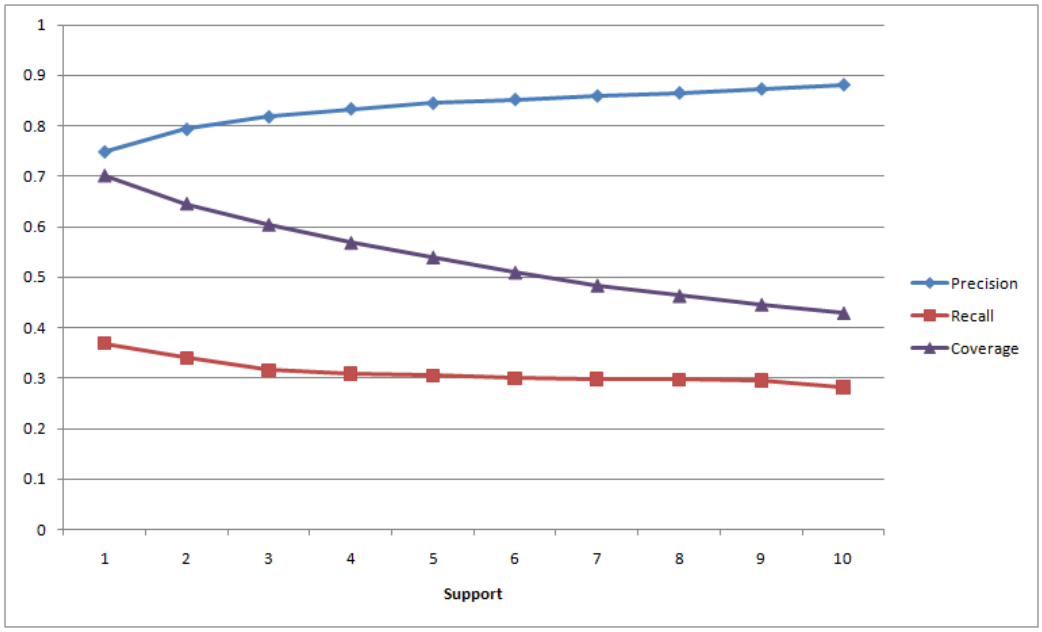


Figure 6.6: The effects on precision, recall and coverage of changing the support parameter on the largest open source projects

While evaluating `Perceive` it has been stated that, typically, the open

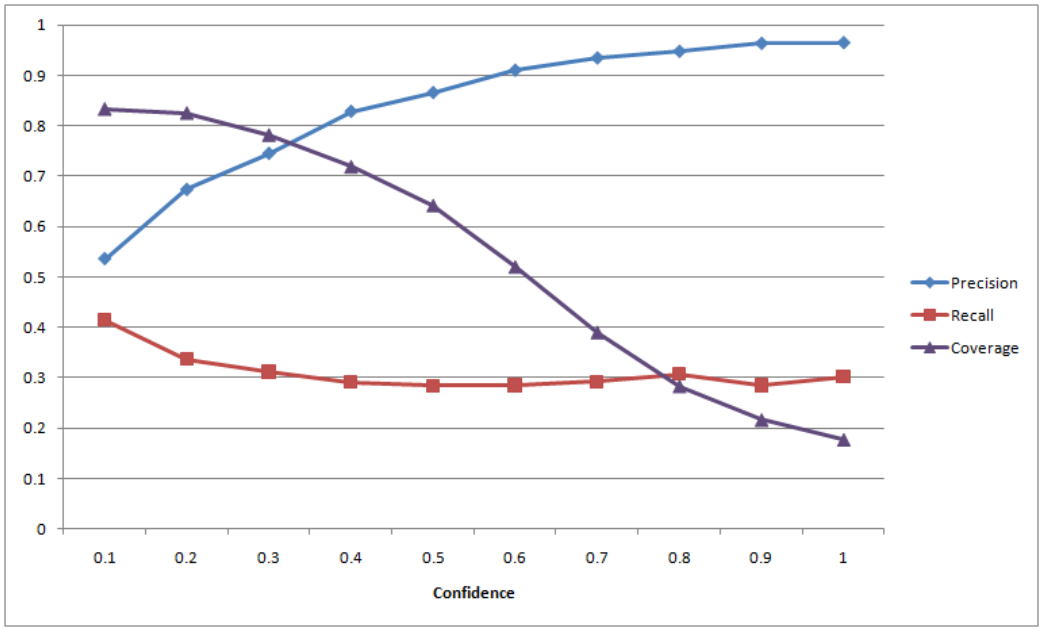


Figure 6.7: The effects on precision, recall and coverage of changing the confidence parameter on the largest open source projects

source projects are larger and longer than the academic projects, which makes it difficult to differentiate between project category and project size/length when investigating factors. However, one of the open source projects is much smaller than the others, comparable in size with the academic projects. Figures 6.6 and 6.7 show the results for the larger open source projects with the smaller one removed. This clearly removes the anomaly in the precision, and demonstrates that precision will clearly continue to rise (although not linearly) until a point is reached where the model can no longer provide sufficient coverage to function.

There appears to be a link between the size of the project and the resulting precision/recall. However, Table 6.7 shows that there is only a weak correlation between project length and the results, or between the number of files in the project and the results. There is a 0.45 correlation between revisions and coverage, suggesting that `Perceive` does indeed perform more reliably as the length of the project increases. However, these results cannot be significant

without more projects to analyse and more information about the projects themselves – a correlation does not imply a causal relationship, and other factors such as development practices may be more important.

In conclusion, we can see that increasing the confidence will always increase the precision while diminishing the recall and coverage. Increasing support will also increase the precision *initially* before tailing off at a point determined by the length of the project. Increasing support also reduces recall and coverage, but not as sharply as increasing the confidence. The size of the project appears to have an impact on the support selected, while the user or task will determine the priority given to precision, recall and coverage when determining the parameters.

6.4.7.2 Phase 2

This section evaluates the effectiveness of `Perceive` in a more real-world scenario, that of predicting a single missing change from a revision. Table 6.8 shows overall figures for each model in this phase. In terms of both precision and recall, overall `Perceive` performs better than the two control models ($\alpha = 0.005$), correctly predicting a mean 17% of the missing files, with a mean false-positive rate of 59%. It is worth noting that in Phase 2, any execution of one of the control models produces a recall equal to its precision – this is due to the fact that one file is expected and one file is suggested – therefore precision and recall will be either 1 or 0. $\texttt{Perceive}_2$ shows a slightly (but significantly) higher overall precision than `Perceive`, but a lower recall – in fact, the recall of $\texttt{Perceive}_2$ is statistically indistinguishable from that of `Frequency`.

As can be seen in Table 6.9 the control models both perform slightly, and significantly, worse on the expanded file set. Neither `Perceive` nor $\texttt{Perceive}_2$ perform significantly differently with the expanded file set. This supports the findings from Phase 1 indicating that `Perceive` is viable to use on more files than source code.

Looking at Table 6.10 it can be seen that overall both `Perceive` and

`Perceive`$_2$ seem to perform marginally better on the open-source projects than on the academic projects, with no difference in recall, while the two control models both performed significantly better on the academic projects than on the open-source projects. However, there are not enough open-source projects to assess the significance of these differences, and it cannot be determined whether or not the category of the project has an effect on the performance. Table 6.12 shows that there is no significant correlation between the number of files or revisions in a project and the performance of `Perceive` or `Perceive`$_2$ in this phase, which matches the results from Phase 1 (see Table 6.7).

Table 6.11 shows that by counting situations where no suggestions were made as a failure (i.e. precision is 0 rather than 1), the performance drops significantly from 75% to 7%, lower than `Random`. However, this is aggregated across the full range of support and confidence, and is not representative of the best performance of `Perceive`.

Overall, these results have confirmed the findings of Phase 1, in that `Perceive` is capable of meaningfully out-performing the control models in an example of a real-world scenario.

6.4.7.2.1 Effects of Parameters on Perceive

This section evaluates the results of `Perceive` in more depth, examining the support and confidence parameters and the effects they have on performance.

Figures 6.8, 6.9, 6.10 and 6.11 show the effects that support and confidence have on precision and recall. In these charts, *Precision 1* is the precision with $FW1$, while *Precision 0* is the precision with $FW0$. As stated above, results in this section will be taken using $FW1$, but the $FW0$ results are included for comparison.

Figure 6.8 shows a similar recall pattern to Phase 1 (see Figure 6.2), slowly falling as support increases. Unlike Phase 1, precision rises, tending towards 1 as support increases. This is due to the different handling of empty suggestions. Figure 6.10 demonstrates that open-source projects have similar

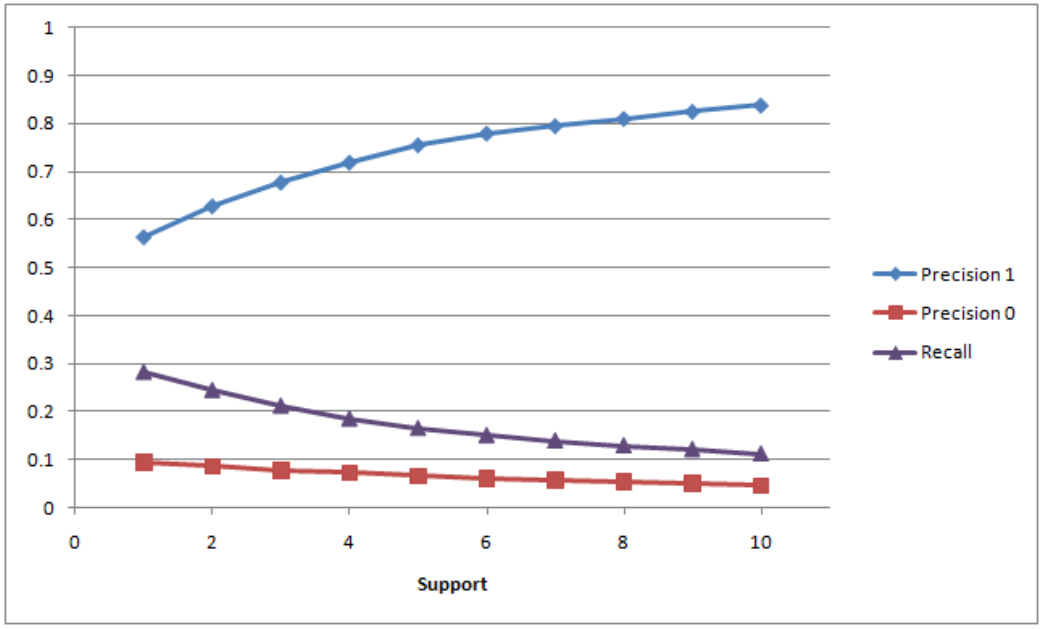


Figure 6.8: The effects on precision and recall of modifying the support parameter on academic projects

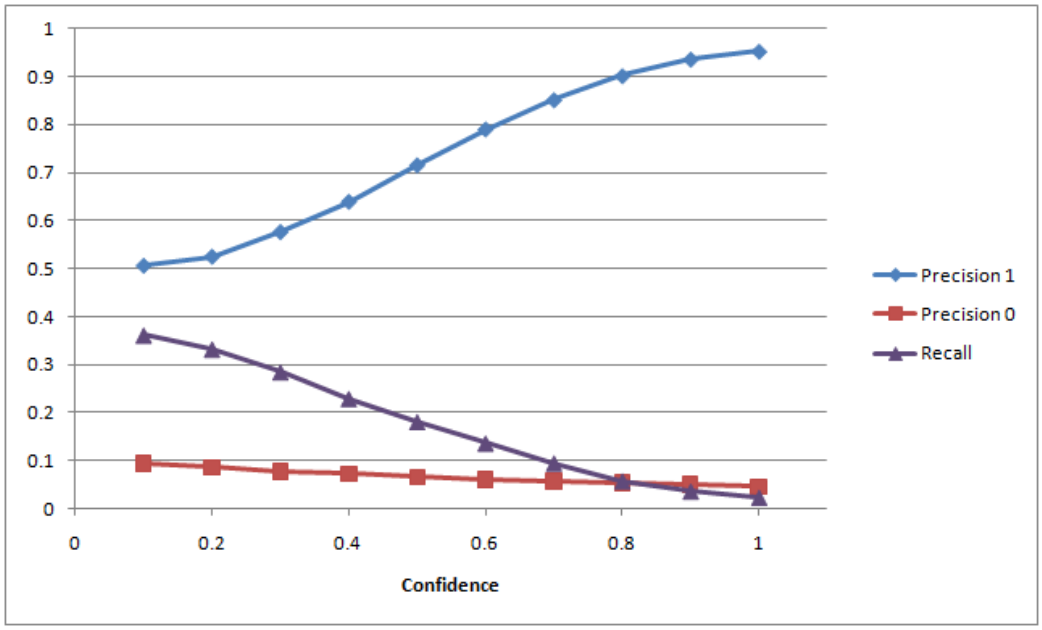


Figure 6.9: The effects on precision and recall of modifying the confidence parameter on academic projects

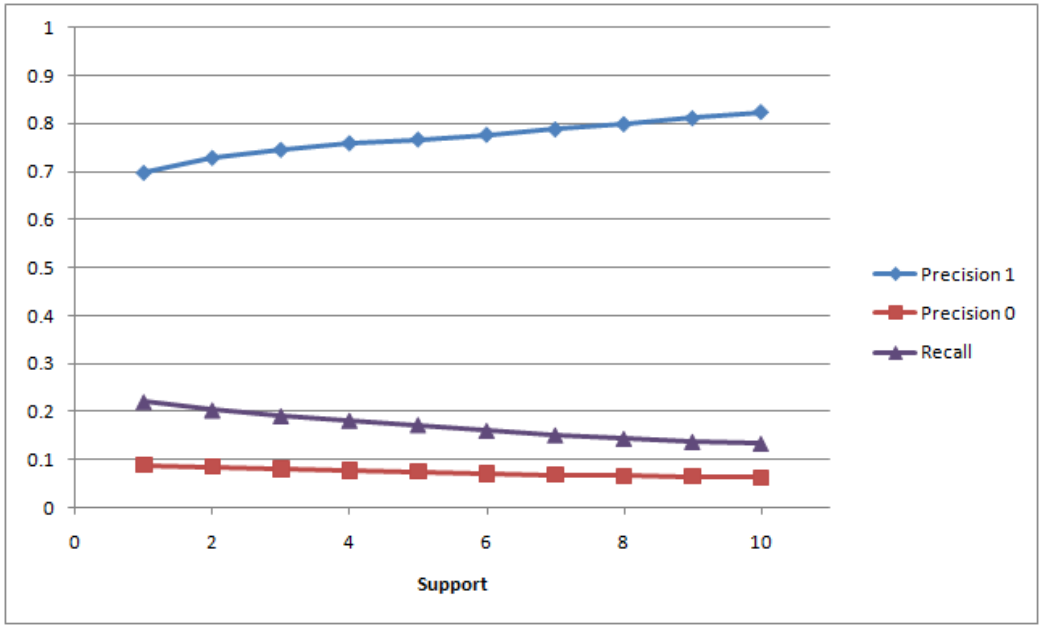


Figure 6.10: The effects on precision and recall of modifying the support parameter on open-source projects

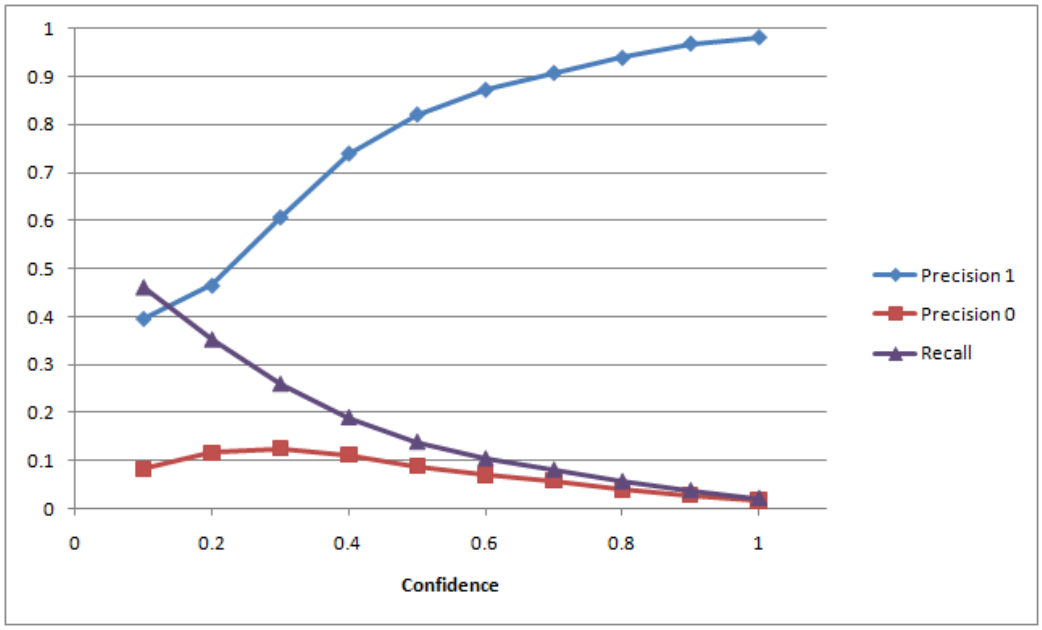


Figure 6.11: The effects on precision and recall of modifying the confidence parameter on open-source projects

trends in this case.

Looking at Figures 6.9 and 6.11, there are clear trends caused by changing the confidence parameter. As with support, higher confidence leads to higher precision and lower recall, but the effect is greater. These results would suggest that confidence is the more important parameter to consider when using a model such as `Perceive`, and the choice will be influenced by the user and the scenario.

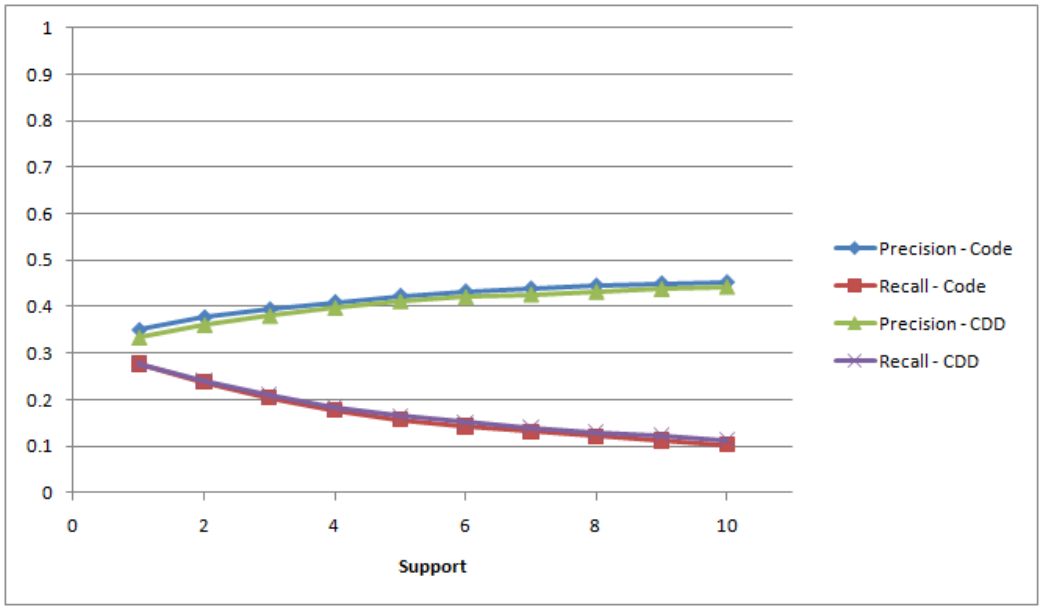


Figure 6.12: A comparison of the effects of support when applied to the code file set and the expanded file set

Figures 6.12 and 6.13 show the similarities of the patterns when the model is applied to different file sets. This confirms the results of Phase 1, demonstrating that `Perceive` is capable of performing as well on a wider range of files as when restricted to source code only.

Figures 6.14 and 6.15 show the effects of every combination of support and confidence, clearly demonstrating the conflict between precision and recall. As with Phase 1, there is no unambiguously optimal set of parameters to produce the best performance. There is a trade-off between accuracy and completeness that must be made, and this compromise must necessarily depend on the user

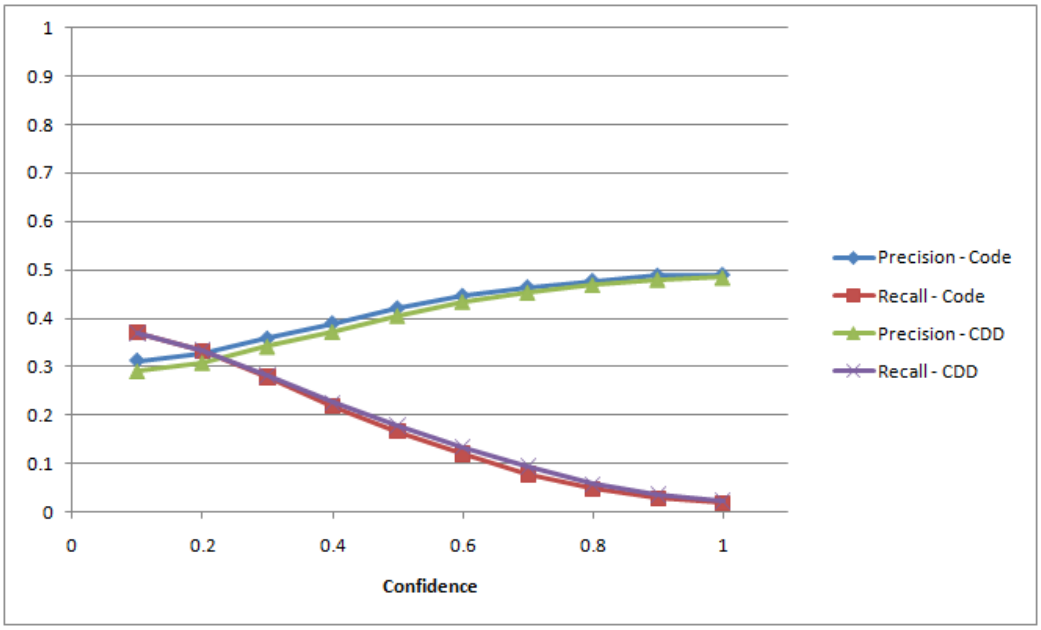


Figure 6.13: A comparison of the effects of confidence when applied to the code file set and the expanded file set

Precision	1	2	3	4	5	6	7	8	9	10
0.1	0.226	0.303	0.383	0.456	0.516	0.560	0.590	0.618	0.649	0.673
0.2	0.270	0.342	0.416	0.477	0.536	0.577	0.606	0.633	0.661	0.685
0.3	0.351	0.416	0.481	0.539	0.594	0.630	0.658	0.682	0.708	0.729
0.4	0.446	0.503	0.564	0.612	0.662	0.694	0.718	0.739	0.760	0.777
0.5	0.540	0.605	0.656	0.698	0.740	0.768	0.784	0.802	0.817	0.831
0.6	0.660	0.704	0.744	0.779	0.812	0.832	0.842	0.856	0.867	0.877
0.7	0.738	0.794	0.821	0.850	0.868	0.882	0.889	0.899	0.907	0.914
0.8	0.803	0.862	0.886	0.904	0.917	0.926	0.930	0.935	0.939	0.946
0.9	0.846	0.909	0.933	0.948	0.952	0.956	0.958	0.961	0.963	0.965
1	0.863	0.926	0.950	0.965	0.970	0.973	0.975	0.976	0.977	0.978

Figure 6.14: A complete overview of the effects of support and confidence on precision when using `Perceive`

Recall	1	2	3	4	5	6	7	8	9	10
0.1	0.595	0.517	0.451	0.397	0.354	0.322	0.295	0.273	0.253	0.234
0.2	0.521	0.461	0.405	0.359	0.322	0.294	0.271	0.252	0.235	0.217
0.3	0.431	0.385	0.340	0.303	0.273	0.250	0.232	0.216	0.204	0.189
0.4	0.344	0.308	0.271	0.240	0.217	0.200	0.187	0.174	0.165	0.153
0.5	0.280	0.244	0.213	0.188	0.170	0.155	0.145	0.136	0.128	0.119
0.6	0.206	0.185	0.162	0.141	0.128	0.118	0.109	0.102	0.097	0.090
0.7	0.149	0.128	0.113	0.099	0.089	0.083	0.077	0.071	0.068	0.064
0.8	0.104	0.084	0.071	0.061	0.053	0.047	0.043	0.040	0.037	0.036
0.9	0.076	0.056	0.043	0.035	0.031	0.028	0.026	0.025	0.023	0.022
1	0.062	0.041	0.029	0.020	0.017	0.015	0.013	0.013	0.012	0.011

Figure 6.15: A complete overview of the effects of support and confidence on recall when using `Perceive`

and the task at hand.

6.4.7.3 `Perceive` and $\texttt{Perceive}_2$

$\texttt{Perceive}_2$ was included in this evaluation to provide a more direct comparison of a history-based model in the context of suggesting a single file. Figures 6.16 and 6.17 compare the performance of the two models. As can be seen, the overall trends are the same, but $\texttt{Perceive}_2$ has better precision and worse recall, although the results converge as the parameters increase. These results are to be expected as `Perceive` makes more suggestions than $\texttt{Perceive}_2$. However, the difference in precision is far smaller than the difference in recall, suggesting that the additional files suggested by `Perceive` are relevant – if the top result is not correct, then the correct result is likely to appear in the set suggested by `Perceive`.

6.4.8 Discussion

This section maps the evaluation of the results for Study 2.1 to the research questions and hypotheses proposed in Section 6.4.1, and then to address the exploratory issues raised.

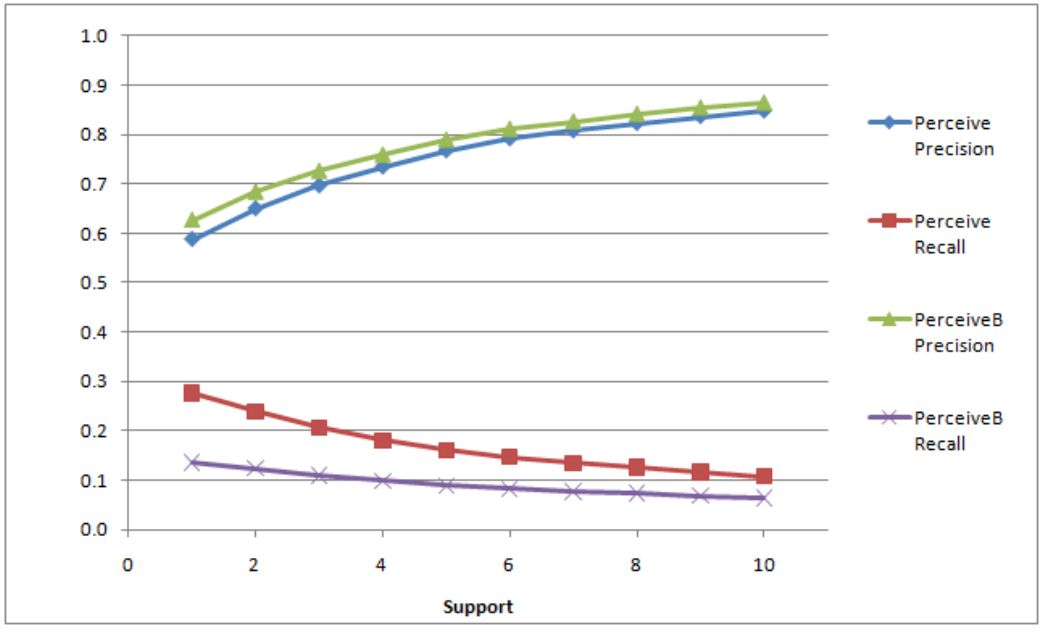


Figure 6.16: The effects of support on the performance of `Perceive` and $\texttt{Perceive}_2$

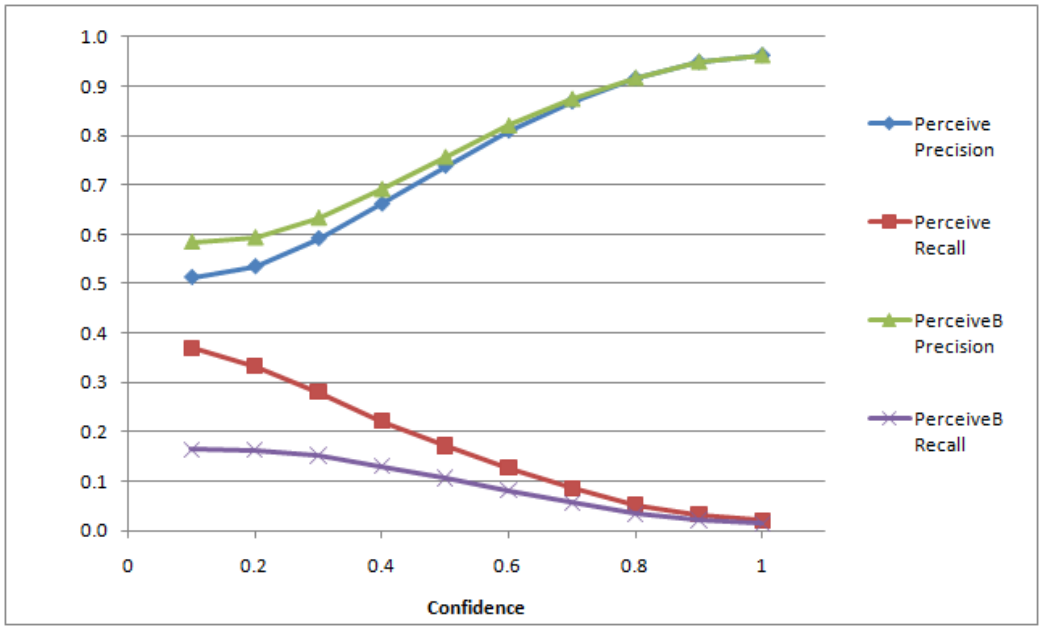


Figure 6.17: The effects of confidence on the performance of `Perceive` and $\texttt{Perceive}_2$

6.4.8.1 RQCP 1-1: How well does Perceive predict software changes?

HCP 1-1: Perceive is a better change-predictor then a random control model.

`Perceive` has been demonstrated to perform significantly better than a number of control models, including two naive, random models and a model based on file activity. This performance was demonstrated in both phases of the study – in Phase 1, the basic predictive ability of the model was assessed, while in Phase 2 a more likely, real-world scenario was assessed. Based on the results shown in Tables 6.2 and 6.8 `Perceive` clearly provides better precision than all control models and provides recall only surpassed by `Random`. The interactions of these results were discussed in Section 6.4.7, and there is sufficient evidence to accept H1.

6.4.8.2 RQCP 1-2: What factors affect the performance of Perceive?

- *HCP 1-2*: Perceive performs better on projects with more revisions.
- *HCP 1-3*: Perceive performs better on projects with more files.

As shown in Tables 6.7 and 6.12 there is no significant correlation between the number of files or the number of revisions in a project and the precision/recall of `Perceive`, although this could be affected by the bias towards smaller, academic projects. More research using a broader range of projects is required to fully investigate this research question, but there is insufficient evidence in this study to accept H2 or H3.

However, as shown in Table 6.7 there is a 0.45 correlation between revisions and coverage, indicating that the longer a project is the more complete the result are. Therefore, while larger projects may not generate *better* results, there is an indication that the results will be more *complete*. Again, more research with a larger pool of projects is required to confirm this new hypothesis.

- *HCP 1-4*: Perceive performs better on open-source projects than academic projects.

In Phase 1, `Perceive` was demonstrated to provide better precision, recall and coverage on open source projects than academic projects. Likewise, in Phase 2, `Perceive` was shown to perform better on open source projects than academic projects, although the difference was less pronounced. As discussed in Section 6.4.8.2 there is no correlation between performance and the number of files or revisions in a project, and therefore it is likely that the open source projects provide better results through some other factor, such as development practices. While H4 cannot be accepted, the results indicate that further research with a larger selection of projects may provide sufficient evidence to support H4.

6.4.8.3 Exploratory Issues

An important aspect of this study was to investigate the effects of the support and confidence parameters on the results. As discussed in Section 6.4.7.1.1 there is no clear combination of parameters that will always – or even usually – produce the best precision and recall. The results show that in most cases, precision can be increased at the expense of recall and coverage, and vice versa. Increasing the support parameter has a slight effect on precision and recall, while the confidence has a much more pronounced effect. However, users of history-based models should be aware that support cannot be increased too far, as precision falls sharply once the support becomes too large for the project to sustain – files or code entities are only co-modified a finite amount of times, and once the requisite support exceeds the number of times files are edited together, relationships can no longer be inferred.

6.4.9 Conclusions

Study 2.1 has reinforced existing research which shows that history-based change prediction is a viable technique for performing change prediction.

Moreover, it has shown that `Perceive`, as implemented here, is a viable change-prediction tool.

This study has also advanced on previous research by investigating the effects of various parameters and factors on the performance of history-based change prediction models. Most importantly, users must choose whether they prioritize precision over recall and coverage. On larger projects, a higher support (>5) can be chosen with little impact on recall and coverage, while confidence must be chosen carefully based on the user's needs, as this will determine whether the analysis favours completeness over accuracy. In general, a higher confidence parameter will be used in situations where false-positives are considered harmful; for example, pre-submission or compilation warnings will tend to occur frequently. On the other hand, a lower confidence will return a larger set of results, which will be more valuable in gaining a broad understanding of a project, typical to a maintainer or a new developer.

A number of unforeseen implementation issues arose during this study (see Section 6.4.4). Most important was the issue of failure and coverage – using precision, recall and coverage it can be difficult to differentiate between a good model and a bad one. Any study investigating history-based models must be sure to take account of these issues and state the approach taken as they can have a severe and misleading effect on the results.

6.5 Study 2.2: Project Applications

6.5.1 Introduction

Study 2.1 investigated the overall performance of `Perceive` across a range of projects. To better understand how `Perceive` applies to individual projects, and to highlight use-cases and obstacles, this study will discuss the application of history-based analysis to a pair of software development projects.

6.5.2 The Projects

Two projects are explored in this study. The first is PuTTY, an open-source SSH client for which the source code and SubVersion repository is publicly available. The second is an academic group-project performed by undergraduate Computer Science students, and which be referred to in this report as the SEG project. For reasons of privacy, no identifying details such as the names of the project, the students involved or any source code are included in this report; where necessary aliases are used for file names.

The choice of project was designed to explore two different areas of software development. PuTTY is worked on by experienced developers; security, reliability and stability are critical to the success of the project. The SEG project is created by a team of students with little to no experience of real-world development, and the project is intended to expose the students to the practices and reality of software development. While PuTTY has been in active development for several years, SEG projects are completed within a small number of months, with the implementation phase of the project occupying only a subset of the total activity.

To compare the projects, a number of metrics[1] are provided in Table 6.13[2].

	Revisions	Files	Changes	Age	Developers
PuTTY	5,123	4,243	14,971	10.5 years	5
SEG	490	2,554	4,786	114 days	8

Table 6.13: Overview of the projects

[1]The tool used to extract these figures ignores some revisions, such as those in which the only changes are to directories. Therefore these figures reflect the data used in this research, not the absolute figures for the projects.

[2]PuTTY is still in development at the time of writing; these figures reflect the state of the project as of July 2009 (Note that this is a more recent snapshot than that used in Study 2.1).

6.5.3 Overview of Perceive

As described in Section 6.4 `Perceive` uses two parameters in its analysis, which define threshold limits: support, which is the number of times in which two files must be modified together to assume a link, and confidence, which is the ratio of co-modifications to modifications. For example, assume file A (modified 5 times) and B (modified 8 times) are modified in the same revision 4 times. If the support parameter is 4 and the confidence parameter is 0.8, then `Perceive` will infer a link from file A to file B, but not the other way around – the support in both cases is 4, which is sufficient, while the confidence of A to B is 0.8 ($\frac{4}{5}$) but the confidence of B to A is only 0.5 ($\frac{4}{8}$).

By changing the threshold parameters users can shape the results they receive – a higher required confidence and support will result in a smaller set of code entities with fewer false-positives but potentially more false-negatives, while a lower confidence and support will result in more false-positives but fewer false-negatives. One aspect of this research is to investigate the effects of these parameters on real projects rather than a high-level aggregate.

6.5.4 Case Study 2.2A: SEG

This first case study explores the use of `Perceive` in a small academic group project, initially demonstrating the workings of `Perceive` and then highlighting the uses of the technique in an academic setting, including various use-cases such as development, maintenance, management and assessment.

In Study 2.1 (Section 6.4) `Perceive` was first used to predict complete revisions given a single changed file, and secondly to predict a single missing file from an otherwise complete revision. In Phase 1 of that study each project was processed twice, based on the choice of files – source code files, or source code, documents and data files. In Phase 2 each project used the two different file selections, and also altered what was named the "fail weight" – whether an empty result set had a precision of 1 or 0. In this research the larger file set is used, and the fail weight is 1.

Table 6.14 shows the results of the academic projects in general, and the SEG project in particular, using these parameters.

	Phase 1		Phase 2	
Project	Precision	Recall	Precision	Recall
All Projects	0.669	0.218	0.742	0.173
Academic Projects	0.657	0.214	0.740	0.174
SEG Project	0.873	0.474	0.526	0.312

Table 6.14: Performance of `Perceive` for various project groups

As an example of `Perceive` in use, suggestions are generated for the most active file (`FileA`) in the project, with a configuration of $P_{1,0.2}$. 10 other files are suggested, all of which are source code files in the same directory, all part of the GUI code. Secondly, suggestions are generated for the same file but with a configuration of $P_{10,0.5}$. This time, only two files (`FileB` and `FileC`) are suggested. Changing the configuration to $P_{10,0.6}$ results in no suggestions – no files are co-modified in more than 60% of the file's revisions. Conversely, suggestions for `FileB` and `FileC` at this higher threshold both suggest `FileA` – while `FileA` is often modified without the other two, they are less frequently modified without `FileA` also being modified. Importantly, all three files are most frequently modified by the same user, adding further support to the relationship inferred by `Perceive`. From this, we can identify with some confidence that if `FileA` is modified, `FileB` and `FileC` are also likely to be modified; likewise, if `FileB` or `FileC` are modified, then `FileA` is highly likely to be modified. Therefore, if modifications are made to some of these files, but not others, then these files should also be examined in case they require modification as well.

This behaviour highlights a problem with history-based analysis. In the event a revision is committed in which `FileA` and `FileB` were modified, but not `FileC`, if the developers then realize that `FileC` requires modification as well, then it is likely to be committed as a new revision, which will further dilute the relationships as inferred by `Perceive`. A mechanism for addressing

this problem could be to, during analysis, "fold" revisions together which are by the same user and are committed within a specific time frame, an extension of the sliding window approach used in CVS preprocessing (Zimmermann and Weissgerber, 2004). This way, in the example given, if the mistake was corrected by the original user, and within a set time, `Perceive` would still reinforce the link between the files, despite the changes being in different revisions.

This example, of three interrelated files, also highlights a potential area for development in `Perceive`. The current model uses first-order analysis when creating links between code entities, in that links are only generated between pairs of files. If two files have links to the same third file, then a simple combination is used (total support, mean confidence). More complex, higher-order analysis could be used to more strongly infer links between clusters of files, which could improve the performance of `Perceive` when using a complete revision to suggest missing files.

Examining a documentation file (`FileD`), `Perceive` generates strong links between that file and related documentation, as well as a number of source code files. The documentation and source code is all functionally related and are part of a reporting GUI. Again, these related files are also most frequently modified by the same user. Following the links to other files, the documentation files are all strongly interrelated, as expected since they will be generated simultaneously and committed together. However, while the documentation files link strongly to their related code files, the source code files do not link as strongly back to the documentation. This suggests that the documentation typically accompanies a change to the code, but changes to the code are not always accompanied by updates to the documentation. Revision folding might be a solution here, but examination of the revision data shows that it is not a case of the documentation and the code being committed separately, but rather the documentation simply being updated less frequently than the source code.

As demonstrated in Study 2.1, history-based analysis can be successfully

used to predict missing changes from a revision. The research was conducted by omitting a single change from a revision and seeking to discover it from the remaining changes. In a real-world scenario, users would use the system to determine if any changes should be made before a revision is committed. This case study attempted to conduct a more focussed, real-world based examination by mining the revision logs for "forgotten" files, by searching for revisions from the same user made in a short time frame, with a comment indicating that a file was missed. However, in this project very few such occasions existed. One explanation for this was that the SEG project had a relatively high change-per-revision ratio (nine changes per revision, as opposed to two for PuTTY, four for both Parrot and GNUstep, two other large open source projects). This indicates that the students would typically do a larger amount of work on a range of files and submit at wider intervals than is typical on real-world projects.

A maintainer joining an existing project with little or no experience of it, and no knowledge of its structure will often use tools to help create a mental model of the project, including how files are connected. By using history-based techniques, a maintainer can use the behaviour patterns of the project developers to form an understanding of the code. `Perceive` can generate a series of visualizations which present the project structure and show the links inferred between files. Such an overview can be of great help to a newcomer to the project. In the academic context, the tasks performed, and the understanding required by an assessor might be comparable to those of a manager or maintainer of a real-world project. Experimental data has shown that when asked to assess a group project, assessors tend to short-list two or three "interesting" files and use them as a starting point to explore the project (see Section 7.3). This branching-out process tends to follow syntactic links – e.g. one class creates objects of another class, so the assessor will look at that class too, and so a model of the project will be formed. However, the files suggested by `Perceive` provide a secondary means of selecting further files for examination, by presenting files that are similar in development terms

rather than purely functional terms.

Another useful application of `Perceive` in the academic context is, as in the case of the SEG project, when the groups consist of students from more than one institution whose work is to be assessed by campus. Looking at a file worked on by students from one campus, it is not immediately obvious which related files were worked on by the same campus, and which ones simply use files created by students from the other campus. `Perceive` can go beyond the syntactic links to show which files are worked on by which campus. For example, one area of the project, a data layer, was worked on by both campuses; one campus generated very strong links between files within the data layer, while the other campus – while working on it more – created fewer links between different folders, and more links to the associated documentation. From this it can be determined that one campus tended to do broader changes, while the other did more contained changes, and also maintained the documentation better. Such information would not have been available from a simple code reading.

In Study 2.1 the research was conducted with the assumption that in general the quality of results generated by history-based analysis techniques are comparable to those generated by static, syntax-based techniques; this assumption was based on previous research in the field (Zimmermann et al., 2005; Hassan and Holt, 2004). These case studies afford the chance to investigate this assumption in more detail by comparing clusters of files with their syntactic and functional connections.

Taking an overview of the project, the main clusters are the JavaDoc files, which are generated and committed together, and so build strong links between them. A second observation is that the strongest clusters of code files consist of files in the same directory. For example, a reporting component, consisting of three directories (corresponding to three layers – data, logic and presentation), exists in the main trunk and in a branch; the branch version has links between files across all three directories, while the trunk version has links only within the logic directory. This suggests that the branch version

was created and worked on as a whole, while work on the trunk version has been contained to individual directories. Both versions were documented, but only the trunk code has formed links with its documentation.

Examining the source code of the reporting component in depth, starting at the main GUI code – the file with the most activity – the benefits of a history-based approach become apparent. The central GUI depends on two components (other than built-in APIs), the core files of the project and another GUI element of the reporting component. Examining a call-graph or a dependency graph would suggest that the extensive list of core files the GUI depends on would link the file very strongly with the core files. However, the file is *never* co-modified with the core files, despite both being actively developed. This suggests that, despite the strong syntactic links, an alteration to the GUI or a core file does not require a modification to the other. Conversely, the other files in the reporting component are very strongly linked to the GUI – a file in the data layer is linked to the GUI with a support of 11 and a confidence of 91%, despite only being connected by a chain of intermediate files.

6.5.4.1 Discussion

Study 2.2A has demonstrated how `Perceive` works, and explored its potential applications for students and assessors in an academic context. As demonstrated in Study 2.1 the change prediction feature is not too effective in these academic projects, as there are too few revisions and the development style is too unstructured to support strong history-based change prediction. However, other educational applications, such as assessment, are well supported by `Perceive`. Finally, instances of `Perceive` finding relevant links that might be missed by static, syntax-based techniques, are identified, as well as highlighting situations where a static technique might incorrectly suggest a connection.

These results suggest that although `Perceive` will not perform perfectly in all cases, nor will traditional techniques, and there is a strong case for using

history-based analysis to support and augment syntax-based techniques.

6.5.5 Case Study 2.2B: PuTTY

This second case study examines the application of `Perceive` to a mature, stable open source project which sees significant real-world use. PuTTY "is a free implementation of Telnet and SSH for Win32 and Unix platforms, along with an xterm terminal emulator", and has a publicly accessible SubVersion repository with revisions spanning more than 10 years.

As with case study 2.2A, `Perceive` was used with the expanded file set and a fail-weight of 1. Table 6.15 shows the performance of `Perceive` compared to other groups of projects using these settings.

	Phase 1		Phase 2	
Project	Precision	Recall	Precision	Recall
All Projects	0.669	0.218	0.742	0.173
Open Source Projects	0.804	0.265	0.770	0.170
PuTTY	0.837	0.281	0.752	0.161

Table 6.15: Performance of `Perceive` for various project groups

As previously stated, PuTTY has a much smaller change-per-revision ratio than the SEG project (2 as opposed to 9), and so it is expected that `Perceive` will infer a larger number of stronger links between smaller clusters of files compared with the SEG project. Examining the development patterns of PuTTY, it can be seen that with each release a new copy of the code is created and tagged with the release, while the original copy of the code continues development. In many projects this practice is accompanied by continuing development on the older releases – often for addressing security issues – but this is not the case with PuTTY. Such use would highlight a problem with `Perceive` in that when copies are created, the files' histories are not copied with them, and the history-based analysis must begin creating links from scratch. Future development of `Perceive` could remedy this by

copying the history of a group of files as they are copied. However, as this practice is not used in PuTTY's development, this drawback does not cause any issues in this case study.

Investigating the most active code within PuTTY reveals a cluster of weakly linked files – `ssh.c`, `putty.h`, `window.c`, `terminal.c`, `windlg.c` and `settings.c`, which are frequently co-modified. However, while there is high support for this cluster the confidence is typically between 20% and 50%. Reviewing revision history, this is because while the files are frequently co-modified, they are also frequently modified alone, with no relation to other files. Taking an overall look at the inter-file links across PuTTY, it can be seen that very few strong links develop between files because of this, making history-based change-prediction or impact analysis difficult, especially for the most active files. However, many of the less active files, such as `sshblowf.c` or `sshdes.c`, have much stronger links to other files, including the most active cluster – for example, `sshdes.c` links to `ssh.c` with a support of 22 and a confidence of 52%, while `sshblowf.c` links to `ssh.c` with a support of 13 and a confidence of 59%. These two files also link to each other quite strongly.

6.5.5.1 Discussion

The fact that PuTTY tends to have very small revisions, with files frequently modified alone – often multiple times in succession – means that history-based analysis does not *appear* to function as well as on some other projects. However, the inability to build links between files is caused by the fact that the developers do not always modify files in groups; given that PuTTY is a highly successful, stable and secure project, it can be safely assumed that the development habits and practices are equally successful, and that the developers have a deep understanding of their code and how to modify it. In that context, not making any suggestions for further modifications is in fact the correct outcome in many cases. A code-based technique would always find syntactic and structural links between files, regardless of developer behaviour and knowledge. In this way, the expertise and habits of the developers guide

history-based techniques to the conclusion that, given a modification, no other files necessarily require updating. In fact, reviewing the revisions, a common follow-up to a changed file is to make a further change to that file (the words "oops", "forgot" and "missed" occur frequently throughout the revision logs).

It is important to bear in mind, however, that this result cannot be generalized to other projects without first understanding the nature of the project. To simply assume that an empty suggestion means nothing should be changed would be a mistake. Therefore, when using any form of impact analysis or change prediction – history-based or syntax based – requires an understanding of the nature of the project to determine what parameters to use and what answers would be accurate. Fortunately, tools exist which can facilitate this comprehension (`Perceive` itself was used to review the revision logs and change histories) and can guide users towards a model of the project sufficient to determine how best to use change prediction techniques, and how to interpret the outcomes.

6.6 Study 2.3: Improving Perceive

Study 2.1 (see Section 6.4) evaluated `Perceive` and concluded that it is a viable tool for conducting change prediction. Previous research has found that adding additional data sets to a change prediction model can show significant improvements in performance. For example, one study has shown that factoring in the author and time period of a change can improve performance (Kagdi et al., 2007b), while a study by Zhou et al. achieves increased performance by incorporating source code dependency levels, co-change frequencies, change significance, age of change and author (Zhou et al., 2008).

Study 2.3 seeks to improve the performance of `Perceive` by incorporating data from thematic analysis into the algorithm. By classifying revisions into one of a set of maintenance activities, it is possible for the change prediction algorithm to alter the weightings given to files when calculating how related pairs of files are.

6.6.1 Research Questions

This study seeks to answer the following research question:

- *RQCP 3-1*: Can thematic analysis data be used to improve the performance of `Perceive`?
  - *HCP 3-1*: Inclusion of maintenance activity data improves the performance of `Perceive`.

6.6.2 Perceive

Two new change prediction models were implemented: `Activity` and $\texttt{Activity}_2$. `Activity` works by restricting the search to revisions which involve the same activity type as the revision under study. For example, a project with 200 revisions might have 40 corrective revisions; when trying to find a missing file from a corrective revision, only previous changes from the other corrective revisions will be counted. This model is predicted to provide higher precision at the expense of recall, by suggesting fewer files with more accuracy. The second model, $\texttt{Activity}_2$, functions identically to `Perceive` but gives additional weight to files changed in revisions of the same type. This model is predicted to increase recall at the expense of precision by suggesting a larger number of files.

6.6.3 Study Design

Two sets of projects were used in this study: 22 student projects and PuTTY. These are the same projects which underwent thematic analysis in Chapter 5, and as such their revisions have already been completely classified by maintenance activity type.

While Study 2.1 used many combinations of variables, this study uses the expanded file set (code, documents and data), a fail-weight of 1 (i.e. an empty result set gives a precision of 1) and uses the process from Phase 2 of

the study (attempting to identify a single missing change from a revision, a typical use-case for this system).

6.6.4 Results

Table 6.16 shows the results of three models (`Perceive`, `Activity` and $\texttt{Activity}_2$) on two sets of projects - SEG and PuTTY.

	PuTTY			SEG		
Model	Precision	Recall	F-Score	Precision	Recall	F-Score
`Perceive`	75.7%	15.9%	26.3%	74.3%	16.9%	27.6%
`Activity`	70.4%	17.0%	27.4%	81.8%	11.3%	19.9%
$\texttt{Activity}_2$	74.4%	17.0%	27.6%	73.2%	17.4%	28.1%

Table 6.16: Overview of the performance of the new change-prediction models

The results for `Perceive` are taken directly from the relevant projects in Study 2.1. It should be noted that, as with studies 2.1 and 2.2, these results are the mean values for the full range of support and confidence parameters; the peak values are somewhat higher, and Table 6.17 shows the peak F-Scores for each project set and model.

Model	PuTTY	SEG
`Perceive`	36.8%	59.1%
`Activity`	39.0%	51.2%
$\texttt{Activity}_2$	36.9%	59.1%

Table 6.17: Peak F-Scores of each model for the two project sets

6.6.5 Evaluation

As predicted, `Activity` displays increased precision and reduced recall for the SEG projects; conversely, it has increased recall and reduced precision

for PuTTY. Investigation reveals this to be a result of the way multiple suggestions for the same file are aggregated to produce a final metric for that file. `Activity` was expected to show higher precision by making fewer guesses; however when a file is suggested more than once for a revision (e.g. two files within that revision have links to it) it is possible for subsequent suggestions with lower confidence to reduce the overall confidence in that suggestion until it no longer reaches the threshold, which is what happened when `Activity` was applied to PuTTY.

$\texttt{Activity}_2$, however, performs as predicted, showing both increased recall and reduced precision in both cases. The increase in recall is sufficient to offset the reduction in precision, achieving higher F-Scores in both cases; a Friedman Test (Friedman, 1940) shows that this improvement is significant (p=0.05).

6.6.6 Discussion

As described in Section 6.6.1, the research question addressed by this study is as follows:

- *RQCP 3-1*: Can thematic analysis data be used to improve the performance of `Perceive`?
  - *HCP 3-1*: Inclusion of maintenance activity data improves the performance of `Perceive`.

The results of the study have shown that the incorporation of thematic analysis data into a change prediction algorithm can in fact improve the performance to a statistically significant degree. Giving more weight to files which were changed in revisions of the same classification as the revision being analysed offers a small reduction in accuracy and an increase in recall, which overall results in a significantly increased F-Score. This allows us to accept hypothesis *HCP 3-1*.

6.6.7 Conclusions

The improvement to history-based change prediction can be added to those developed in other research, and contributes to the goal of making change prediction as accurate and complete as possible. Every technique which can produce an improvement in the quality of results, whether as part of a static or history-based analysis, will serve to support developers and maintainers as their code and projects evolve.

Source code dependency levels, co-change frequencies, change significance, age of change, author and time periods have all been demonstrated in other research to improve history-based change prediction; now maintenance activity classification can be added to this set.

6.7 Case Study Discussion

The goals of this case study were to measure the performance of history-based change prediction, to explore the factors which affect its success, and to seek to improve the model using data generated by the thematic analyses in Chapter 5. These goals are addressed using the following research questions and hypotheses:

- *RQ 4*: Is history-based change prediction a viable technique?
- *RQ 5*: When does the technique outperform syntax-based methods?
- *RQ 6*: Can project profiling be used to improve history-based change prediction?

6.7.1 *Research Question 4*

- *RQ 4*: Is history-based change prediction a viable technique? ***Proven: Yes***

The viability of history-based change prediction as a change prediction technique has been explored in other research; this case study contributes

to this existing body of knowledge by adding data from more projects and techniques.

Study 2.1 (see Section 6.4) benchmarked the performance of `Perceive` along with a series of control models. It concluded that the technique implemented here was significantly better than the control models, showing that history-based change prediction is possible even in small, poorly-organized student projects.

In Section 6.4 Study 2.2 explored in detail the application of `Perceive` to a pair of projects, showing that the technique has further applications in the assessment and management of student group projects.

6.7.2 *Research Question 5*

- *RQ 5*: When does the technique outperform syntax-based methods?

As described in Section 6.5, Study 2.2 highlights certain cases where a history-based change prediction model can successfully infer a relationship between a pair of files which cannot be found by a static analysis technique, especially between files of different types such as documentation or graphics. Conversely, there are also cases where a history-based technique cannot perform as well as a static technique; for example, there will always be a "training period" for a new file, where insufficient data exist to make predictions based on, or for, that file. This leads to the conclusion that an ideal change prediction model will incorporate elements of both static and history-based techniques.

6.7.3 *Research Question 6*

- *RQ 6*: Can project profiling be used to improve history-based change prediction? ***Proven: Yes***

Study 2.3 (see Section 6.6) showed that it is possible to improve the performance of a history-based change prediction model to a statistically significant

degree using data generated by a thematic analysis: adding maintenance activity classifications to the model increased the mean F-Score for both PuTTY and the SEG projects.

Unlike other research, which has used source code analysis to augment history-based change prediction, this study has improved the change prediction model while still only using language-agnostic transactional data from RCS repositories.

6.8 Case Study Conclusions

This research has the overall goal of identifying and evaluating techniques which use analysis of transactional repository data to support project comprehension, using the following research question:

- RQ 1: How can data mining of revision control systems be applied to support project comprehension?

The case study reported in this chapter has proved that history-based change prediction is a viable technique, and one that can be improved using thematic analysis of project data. Change prediction is a process which requires project comprehension; an algorithmic technique to improve change prediction will therefore have the effect of supporting project comprehension. Therefore, this case study provides the following answer to the above research question:

- Change prediction using repository analysis is a technique which supports project comprehension.

6.9 Future Work

There are several avenues which should be pursued following this research. Firstly, user-based studies are necessary to be able to investigate the trade-offs which should be made in terms of support and confidence – is it more

important to have high recall and give complete results, or is high precision, with fewer false-positives, preferable? Are there situations or use-cases where this will change?

Secondly, more project studies are required to be able to better address RQCP2. A wider range of open-source projects, academic projects and even commercial projects would enable deeper analysis with higher confidence.

`Perceive` uses a simple weighting system to infer relationships between files, increasing the weight between two files by 1 whenever they are co-modified. There are, however, a number of other factors which could potentially lead to better prediction. Such factors include the creator of the files, the current author, files which are co-created, the file hierarchy, and chronological-based weighting where earlier activities are given less importance than more recent ones. Research will be conducted to determine which, if any, of these factors lead to better change prediction.

Comparisons with syntax-based change prediction tools currently used in development environments would allow for results which would be of greater value to industry practitioners. As the goal of history-based change prediction is not to replace current methods but to support them, these comparisons will allow the benefits of augmenting existing methods to be determined.

Finally, although the thematic analyses described in Chapter 5 required a manual classification process, a number of the revisions could be automatically classified based on keywords from the revision comments (e.g. 'Fixed a bug'). Further research is planned to determine if an automated classifier could be employed; this would have to be assessed against the thematic analyses already conducted, and then against the change prediction baseline established in this case study. An automated classification system would not be able to classify every revision, but as demonstrated by the fact that the SEG projects were incompletely commented – and therefore incompletely classified – an automated classifier would not necessarily have to classify every single revision to be effective.

6.10 Summary

This chapter detailed the benchmarking and application of a history-based change prediction model; the results add evidence to existing research that the method can viably support tradition techniques and demonstrate that care must be taken when defining the parameters and threshold values of the algorithms, as the best results depend on the nature of the project and the task being performed. Lastly, an improved model was developed using the thematic analysis research conducted in Chapter 5; evaluation showed the improvement to be statistically significant.

Chapter 7 describes the final two case studies, in which Perceive is used to support project management and assessment. A series of features such as visualizations and project reports are implemented, guided by feedback from student managers. An experiment is also performed to determine the use of Perceive in automating a time-consuming and difficult aspect of the assessment process: the selection of specific components of a project to be examined in more detail, being representative of the whole.

Chapter 7

Case Studies: Project Management and Assessment

7.1 Introduction

The previous two case studies have investigated two areas of project comprehension supported by data mining of revision control repositories in both educational and industrial domains. This chapter explores the use of Perceive in a purely educational context, in the management and assessment of group projects.

Project comprehension is an important element of both management and assessment activities, and Chapter 2 describes a number of tools and techniques for supporting these tasks. This chapter describes a pair of case studies which aim to explore how transactional data from RCS repositories can be used to support project comprehension in management and assessment scenarios. Section 7.2 reports a case study in which `Perceive` is developed as a project management tool and provided to student managers to support their roles, while Section 7.3 describes a second case study investigating the use of `Perceive` to support project assessment.

7.2 Case Study: Software Visualization and Project Management

The data stored in RCS repositories is varied, complex and extensive, and in order to use that data to support project comprehension it must be extracted, analysed and presented in some way. The case studies in Chapters 5 and 6 have demonstrated some methods by which the data can be made to support project comprehension; this case study explores how software visualization can be used to present the data to users in order to further support project comprehension.

7.2.1 Case Study Design

This section details the goals, techniques and evaluation of the case study using the DECIDE framework. The subject of the case study is a sub-study in which `Perceive` is developed into a software visualization suite for use by student project managers to support them in their roles, and draws conclusions from the outcomes of this sub-study.

The structure of this case study and its research questions are outlined in Figure 7.1.

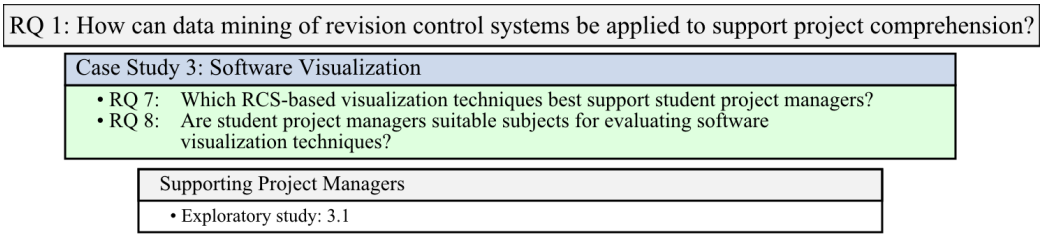


Figure 7.1: Structure of the research questions and studies of the software visualization case study

7.2.1.1 Research Goals

This study seeks to investigate what forms of software visualization are useful, accessible and appropriate for use by student project managers; although there exist tools and techniques for visualizing RCS repository data (see Chapter 3) there remains scope for further research and development. The findings of this case study will contribute to the field of repository visualization by establishing the success or failure of the techniques developed and deployed in this chapter.

Furthermore, as described in Chapter 2, there is little empirical work conducted on the success and adoption of software visualization tools and techniques outside of an academic setting. Research has shown that in some fields, students are viable substitutes for industrial practitioners when conducting empirical studies (Host et al., 2000); this case study also aims to explore whether or not the results can be mapped to industrial practitioners. If it is demonstrated that students are good substitutes for industrial practitioners, it will allow more studies to be conducted to assess the usefulness and adoption of software visualization tools and techniques.

7.2.1.2 Research Questions

This case study is conducted to provide evidence to the overarching research question of "*How can data mining of revision control systems be applied to support project comprehension?*"; software visualization and project comprehension are closely connected, and therefore this case study will evaluate the effectiveness of visualization techniques based on transactional RCS data in supporting project comprehension. Therefore this case study seeks to answer the following research questions to help address the primary goal of this research:

- *RQ 7*: Which RCS-based visualization techniques best support student project managers?

- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques?

If *RQ 7* identifies any visualization techniques which successfully use repository data to support project management, it will provide additional evidence that data mining of RCS repositories can support project comprehension. The outcome of *RQ 8* will have a direct impact on the validity of the outcome of *RQ 7*; if students are not suitable subjects for evaluating visualization techniques, then the evidence supporting *RQ 7* will not be as strong, whereas if students are indeed suitable subjects, then it will lend weight to the outcome of *RQ 7*.

7.2.1.3 Evaluation Paradigm and Techniques

The study described here is evaluated using feedback from project managers in unstructured interviews; initially this feedback is used to guide an iterative development process to better tailor `Perceive` to their needs, and then finally to gauge the success, in terms of usefulness and adoption, of the visualization tools they were provided with.

This final feedback will form the basis for a discussion to evaluate what forms of visualization techniques are most useful for student projects managers; their experiences, behaviours and use-cases will also be compared to those of industrial practitioners to assess the degree to which student project managers can be used as substitutes for industrial project managers in software visualization research.

By determining which, if any, software visualization tools and techniques using transactional RCS data can be used to support project comprehension, this case study will provide further evidence towards the evaluation of the overall research goal of this research.

7.2.1.4 Practical Issues

This study involves the participation of students who took on project management roles as part of a Software Engineering module. Therefore, while it

would have been preferable to use one cohort of students to prototype and trial both the study and the tools, only one year was available for conduct of the study, and so the development and evaluation was performed within the same cohort of students. Fortunately, the managers were sufficiently engaged in the study to enable it to be conducted within a single cohort.

7.2.1.5 Ethical Issues

Ethical clearance was sought and granted to access the repository data; all student managers signed ethical clearance forms to take part in the study and to allow use of the findings. During their training process, a project from a previous year was used as an example; this project was anonymised to protect the identities of the students involved. Finally, the managers were provided with login credentials to ensure that they had access only to the data for their respective projects, and no others.

7.2.1.6 Evaluation and Discussion of Results

After the sub-study is reported, the findings will be discussed in the context of the research questions described above, relating the study's outcomes to the overarching research goals.

7.2.2 Study 3.1: Software Visualization and Project Management Using Perceive

7.2.2.1 Research Aims and Method

Project management is an activity which cannot effectively be carried out without a suitable degree of project comprehension – the more deeply and completely a manager understands a project, the more effective the management and decision-making process will be. This study reports the development of `Perceive` as a project management and software visualization tool, and the experiences of a group of student project managers as they employ `Perceive` to support their roles.

The project managers were a group of 20 third year software engineering students who took on their roles as part of a Project Management module, and were assigned groups of second year students taking part in the group projects, with two managers per group. The managers were given training in the use of `Perceive` and were given access to the live data for their groups for the duration of the projects. After the projects were completed, the managers were interviewed for feedback on the features.

The purpose of this study was to assess how well managers would adopt and use tools designed to support them in their roles. The features and visualizations provided by the tool were designed to support common tasks in software development, and are based on research described in Section 2.6.

7.2.2.2 Features and Visualizations

Because of a careful design and class structure the system developed in Chapter 4 was easily modified to become an end-user facing application to support project comprehension. Initial features included:

- *Project Overview*: A page showing overall metrics about the project, including duration, number of revisions, students, files and changes, and a breakdown of the types of changes (Figure 7.2).
- *Revision Information*: A list of revisions could be browsed, searched and sorted (Figure 7.3).
- *File Activity*: A list of the most active files. For example, Figure 7.4 shows the most modified files from PuTTY, with the files colour coded by type.
- *Student Information*: Details about the information of each student who has contributed to the project, such as the types of changes and commonly modified files. Figure 7.5 shows an overview of one of the PuTTY developers, giving a breakdown of the types of files he worked

on, the types of actions he performed (updating, adding, deleting, etc...), the files he edited the most and a short overview of his recent changes.

- *File Information*: Details about the selected file, including which revisions it has been changed in and which students modified it. Figure 7.6 shows the information for a single file, including the types of actions performed on it, the developers who use it the most and a list of related files (using the algorithm described in Chapter 6).

These features were implemented to provide "at a glance" information about a project with the aim of letting a manager quickly understand the structure of the software and the dynamics of the group.

	Overall
Start Date	22/11/2007
Last Date	14/03/2008
Project Length	114 days
Students	9
Files	2,554
Revisions	490
Messages	401 (81%)
Changes	4,786
Additions	1,436 (30%)
Deletions	275 (5%)
Updates	1,792 (37%)
Changes Per Revision	
Mean	9
Range	1 - 296
Diff Size	395,448 lines

Figure 7.2: An overview of a student project; this is the first information shown to a user when Perceive is loaded. Note that the "messages" field fades to red as the ratio of messages to revisions drops – this was introduced when it became clear that students were not keeping good logs and a visual warning could help to mitigate this.

It was important that the interface be as interactive as possible, and so it is possible for the user to select a range of users, or revisions and view the

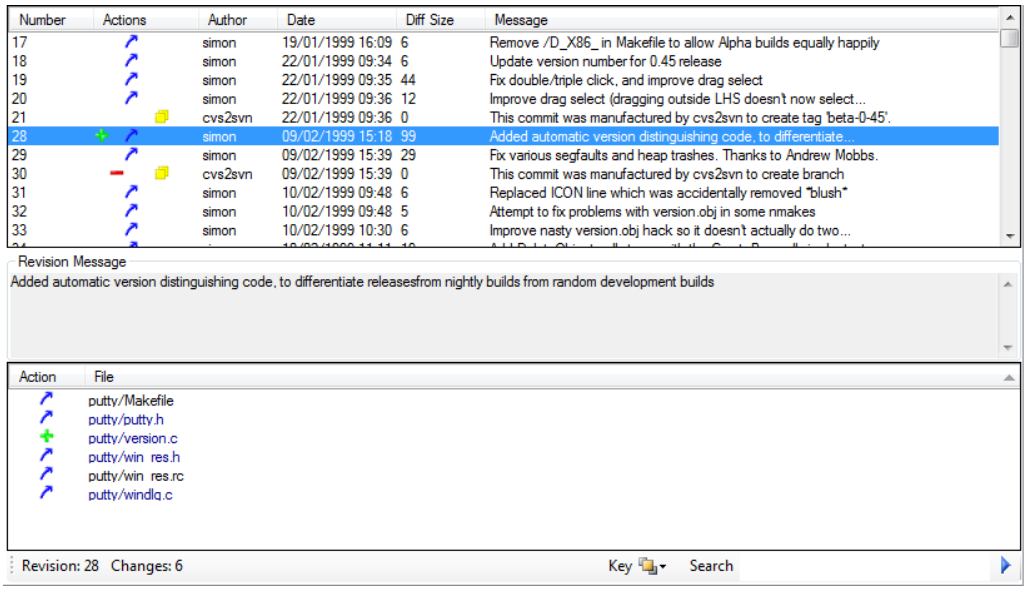


Figure 7.3: The list of revisions – activity types are coded by icons, and file types are coded by colour. In this example, the selected revision contains six changes – one new file and five modifications to existing files.

information specific to that selection. For example, as shown in Figure 7.7 it is possible to see the contributions of a single student to a project.

7.2.2.3 Visualizations

A series of visualizations were integrated with `Perceive` to aid project comprehension. These were based on existing techniques and were selected based on two criteria. Firstly, the input data of the visualization must be suitable to match the data generated by the RCS analysis, and secondly, the visualization must provide useful functionality to the tool and be appropriate to the type of tasks performed by project managers.

7.2.2.3.1 Modified Icicle Plot (MIPVis)

The first visualization implemented was a modified icicle plot (Barlow and Neville, 2001), as shown in Figure 7.8. This visualization was designed with the dual purpose of showing a simple layout of the directory and file structure

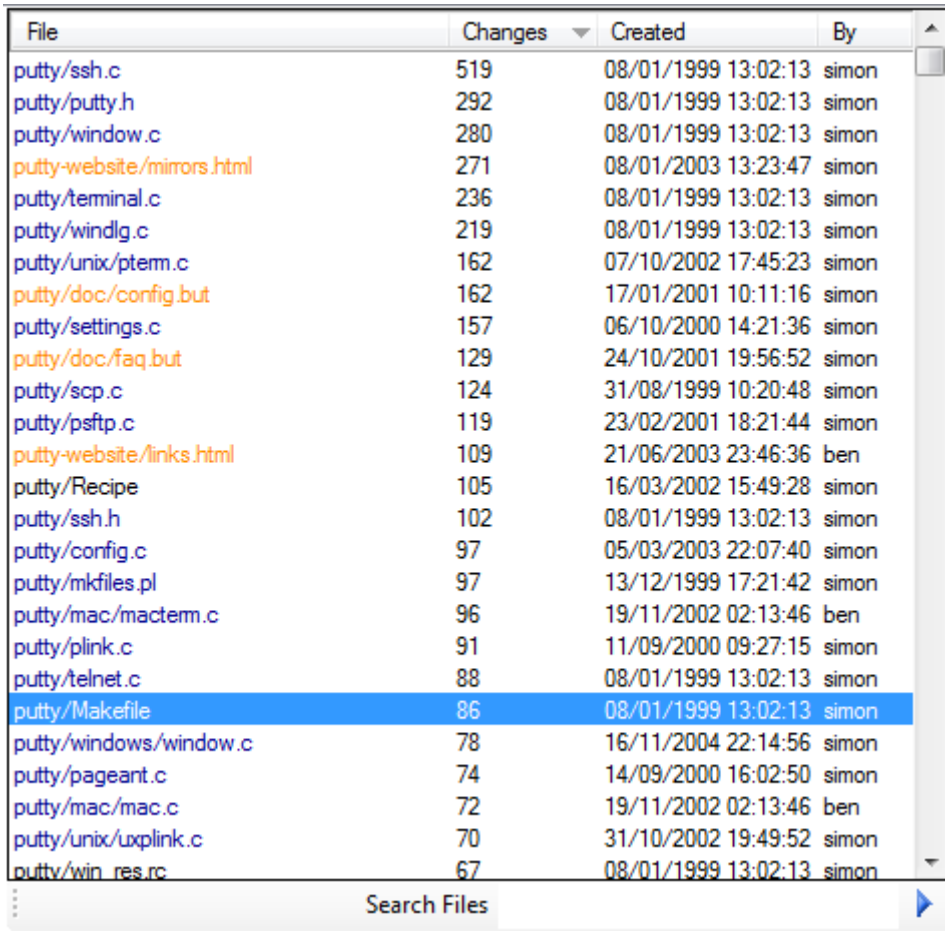


Figure 7.4: This feature shows the number of times each file has been modified, allowing users to quickly see which areas of the project are the most active. This figure shows that the most active files in the PuTTY project are the core source code files and the web pages. This list can also be sorted and filtered to more easily locate files of interest.

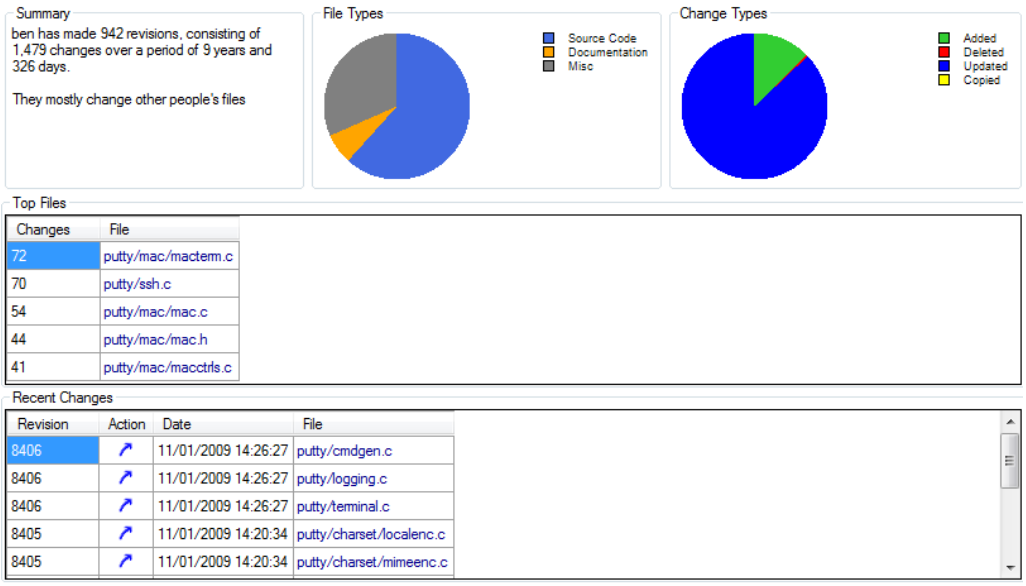


Figure 7.5: A developer's page – this shows the information available about each developer in the project. This example shows one of the PuTTY developers; he works primarily with existing files, and is spread over a large area of the project.

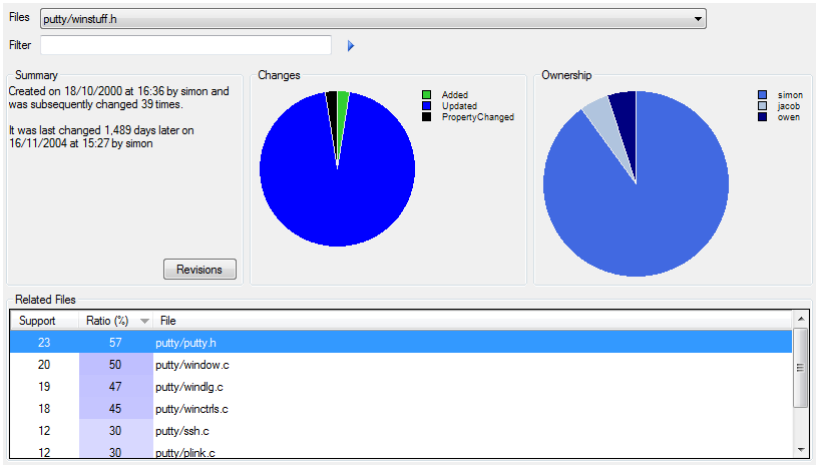


Figure 7.6: Information available for a single file in a project. This example is one of the files in PuTTY, and shows that it is edited almost exclusively by one developer.

	Overall	Selection
Start Date	22/11/2007	08/02/2008
Last Date	14/03/2008	14/03/2008
Project Length	114 days	36 days
Students	9	1
Files	2,554	623
Revisions	490	91
Messages	401 (81%)	90 (98% of selection)
Changes	4,786	904
Additions	1,436 (30%)	401
Deletions	275 (5%)	39
Updates	1,792 (37%)	448
Changes Per Revision		
Mean	9	9
Range	1 - 296	1 - 251
Diff Size	395,448 lines	88,269 lines

Figure 7.7: The project overview feature with a single student selected, highlighting their contribution to the project. The graphical elements provide a quick visual cue to how much of the project is contained in the selection – in this example the selected student has been contributing to the project for a third of its duration, and by all three metrics (revisions, changes and change sizes) has contributed about one fifth of the total effort.

of the project in a way already familiar to the user (a standard hierarchy) and to show much deeper information about the files in the project. Each entity in the visualization is colour-coded based on the file type – individual files are directly related to the file type, while directories are graded based on the files they contain. Arcs drawn between files show file relationships using the same impact analysis system as described in Chapter 6. If a student or revision subset is selected, then transparency is used to reflect that selection's contribution to the overall project, as shown in Figure 7.9.

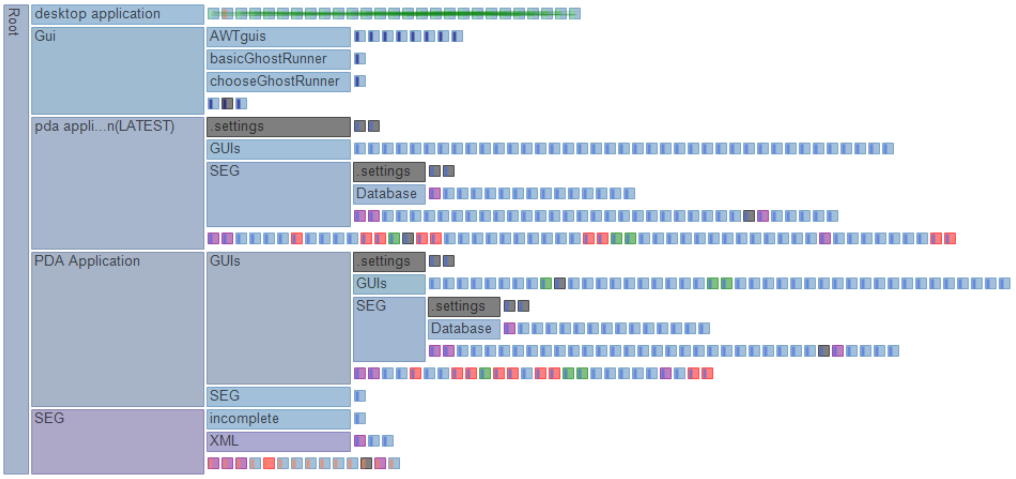


Figure 7.8: The Modified Icicle Plot (MIPVis)

The MIPVis provides a great deal of information to the user – the structure of the project, relationships between files, distribution of file types and distribution of effort. This information is limited to the state of the project in the latest selected revision, and does not show any changes to the project over time.

Figure 7.8 shows a small project rendered with MIPVis. It is read from left to right, with the large rectangles being directories and the smaller boxes being files. Colour coding is based on file types – for example, the `Gui` directory and its subdirectories consist almost exclusively of source code files, and as such the directories are blue too, whereas the `SEG` folder contains a mix of file types (source code, data, archives and uncategorized files) and

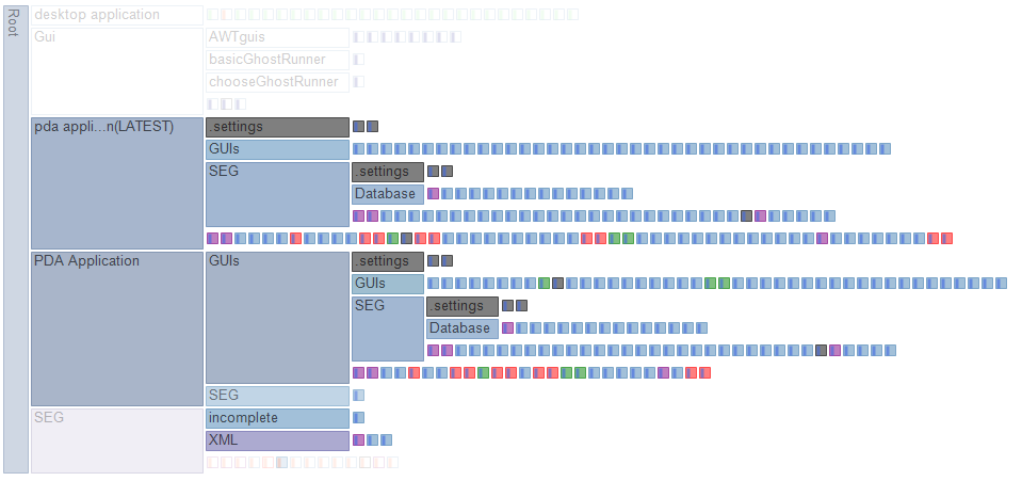


Figure 7.9: The MIPVis with a single user selected

the directory's colour is blended to reflect this. The `desktop application` directory at the top has a number of lines drawn between its files – these lines represent inferred links, using the algorithm described in Chapter 6. The threshold values for this algorithm can be changed by the user, or the lines can be disabled altogether.

Figure 7.9 shows the same project but with only one student selected. When a selection is made the visualization uses opacity to show how that selection affects the project. In this example, the student has done no work on the `desktop application` and `Gui` directories, while substantially contributing to the remainder of the project, with the exception of some files in the `SEG` directory.

7.2.2.3.2 Radial Visualization (RadVis)

The second visualization implemented displays the same information as the MIPVis, but in a different format, that of a radial tree, which has been shown to be highly comprehensible in comparison to other visualizations with similar aims (Padda et al., 2009). The RadVis, as shown in Figure 7.10, is more space-efficient, able to display larger projects than the MIPVis, but is also more cluttered and requires more experience to easily understand the

information shown.

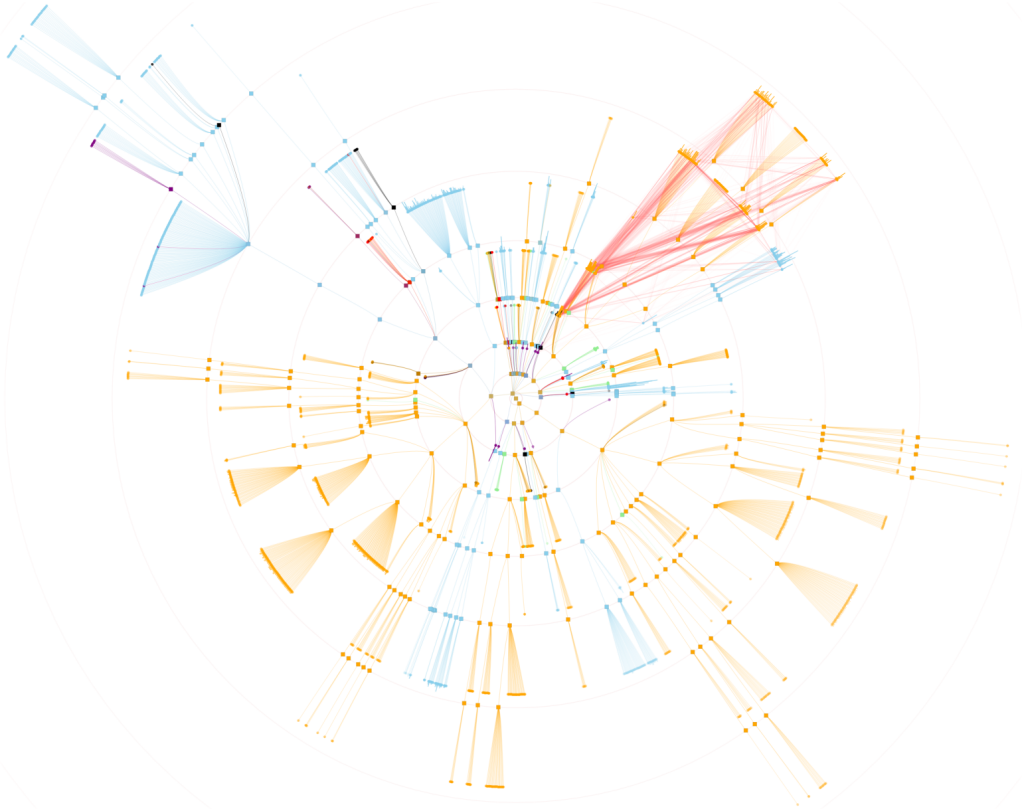


Figure 7.10: The radial visualization can show much larger projects than the MIPVis

Figure 7.10 shows a Radial Visualization of one of the larger SEG projects. As can be seen in the image, orange files – documentation – dominates the project. In this project the documentation is the automatically generated JavaDoc files. In the top right is a series of red lines linking files together. As in MIPVis, these lines represent inferred links between files, and it can be seen that one cluster of documentation is strongly linked within itself, and has some links to its related source code.

7.2.2.3.3 Hilbert Visualization (HilbertVis)

The Hilbert Visualization again shows much of the same information as the MIPVis and RadVis, but sacrifices the directory structure for simplicity and space efficiency (Breinholt and Schierz, 1998). As shown in Figure 7.11 it can visualize very large projects in a relatively compact space. Rather than show the directory cluster, files are placed on a Hilbert Curve, a space-filling curve with good locality-preservation. Elements near to each other in the visualization tend to be near each other in the project structure.

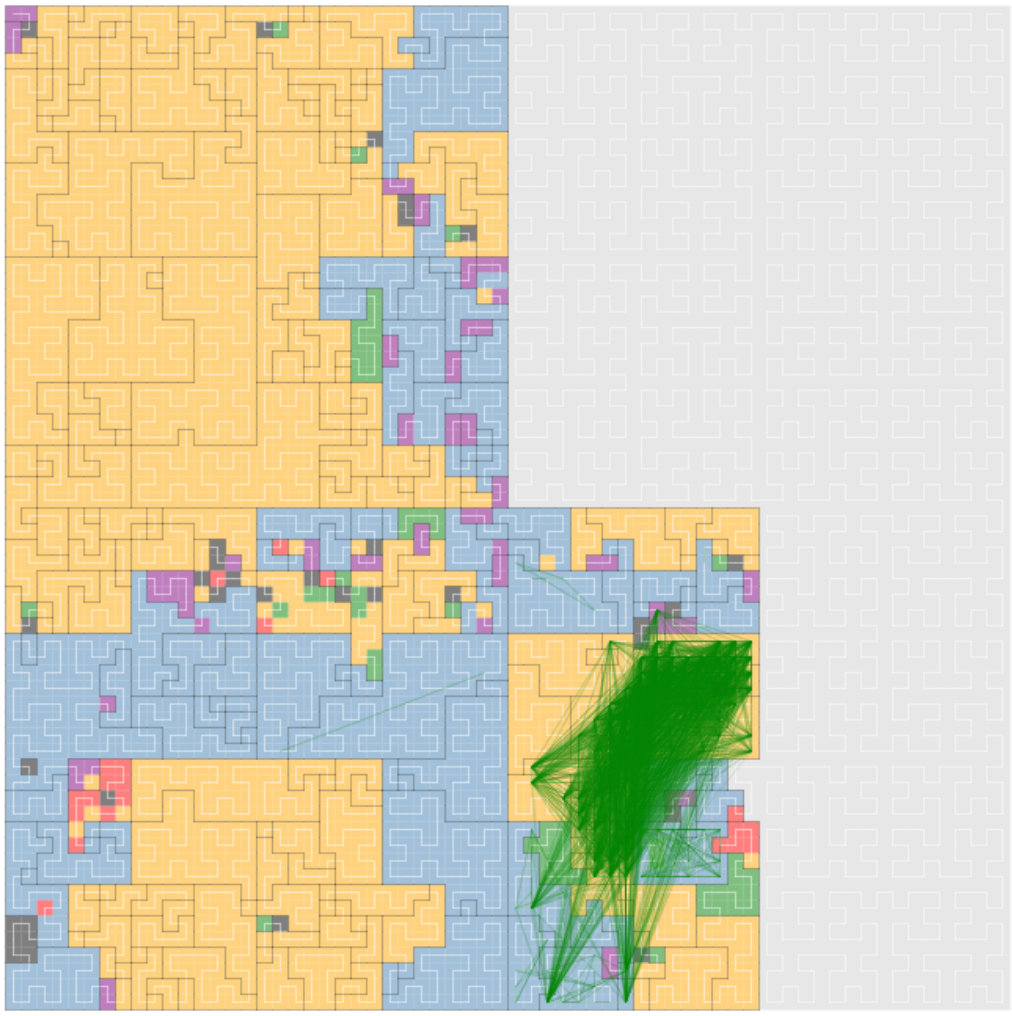


Figure 7.11: Files plotted along a Hilbert Curve

The HilbertVis is more customizable than the previous visualizations. The elements can be sorted and coloured by a series of metrics, as shown in Figures 7.12 and 7.13. The files can be sorted by name, time since creation and time since the last edit; files can be coloured based on creator, last editor, file type, activity and change types.

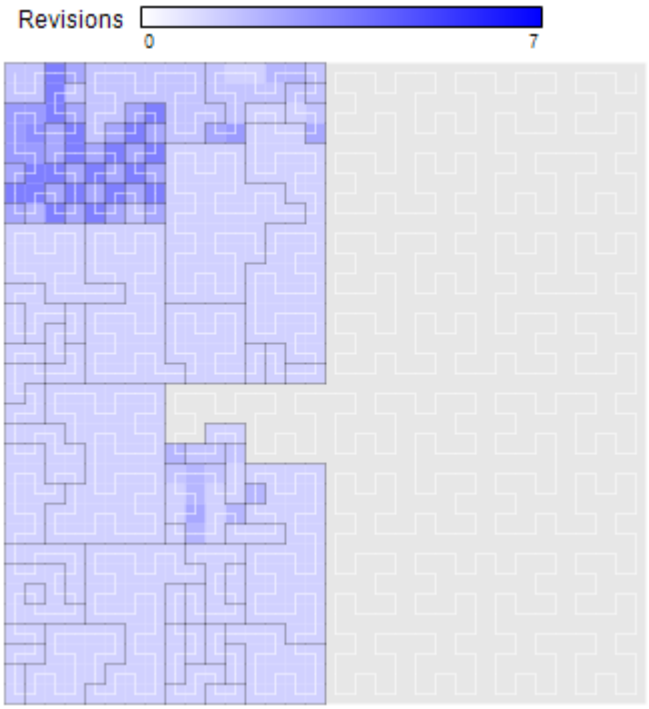


Figure 7.12: A HilbertVis with the files coloured by the number of modifications

Figure 7.11 shows a complete project. Each square represents a file, and in this example is coloured by file type. This is the same project as shown in the RadVis in Figure 7.10 and clearly shows the documentation files being a large part of the project. The white trail leading from the top left corner is the actual Hilbert Curve itself, along which the files are plotted. The darker, irregular shapes around clusters of files represent directories. In this example the files are ordered by name, and so no directory will be split up – something not guaranteed in other orderings. Finally, in the bottom right is a cluster of green lines showing linked files, which can be disabled or tweaked if it

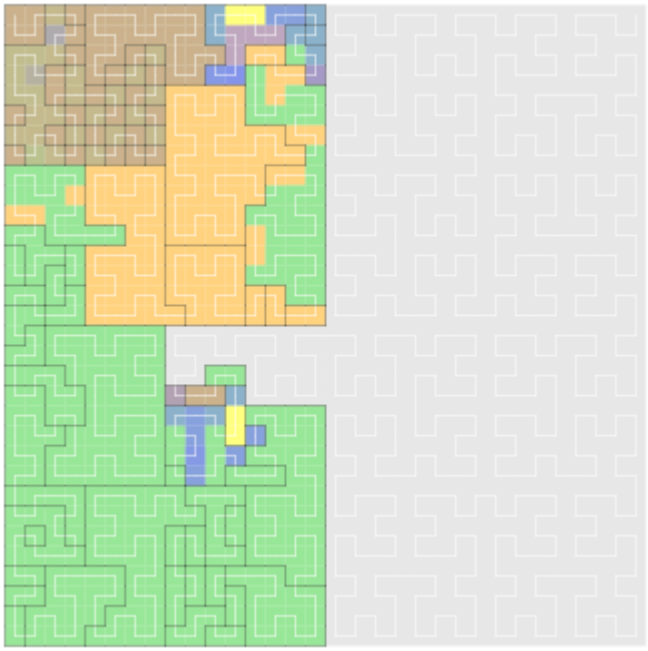


Figure 7.13: A HilbertVis with the files coloured by change type

obscures desired details.

A smaller project is shown in Figure 7.12, this time colour coded by activity rather than file type. It can easily be seen that the most modified files are in a contiguous set of directories, while the remainder of the project is largely inactive.

The same project is shown in Figure 7.13, but with the files coloured by change type. The files identified in Figure 7.12 to be inactive are here shown to be purely green, which indicates they were created and then never altered. The active files in the top left are a blend of colours, reflecting a variety of activity types.

7.2.2.3.4 Flow Visualization (FlowVis)

The MIPVis, RadVis and HilbertVis all show the state of a project at a given point – typically the latest revision. FlowVis shows the progress of a project over time, using a grid with files on the X-axis and revisions on the Y-axis,

as shown in Figure 7.14.

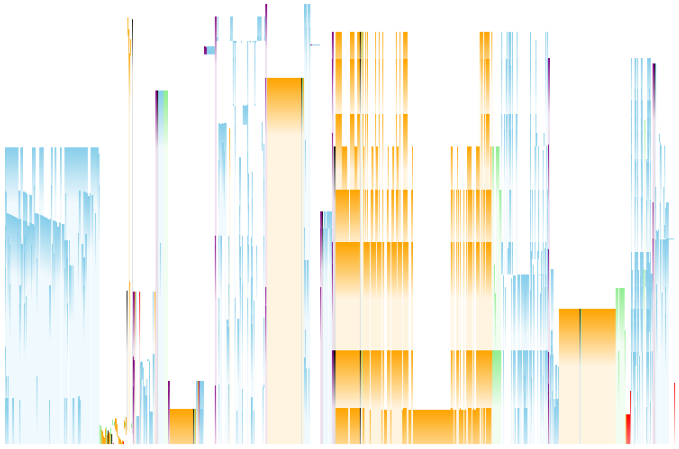


Figure 7.14: A section of a Flow Visualization, showing activity on a variety of file types over time

In terms of space efficiency, FlowVis is highly dependent on the number of files in the project – each file requires one pixel of width, and each revision requires one pixel of height. On larger projects with more files, it is only feasible to generate a FlowVis for subsets of the project. However, when it is rendered, it can show a great deal about how activity changes across a project as time goes by, which files or modules are worked on, and by whom.

The example shown in Figure 7.14 shows a subsection of a large project, revealing a mix of documentation and source code. Reading downwards, as a file becomes inactive it fades to near-transparency, and then when it is modified again it reverts to opaque. The example shows that the documentation is updated at regular intervals – reflecting good practice – and in the bottom left corner can be seen a small set of new and highly active files created in the final few revisions before the deadline.

7.2.2.3.5 Owner Visualization (OwnerVis)

Like the FlowVis, the Owner Visualization shows how a project changes over time. While the previous visualizations focussed on project structure and activity, OwnerVis focuses on how file ownership changes over time. Users

are displayed evenly in a circle, and whenever a file is created it is placed on a point between the centre and its creator. Whenever a file is changed, it is moved slightly towards the student who changed it, and a trial is left, showing how the file's ownership has changed. Green circles are shown where files are created, and blue circles show where they end – the size of the circle is proportional to the number of file at that exact point.

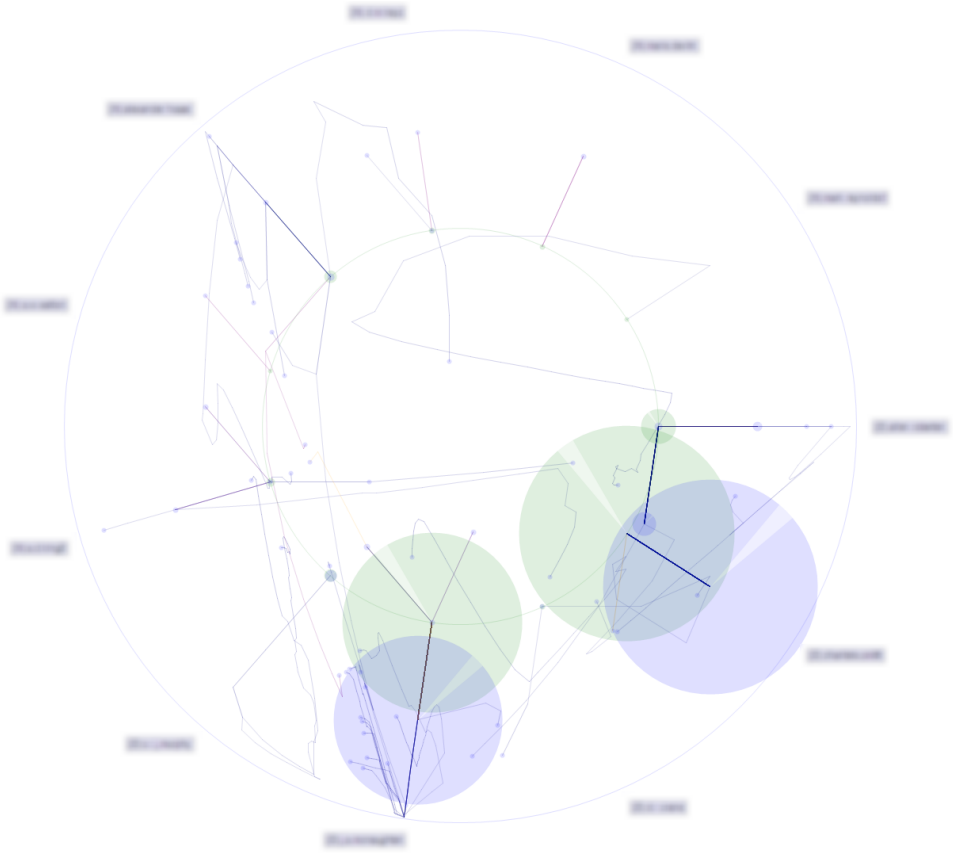


Figure 7.15: OwnerVis, showing how files move between users over the length of a project

A stationary file is one which was created and never modified. Files which move towards their creator are only ever modified by their creators, and files which move in straight line towards a different user are created by one user and then only ever modified by another. Files which meander across the visualization are ones with no clear owner and which are worked on by

many users. Unlike the previous visualizations, it is not possible to identify individual files – only project-wide trends can be seen.

The example in Figure 7.15 shows that two users are responsible for the creation of the majority of the files in the project, three of the users made few contributions and the remainder were primarily working with files created by the others.

7.2.2.4 Results

Despite being offered training in the software, few managers took it up. Debriefing interviews with the managers revealed that few of them made use of the software to support their roles, for a variety of reasons. Managers felt that they had too little time available and did not see any perceived benefit in using a new tool. One manager said that their group used pair programming and so Perceive would not accurately reflect the group's work. Another manager felt that their group progressed well and required no additional support, while a third said that they used the tool very briefly to reinforce his assessment of his group, but then used it no further.

The remaining managers offered useful advice for the next iteration of the software. One manager said that while they did not use the visualizations, they were what attracted them to the software in the first place. They suggested simply using charts to plot the metrics – they had no need of complex visualizations, but a visual way of quickly being able to track individual students' efforts and progress would be invaluable. Another manager suggested a "progress report", a single page showing the activity since the previous group meeting, so that the group would have a way to assess their progress in an open, visual manner.

Both of these features were implemented. Figures 7.16, 7.17 and 7.18 show example of the graphs generated by the application, and Figure 7.19 shows an example of a one-week progress report.

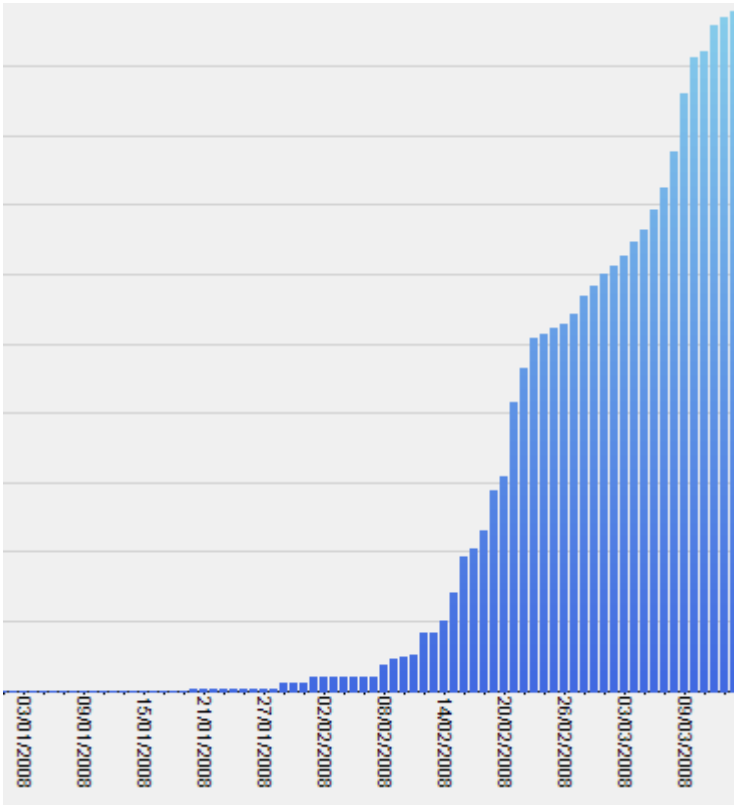


Figure 7.16: A bar chart showing the cumulative number of revisions in the project over time

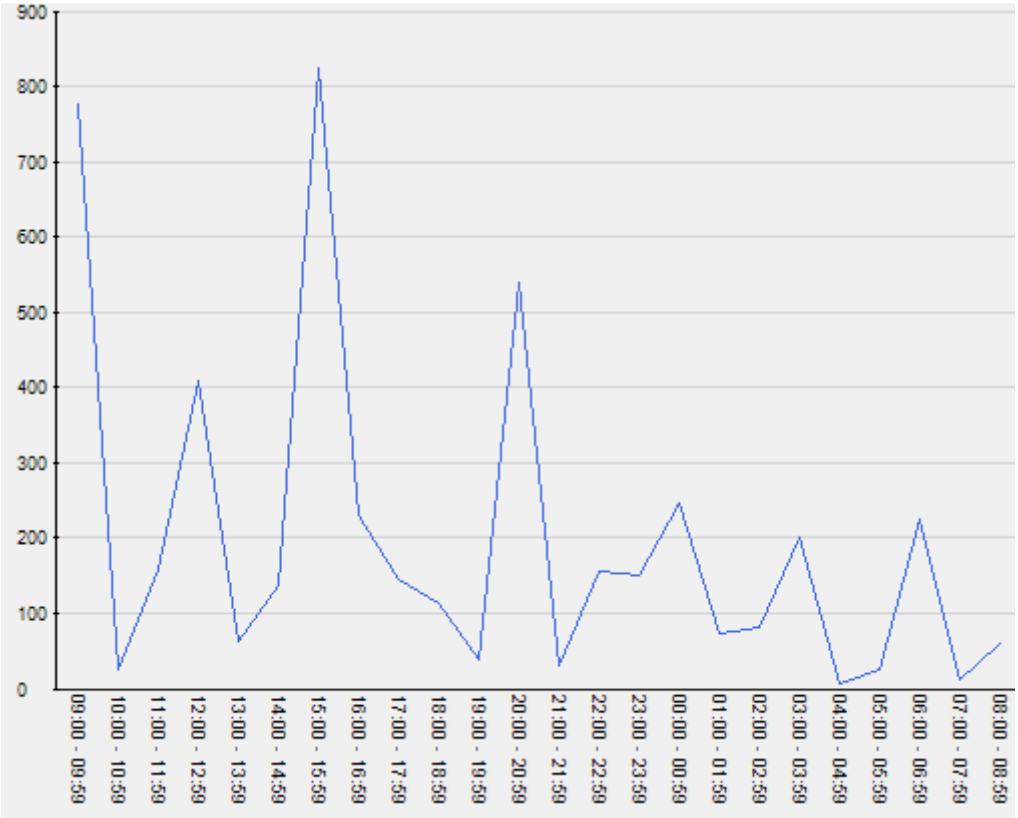


Figure 7.17: A line chart showing the aggregate number of changes submitted, grouped by time of day

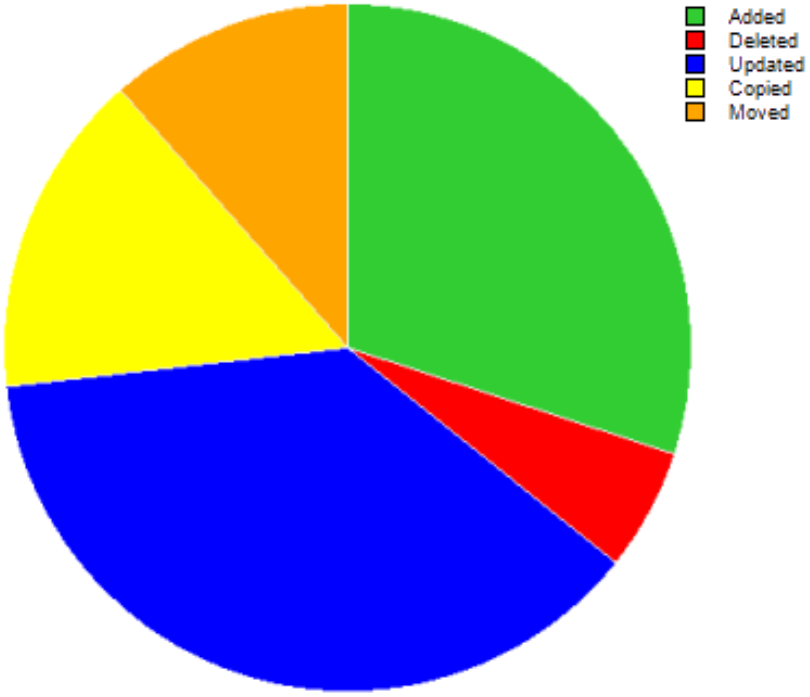


Figure 7.18: A pie chart showing the distribution of change types across the project

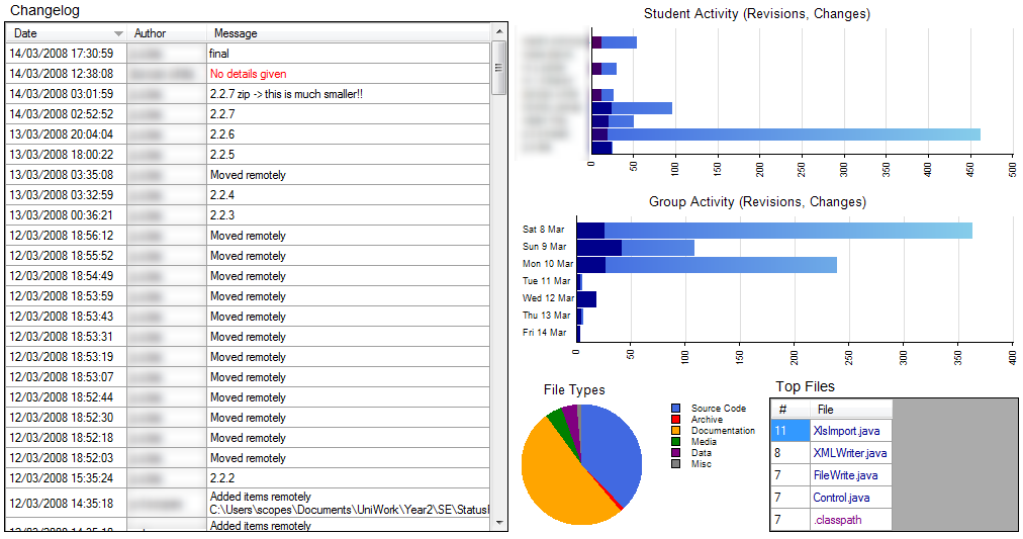


Figure 7.19: A seven day progress report generated for a group by Perceive

7.2.2.5 Outcomes

As described in Chapter 2, only a small proportion of the software visualization tools ever developed see widespread use in the software industry (Price et al., 1993). This has been ascribed to an innovation-centric approach to design rather than a user- or task-centric approach (Hundhausen et al., 2002), whereby researchers develop visualizations to provide novelty or innovation rather than identifying tasks or applications which would benefit from visualization support. Combined with the lack of empirical research and evaluation of software visualization (Burn et al., 2009) this significantly reduces end-user adoption.

This effect was demonstrated clearly in this study; the visualizations developed were designed to satisfy common requirements for visualization design – for example, representation, abstraction, navigation, correlation, automation, interaction and scaling (Young, 1999); the choices of information to be visualized were drawn from experience with software development and groupwork. Despite these factors, the managers did not make use of `Perceive` to support their work and did not take the time to explore the features to see how they could be used to assist them.

This reflects the findings of previous research, reinforcing the need to work with the target users to develop features which they believe necessary, and to present them in a way which required no further training or investment of time. Following feedback from the project managers, the implemented features are much more task-oriented and reflect what the managers really want from a support tool.

These findings highlight the conclusions of many in the software visualization field: visualizations and tools *must* be developed for the end-users and designed to perform real tasks, solving actual problems, not just implementing features the designer thinks would be good, or are new and innovative.

7.2.3 Case Study Discussion

This section discusses the findings of the project management study in the wider context of software visualization and project comprehension, and addresses the case study's research questions, which are restated here:

- *RQ 7*: Which RCS-based visualization techniques best support student project managers?
- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques?

7.2.3.1 *Research Question 7*

- *RQ 7*: Which RCS-based visualization techniques best support student project managers? ***Not proven***

Just as with industrial project managers, the primary motivations the student project managers cited were whether they felt the tools addressed their needs, and whether the tools would require sufficiently little training and effort to adopt. Therefore, their feedback indicated that they found most use for the simpler and more familiar features; notably the charts and the progress report, neither of which required training and provided clear information they deemed useful.

This research has led to the conclusion that when using visualizations of RCS data to support project comprehension the key factors are familiarity, simplicity and clarity of function – the end-users must be able to use the system rapidly with little to no training and also immediately understand *why* they should do so, and what benefit the system will bring them.

7.2.3.2 *Research Question 8*

- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques? ***Not proven***

While it is clear that the needs of student and industrial project managers are much different in terms of personnel, platform, technology, scale, budget and resource constraints, many of the motivations and behaviours exhibited by the students are the same as those of industrial practitioners.

The necessity for tools to accurately and clearly address the requirements of the managers in order to justify adoption and training is the same regardless of whether the manager is a student or an industrial practitioner. An important lesson learnt from empirical software visualization research is that the needs and constraints of the end-user are more often than not secondary to the academic desire to "fill a niche", to address a problem which might not exist in practice, let alone require solving.

It can be difficult to find practitioners to use as subjects in empirical studies, and too frequently this means that visualization tools never progress beyond implementation and a proof-of-concept technical demonstration incorrectly described as a case study. This raises the possibility of using students as "first round" subjects for empirical software visualization research – students may lack the experience or requirements of industrial practitioners, but they do share the resistance to adopt new tools and techniques that are not perceived as being immediately necessary. If a preliminary experiment using students shows a willingness to adopt the tool, or an acceptance that the tool addresses a real problem in a useful manner, then further studies can be conducted in an industrial setting.

Despite this case study showing that students have some similarities to industrial practitioners, there is no formal evidence to support this outcome, and therefore *RQ 8* cannot be categorically answered.

7.2.4 Case Study Conclusions

This case study was conducted with the goal of evaluating software visualization techniques based on repository analysis in the context of determining if such techniques support project comprehension. There was insufficient evidence to answer research questions *RQ 7* and *RQ 8*, and so software

visualization of repository data cannot be definitively identified as a technique which supports project comprehension. However, the case study has identified avenues for research which could re-address the two research questions using empirical, user-centric studies.

7.3 Case Study: Project Sampling

The tendency of a project to grow over time can render it impossible for a developer to maintain a comprehensive mental model of the entire project. Furthermore, the performance some algorithmic processes, such as visualization, are dependent on the size of a project. For example, several of the visualizations described in Section 7.2.2.3 use data structures with complexity of $O(n^2)$, which can reduce performance or require increased computing capacity to execute. As described in Chapter 2 many visualization techniques do not scale well in terms of comprehensibility – even if they successfully execute, the amount of data being presented can render it useless to the user.

Therefore, there are a number of scenarios in which it is desirable to select a subset of a project, which should be representative of the project as a whole. This case study explores how a process of project sampling – the selection of a representative subset of files – can be used to support activities related to project comprehension by reducing the amount of data presented without in some way distorting the data.

7.3.1 Case Study Design

This section details the goals, techniques and evaluation of the case study using the DECIDE framework. The subject of the case study is a quasi-experiment[1] which evaluates the effectiveness of `Perceive` at generating a representative subset of a series of student projects in the context of student assessment.

[1] A quasi-experiment is a controlled experiment in which the allocation of subjects to treatments is not or – as is the case in this research – cannot be randomized.

Figure 7.20 shows the structure of this case study, and how the sub-study feeds into the parent case study, which in turn evaluates the results in the context of the wider research goal of identifying ways in which repository analysis can support project comprehension.

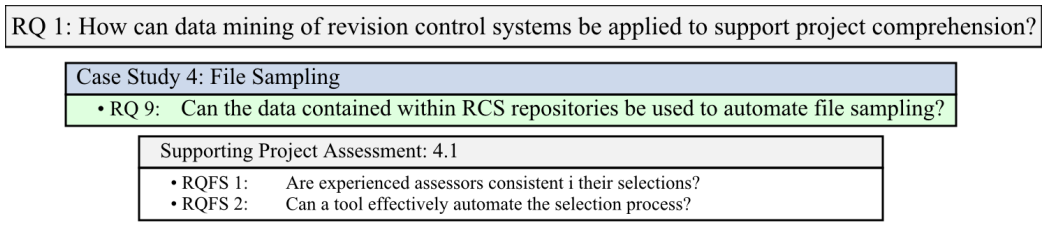


Figure 7.20: Structure of the research questions and studies of the file sampling case study

7.3.1.1 Research Goals

This study investigates whether it is possible to algorithmically generate a subset of a project that is representative of the whole. If it is shown that an automated tool can successfully generate a representative subset, it will have a range of implications, including assessment, visualization, metric processing and thematic analysis.

7.3.1.2 Research Questions

This case study has the following research question:

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling?

7.3.1.3 Evaluation Paradigm and Techniques

The experiment reported here will evaluate the success of `Perceive` to automatically generate a representative subset of files from a project in the context of academic assessment. The results of the experiment will then be

discussed in the wider context of using a project sampling process to support project comprehension activities.

7.3.1.4 Practical Issues

The subjects of the experiment are a series of student group projects, taken from a single cohort. These projects are allocated to human experts, who perform a selection process. The human subjects are a combination of lecturers and postgraduate students, all of whom have experience in assessing summative software development projects.

7.3.1.5 Ethical Issues

Ethical clearance was provided to make use of the student projects, and the experts completed consent forms allowing their results to be used and published.

7.3.1.6 Evaluation and Discussion of Results

After the experiment is reported, the findings will be discussed in the context of the case study's research questions described above, relating the study's outcomes to the overarching research goals.

7.3.2 Study 4.1: Supporting Group Project Assessment

7.3.2.1 Research Aims

Assessment of software projects is difficult, having to capture a series of objectives for which there are often no clear assessment criteria, relying instead on the subjective evaluation of the assessor. Group projects are even more difficult to assess as the factors are compounded by the need to capture the effort and achievement of individuals within the group as well as that

of the group as a whole, to somehow assess the collaboration itself without anomalous individual performances overly impacting the marks of others.

Assessment of the implementation phase of the Durham University SEG projects forms only a part of the overall score, and assessment of the code itself is only one of a number of aspects of the implementation that is assessed. However, students typically place great emphasis on the source code, and a transparent, repeatable assessment process, less dependent on the subjective opinions of the assessor would increase student confidence in the assessment.

In large projects, it is clearly not feasible to read, comprehend and assess every file in its entirety. For context, one SEG project was delivered for assessment containing over 1,200 Java source files, and of the projects available from 2006 to 2008 the mean number of Java files is over 250. Because of this, the first thing an assessor must do is choose a subset of files which will be assessed. This is a non-trivial process, as the selection must capture a fair sample of students' contributions – if five students write code, but only work from two of them are assessed, this leads to an unfair outcome. The selection process will also ideally capture 'interesting' files, which demonstrate talent, inventiveness and collaboration.

7.3.2.2 Research Questions

This section reports a quasi-experiment investigating a single aspect of the code assessment process – the process of selecting a subset of code entities for assessment. Using a set of SEG projects, the experiment empirically addresses the following research questions:

- *RQFS 1*: Are experienced assessors consistent in their selections?
- *RQFS 2*: Can a tool effectively automate the selection process?

Therefore, the following null hypotheses are proposed:

- *HFS 1*: Assessors select similar subsets of code entities.
- *HFS 2*: An automated tool can effectively automate the process.

7.3.2.3 Experimental Design

The design consists of six assessors, all with experience of assessing Java-based group projects, and eight SEG projects. The projects were drawn from the same cohort, and were stratified by the final implementation mark achieved. The projects and assessors were each assigned random aliases and each assessor was assigned four projects from the stratified blocks, so that each assessor received one top tier project, two middle tier projects and one bottom tier project, and each project was allocated to three assessors, as shown in Table 7.1.

Assessor	P2	P7	P1	P3	P4	P6	P5	P8
A1	x		x	x			x	
A2		x			x	x	x	
A3	x			x	x			x
A4		x	x			x		x
A5	x		x		x		x	
A6		x		x		x		x

Table 7.1: Allocations of projects to assessors

It was decided that code entities would be selected at the file level, as almost every file consisted of a single class, and further granularity would place an undue burden on the assessors. In addition, informally, assessors indicated that the selection of code entities would be made at the file level in a normal assessment.

Each assessor was provided with a copy of the projects they were assigned. Each project had been processed such that only Java code files were included, and obvious library code (such as code provided to each group) had been removed. The remaining files provided the corpus of code entities from which the selection was to be made. This cleaning process was simply to make the experiment more straightforward for the assessors, and would not affect the final selections. Assessors were given instructions to select a set of files which

they would use to assess the project, although no actual assessment would take place. No limits or requirements were placed on the number of files to be selected, nor were any criteria or guidelines included, as there are no equivalents in the real assessments.

Each assessor returned a list of files for each project. These lists were transformed into a set of binary decisions – include/exclude – for every file in the project; a comparison could then be made for each project between their three assessors to evaluate how consistent the selections were. Each project was also processed by an automated tool, which returned three sets of results for each project, using a different set of criteria for each model.

A coverage metric (described in Section 7.3.2.5) is used to compare the coverage of a project achieved by the experts and by `Perceive`. These results are statistically compared to determine what degree of coverage experts and `Perceive` achieve for each project.

7.3.2.4 The Models

Perceive was programmed with three models for selecting a subset of files for a project. The three methods are:

- Activity-based ($P_{Activity}$): The most modified files are suggested on the basis that the more active files are likely to provide a better indication of effort. The primary sort is on the number of modifications made to a file (`num changes`), and the secondary sort is on the number of users who modified the file (`num students`).

- Student-based ($P_{Students}$): Suggestions based on the number of students who have modified the file; a wider spread of students provides evidence of collaboration. The primary sort is on the number of students who modified a file (`num students`), and the secondary sort is on the total modifications to the file (`num changes`).

- Hybrid (P_{Hybrid}): A combination of $P_{Activity}$ and $P_{Students}$, calculated as (`num changes` $\times$ `num students` $\div$ `total students`).

Each model returns at most 10 results, and this list is then truncated at the point at which it falls below a threshold: $P_{Activity}$ and P_{Hybrid} cut off when the metric drops to a third of the highest value, and $P_{Students}$ cuts off when the metric drops to a half of the highest value. This is to stop the models from returning too many files, achieving a high recall at the cost of precision. If the numbers of files returned are considerably out of line with those of the assessors, then future iterations of the models will be modified to return more appropriately sized lists.

In Section 7.3.2.7 the three models are compared against the results from the expert assessors to determine which, if any, are effective at automating the process of file selection.

7.3.2.5 Coverage Metric

To more effectively compare the performance of `Perceive` and the experts, a *coverage metric* is devised to measure what proportion of a project is represented by a selection of files. The metric works by measuring how many of the users are represented by a file set, and in how many revisions those files are modified, normalized to [0..1].

To control for the fact that `Perceive` selects up to ten files per project while the median number of files for the experts was three, with a mean of four, a number of variations of `Perceive` were used, which limited the selections to varying amount of files; these variants were named accordingly, e.g. $P_{Hybrid4}$ or $P_{Students2}$.

7.3.2.6 Limitations and Threats to Validity

The main limitation of this study is the relatively small number of projects examined, and the number of assessors involved. A further study would attempt to expand the range of projects used, and to increase the number of assessors. This would have the dual advantage of increasing the significance of the results within the context of the SEG projects, while widening the scope of projects to which the results can be generalized.

There are also potential problems with the algorithms used in Perceive – the activity and student metrics are dependent on the data extracted from the source code repository being representative of each group's work. However, as described in Section 7.2.2.4 feedback from project managers has revealed that some groups make use of pair programming, and all work for a pair is submitted under one username; other groups do significant work "offline" and commit a single revision consisting of a large amount of effort. The former can be accounted for by understanding how each team has worked using feedback from their managers. The latter is a harder situation to address, but the behaviour is easily detected using a tool to view repository activities – infrequent spikes of activity are easily identified.

The measures used to implement the coverage metric are the same as those used by `Perceive` to generate its file sets, and so might be expected to favour the tool. However, it should be noted that at least one of the experts explicitly stated that their strategy involved looking files with more authors, considering them more interesting. Overall, when assessing a group project it is not unreasonable that the work of as many contributors as possible is examined, and that the files should be indicative of as much of the development process as possible.

7.3.2.7 Results

When the assessors had completed the task, one project (P3) had to be discarded as two assessors could not determine which of a number of copies of the software were to be assessed. The remaining seven projects had complete data and were used in the analysis. Table 7.2 shows a summary of the results for each project. *Singles* refers to files suggested by one assessor, *doubles* to files suggested by two, and *triples* to files suggested by all three.

Table 7.3 shows the number of files suggested for each project by each assessor. Assessors with results for only three projects were those assigned P3.

Table 7.4 shows the coverage achieved by each assessor and model for each

Project	Total Files	Unique Files	Singles	Doubles	Triples
P1	6	4	2	2	0
P2	11	9	7	2	0
P4	14	12	11	0	1
P5	15	13	12	0	1
P6	17	13	10	2	1
P7	16	11	7	3	1
P8	8	7	6	1	0

Table 7.2: Summary of the number of files suggested for each project

Assessor	P1	P2	P4	P5	P6	P7	P8
A1	2	4		2			
A2			9	11	12	10	
A3		5	3				4
A4	2				2	3	2
A5	2	2	2	2			
A6					3	3	2

Table 7.3: Number of files suggested for each project by each assessor

project. Only the results for the models which generated sets of four files (e.g. $P_{Hybrid4}$) are shown, as four was the mean number of files selected by the experts, and the coverage only improves as the number of files selected increases.

Project	Expert 1	Expert 2	Expert 3	$P_{Activity4}$	$P_{Students4}$	$P_{Hybrid4}$
P1	23%	23%	18%	31%	32%	39%
P2	32%	26%	38%	41%	43%	41%
P4	48%	34%	26%	49%	35%	47%
P5	20%	44%	21%	42%	23%	42%
P6	35%	14%	27%	41%	27%	41%
P7	63%	18%	25%	48%	49%	48%
P8	25%	33%	29%	55%	43%	55%

Table 7.4: Coverage achieved by each expert and model

7.3.2.8 Evaluation

7.3.2.8.1 RQFS 1: Are Experienced Assessors Consistent in their Selections?

The first research question, "are experienced assessors consistent in their selections?" can be addressed by first examining the number of files selected, then the overlap in file selection and finally by looking at the selection strategies used.

The mean number of selected files was 4, with a standard deviation of 3.3 – there was little consistency between assessors as to how many files to select, although 2 to 3 is the most common range. Comments from some of the assessors indicate that they would use the suggested files as a starting point from which to continue the assessment, following the class structure to other files.

As can be seen in Figure 7.21 a file is only suggested by all three examiners in projects where a larger amount of files are suggested. Files are suggested

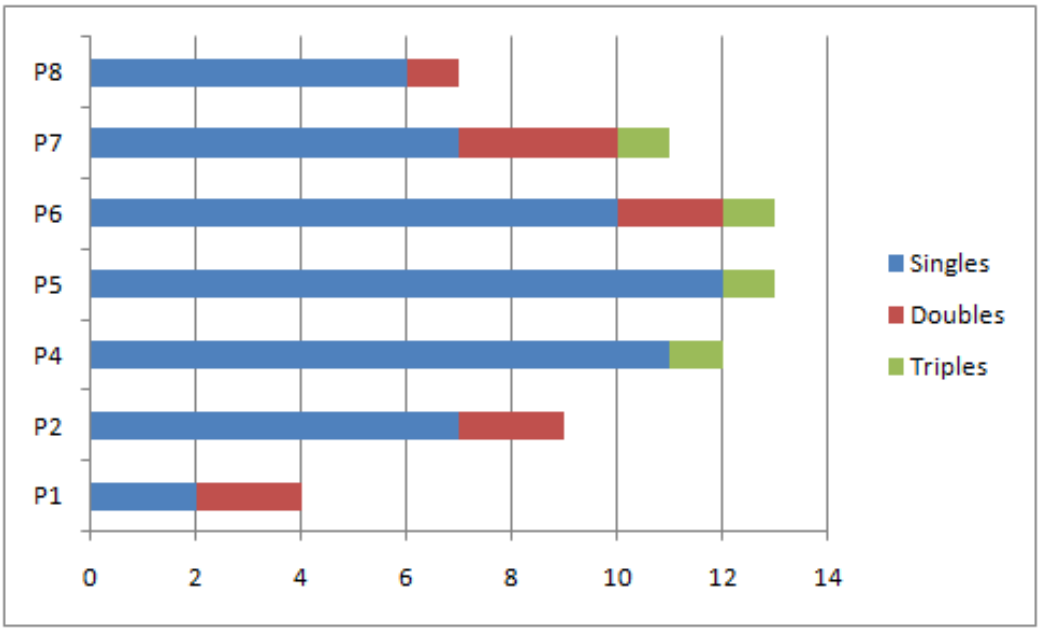


Figure 7.21: Overlap between file suggestions – one assessor, two assessors and three assessors

by two of the three assessors more routinely – in no project are there no files suggested by more than one assessor. On a data set of this size it is infeasible to perform a statistical test, such as a Kappa test (Fleiss, 1971), to measure agreement between assessors. However, it can be seen that there is some agreement, but that this is not consistent. Comments from the assessors reveal a series of different strategies used to select files. These strategies include:

- Check the largest files first, as it is plausible that they would have the most effort put into them
- Ignore GUI files, as they are commonly automatically generated
- Check the file comments for indications of collaborations – the more users that were involved in creating the file, the more likely it is to be representative of the group's work
- Infer functionality from the filename

These strategies can be contradictory and have led in some cases to very different sets of files. This suggests that without specific marking criteria the assessment of a large project can be entirely dependent on the person marking it. None of these assessment strategies can guarantee that a project will be completely assessed, covering work from all students and focussing on code which reflects the effort which was put into a project.

7.3.2.8.2 RQFS 2: Can a Tool Effectively Automate the Selection Process?

The second research question, "can a tool effectively automate the selection process?" requires comparing the selections made by Perceive with the files selected by the assessors. The first step is to compare the number of files selected by Perceive with the experts' lists, and then to examine how much the lists overlap.

Model	P1	P2	P4	P5	P6	P7	P8
Activity	10	4	6	8	8	10	10
Students	5	10	10	10	10	8	10
Hybrid	8	4	9	8	8	10	7

Table 7.5: The number of suggested files found by each model for each project

Table 7.5 shows the number of files suggested by each model of Perceive for each project. The lists are generally much larger than those of the assessors; if an assessor accepted the use of Perceive to suggest files, examining the list of files might well take longer than their own list, but would be generated instantly rather than going through a potentially lengthy process of examining an entire project.

Table 7.6 reveals that none of the models were able to detect all of the files suggested by assessors, even when returning larger lists. If we assume that the most important files are those suggested by two or three assessors then all three models achieve around 50% recall. However, this assumes that

Model	Singles (of 55)	Doubles (of 10)	Triples (of 4)
Activity	12 (22%)	5 (50%)	2 (50%)
Students	17 (31%)	5 (50%)	2 (50%)
Hybrid	15 (27%)	6 (60%)	2 (50%)

Table 7.6: The number of suggested files found by each model, in singles, doubles and triples

the strategies used by the assessors are correct. Looking more closely at the two triples that were missed by the models, we see that they are both route-planning algorithms, and their names suggest functionality attractive to assessment. However, in P7 the file was in fact an alternative algorithm, worked on by only two students and modified only four times – a trivial level of activity in the context of P7; the primary algorithm was only suggested by two assessors, and had equally little activity relative to the rest of the project. The revision logs for these files show that even the follow-up modifications after the initial creation were only minor changes including refactoring and documentation. The route-planning algorithm in P6 however was considerably more active, and is certainly a good candidate for assessment. If the strategies of ignoring GUI files were to be incorporated into the Perceive models – a tractable problem – then the missed P6 algorithm would in fact have been suggested, as many of the suggestions that kept it from the top of the list were highly active GUI classes.

Figure 7.22 shows a comparison of the performance of the three models. As can be seen, $P_{Activity}$ is the least accurate of the models, while $P_{Students}$ and P_{Hybrid} are typically equal, each outperforming the other on one project. More data and evidence would be required to reinforce this outcome, but it can be hypothesized that the number of students involved in a file's history should definitely be a factor in any automated selection system.

T-Tests were used to compare the performance of the variants with the experts. Using the coverage metric as a performance metric, all three models ($P_{Activity}$, $P_{Students}$, P_{Hybrid}) outperformed the experts at a significance

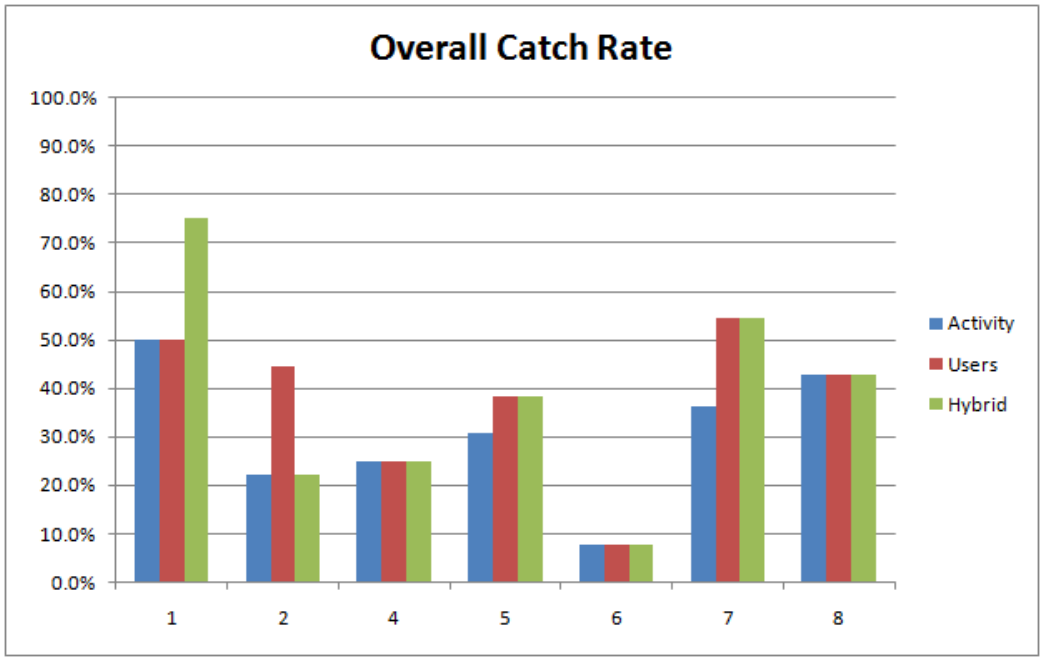


Figure 7.22: Overall performance of the three Perceive models

level of p=0.05 when selecting four files, and P_{Hybrid} significantly outperformed the experts with three files. $P_{Activity2}$, $P_{Students2}$, $P_{Hybrid2}$ also outperformed the experts, but not to a statistically significant degree.

As previously stated, while the measures used to develop the coverage metric are the same that `Perceive` uses to make its suggestions, there is also some overlap with the strategies employed by the experts. For example, I5 stated that they intentionally looked for files with more authors, considering them more interesting. However, I5 did not outperform any of the `Perceive` models in terms of coverage, showing that even when a strategy is in place, it will not necessarily be successful.

7.3.2.9 Conclusions

Group projects typically use flexible marking methods to enable the students to work in a way that suits them, but with something as concrete as code assessors have to use their own strategies for assessment, which leads to

highly divergent sets of files being assessed, potentially under-representing the work of group members. While the Perceive models did not always suggest the same files as the assessors, even when the assessors were in agreement, in general the automated systems were no more or less consistent than the assessors were amongst themselves. As discussed in Section 7.3.2.8.2 if GUI files were removed from the set Perceive processes then the results would be more in line with those of the assessors. However, attempting to modify the algorithm to match the assessors results must be approached with caution, as was demonstrated in P7 where a file selected by all three assessors turned out to be relatively unimportant in the context of the project, and not greatly representative of the group's effort.

When performance is measured using the coverage metric, `Perceive` performs significantly better than the experts, even when limited to generating set of only four files.

In conclusion, the experts are inconsistent between themselves with regards to their selections; `Perceive` therefore performs no worse than the experts. In fact, measuring performance with the coverage metric, `Perceive` outperforms the experts to a statistically significant degree.

Considering the rapidity with which the lists are computed it is suggested that such models are of value to assessors. Some of the assessors considered the lists they created as being "starting points", and that they would expend yet more time and effort following the class structure for more files. In combination, these two facts would suggest that they instead use a tool to generate their initial list, which would allow them to move directly to assessment and examining further files.

Two hypotheses were identified in Section 7.3.2.2 for this experiment, as follows:

- *HFS 1*: Assessors select similar subsets of code entities.
- *HFS 2*: An automated tool can effectively automate the process.

There is no evidence to suggest that we can accept *HFS 1* – assessors

do not appear to select similar sets of files for assessment. We can however accept *HFS 2*: `Perceive` does effectively automate the file selection process, outperforming the experts using the coverage metric described above.

7.3.3 Case Study Discussion

This section discusses the findings of the project assessment study in the wider context of project sampling and comprehension, and addresses the case study's research question, which is restated here:

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling?

7.3.3.1 *Research Question 9*

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling? ***Proven: Yes***

The experiment described here has confirmed that an automated tool can effectively select files from a project that are representative of the project as a whole, using measures which reflect the development of the project and developer contributions.

This study was conducted in the context of selecting files from student group projects for assessment to reduce assessor workload and to improve the stability of the assessment process. The principle extends to other applications however, such as selection of files for visualization in situations where visualizing the entire project is not feasible.

In terms of project comprehension, taking a subset of a project will obviously be limited in the depth and completeness of comprehension that can be attained. However, it allows for a broader overview or a high-level mental model to be constructed more easily, which will in turn allow a manager, maintainer or developer to focus more closely on the details pertinent to their task while maintaining a working mental model of the project as a whole.

The findings of this study have an impact on the techniques used in the case studies reported in Chapters 5 and 6, and Section 7.2. For example, in Thematic Analysis Study 1.2 (see Section 5.5) the open source projects analysed were too large for complete analyses to be conducted, and so partial sections of the projects were used instead, by using only the earliest revisions before the first milestone release. As demonstrated in Thematic Analysis Stage 1.3 (see Section 5.6) the nature of PuTTY changed significantly after the early period of the project. By using a project sampling technique, a more representative subset of revisions could have been selected for study. The heuristics used in the selection would necessarily have to be modified to suit the task but the principle of automated file selection, having been established as a viable technique, would allow for a more representative set of comments to be used in a thematic analysis.

7.3.4 Case Study Conclusions

The overall goal of this research is to identify techniques whereby transactional data from RCS repositories can be used to support project comprehension. File sampling has been identified as a technique which is used to support comprehension by allowing users either to gain a broad overview of a project or to create a "starting point" from which to begin an exploration of a project in more depth. This case study was conducted to evaluate a file sampling technique based on repository analysis rather than static analysis, and research question *RQ 9* was proven to be positive: repository analysis can be used to create an automated file sampling technique. Therefore, file sampling based on repository analysis is a process which can be used to support project comprehension.

7.4 Conclusions

This research is conducted with the overall goal of evaluating techniques by which project comprehension can be supported through repository analysis;

the following research question was identified:

- RQ 1: How can data mining of revision control systems be applied to support project comprehension?

The software visualization case study has shown that the subjects of the study received little benefit from the visualizations, and that there is insufficient evidence to answer either *RQ 7* or *RQ 8*, and thus cannot support the hypothesis that software visualization is a technique by which project comprehension can be supported.

The file sampling case study, however, proved that an automated tool can successfully generate a representative subset of files from a project. This process can be used either to form a "starting point" for exploring a project, or to gain a broad understanding of a project; in either case, file sampling is a useful technique for supporting project comprehension.

Therefore, this chapter provides the following answers to the overall research question, *RQ 1*:

- Software visualization of repository data has not been proven to be a technique which can be used to support project comprehension
- File sampling using repository analysis has been proven to a useful technique for supporting project comprehension

7.5 Summary

This chapter reported a pair of case studies which aimed to explore the ability to support project comprehension through the visualization RCS repository data and by using transactional data to extract a subsection of a project which is representative of the whole.

As is common with visualization tools, the visualization suite provided to student project managers did not all suitably address their needs and they were not widely adopted. The simpler tools, charts and progress reports, were

welcomed by the managers however, reinforcing existing research which shows that visualization tools need to be developed solely around the needs of the target users, and be intuitive and familiar enough to require little or no training to adopt. More positively, due to sharing many traits (if not experience) with industrial practitioners, student managers can be useful substitutes for industrial practitioners in empirical studies of software visualization tools, which can go some way to mitigating the lack of user-centric studies in the field.

`Perceive` was successful in automatically generating representative subsets of projects, more so than experienced assessors, which has applications in a wide range of activities related to project comprehension, such as visualization, assessment or thematic analysis.

Chapter 8 draws together the findings and conclusions of all four case studies and discusses them in the wider context of supporting project comprehension using transactional data from RCS repositories. It then goes on to discuss avenues for future work, enhancements to the software developed, and follow-up studies to provide further evidence to the findings of this research.

Chapter 8

Conclusions and Future Research

8.1 Introduction

Project comprehension encompasses a number of fields and activities, relevant to a range of users, including managers, developers and maintainers. These activities are present in any software development environment, from education to industry. Chapter 1 stated the primary question of this research as follows:

- *RQ 1*: How can data mining of revision control systems be applied to support project comprehension?

To address this question four aspects of project comprehension – project profiling, change prediction, software visualization and project sampling – were identified as areas which might be facilitated, supported or improved through the use of RCS repository analysis. A case study was designed for each of these four areas with the goal of evaluating whether that area could be supported or facilitated using a technique or process based on repository analysis. These case studies were reported in Chapters 5, 6 and 7.

In each case study, a technique was trialled in a sub-study: thematic analysis, history-based change prediction, project management support and

file. The sub-studies each reported the success or failure of the trial; these conclusions were then evaluated in turn by the "parent" case study in the wider context of determining whether or not the process successfully demonstrates that an aspect of project comprehension is supported or improved by the technique under study.

This chapter assesses the outcome of the four case studies by taking their conclusions and evaluating them in the context of the overall goal of this research, that of identifying project comprehension activities which can successfully be supported using techniques based on data mining of revision control system repositories.

The chapter then concludes with a discussion of potential future research, to build on the reported studies and to further explore issues and opportunities raised by the research.

8.2 Case Study 1: Profiling Projects

The ability to examine a project to profile or categorize certain aspects of it has many applications. For example, Hindle et al. examined a set of large revisions (Hindle et al., 2008) to determine their impact on a project, and whether or not the conventional belief that they were harmful was accurate, and discovered that large commits frequently consisted of perfective maintenance activities and frequently concerned the architecture of a project. Thomson and Holcombe (Thomson and Holcombe, 2008) used repository data to profile students' use of RCS tools, classifying common errors.

Such research is important to software engineering, and can be considered the software equivalent of empirical secondary studies. Just as such studies aggregate the findings of a number of primary studies, so profiling projects based on repository data can be used to make broad statements about software development, and have the advantage of being easily repeatable, allowing replication studies to be carried out and enabling additional projects or domains to be added to the body of data.

The following research questions were identified in relation to project profiling:

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data?
- *RQ 3*: Can profiling support project comprehension in student and open source projects?

Chapter 5 reported on a case study in which the process of thematic analysis was applied to a number of open source and student projects in order to address these research questions.

8.2.1 *Research Question 2*

- *RQ 2*: Can the process of thematic analysis be applied to RCS repository data? ***Proven: Yes***

As the literature reviews reported in Chapters 2 and 3 revealed no studies reporting thematic analyses being performed on revision logs, it was not certain whether or not the process, designed primarily for conversational text, could be applied to this data. Studies 1.1, 1.2 and 1.3 reported in Chapter 5 demonstrate that there are no practical barriers to performing such analyses, regardless of the style or content of the text; for example, in addressing research questions *RQTA 2-5* and *RQTA 2-6* (see Study 1.2, sections 5.5.5.5 and 5.5.5.6) it was demonstrated that not only could thematic analysis be performed on both student and open source projects, the process was robust enough to allow comparison between the two domains. Furthermore, despite concerns that training and time requirements might prove prohibitive, only the process of creating the codes was time consuming. As reported in Study 1.3 (see Section 5.6 the categorization of messages to those codes was a rapid process, requiring no more domain knowledge that would be required for any use of the repositories.

8.2.2 *Research Question 3*

- *RQ 3*: Can profiling support project comprehension in student and open source projects? ***Proven: Yes***

The application of thematic analysis to student projects, especially when contrasted with open source projects, provided valuable insight into the nature of student projects, developer behaviour and the collaboration process. For example, in addressing the research questions *RQTA 1-1* and *RQTA 1-5* (see Study 1.1, sections 5.4.4.1 and 5.4.4.5), it was demonstrated that useful information regarding both the development process of students, and students' use of project support tools could be extracted from project comment logs. These results were successfully fed forward to a second cohort, who demonstrated improved use of tools and adherence to best practices.

8.2.3 Summary

There is a great deal of data contained in RCS repository comment logs, even incomplete logs. While a simple reading of these logs can improve the reader's understanding of a project, a formal analysis can provide a much deeper degree of project comprehension, able to quantity essentially qualitative data. The case study reported in Chapter 5 demonstrated that thematic analysis can be a valuable tool in profiling projects; these findings have potential applications in a range of activities, including project management, quality assessment or academic assessment.

In the wider context of this research, each case study seeks to contribute an answer to the overall research question:

- *RQ 1*: How can data mining of revision control systems be applied to support project comprehension?

This case study has demonstrated that not only is thematic analysis a viable technique for profiling a project, the information acquired as a result leads to an improved, deeper understanding of the project being analysed.

Therefore, this case study leads to the conclusion that project profiling in general, and thematic analysis in particular, can be performed using repository analysis to support project comprehension.

8.3 Case Study 2: Change Prediction

As a project evolves, it becomes more and more difficult to accurately predict the impact of a change; a change to the requirements specification or design will require modifications to the implementation, and a change to the implementation will almost certainly require further changes. Predicting the effects of a change is the subject of a great deal of research, as discussed in Section 2.5. Change prediction is primarily used during code modification to reduce the occurrences of missed changes – such as a bug-fixing patch which introduces new bugs – but also has applications in other fields, such as unit testing (Ren et al., 2004).

Research has shown that a code-based, syntactic change prediction technique is not always reliable (Hassan and Holt, 2004; Bieman et al., 2003), leading to the necessity to augment the techniques with other models. The use of historical project data to perform change prediction provides encouraging results (Hassan and Holt, 2004; Zimmermann et al., 2005), and appears to be a viable candidate for supporting existing methods.

The case study described in Chapter 6 builds on existing research to answer the following research questions:

- *RQ 4*: Is history-based change prediction a viable technique?
- *RQ 5*: When does the technique outperform syntax-based methods?
- *RQ 6*: Can project profiling be used to improve history-based change prediction?

8.3.1 *Research Question 4*

- *RQ 4*: Is history-based change prediction a viable technique? ***Proven: Yes***

While use of history-based change prediction has been explored in previous research this study has empirically measured the performance of a history-based change prediction model, `Perceive`, against a range of control models, as described in the Study 2.1 reported in Section 6.4. This benchmarking study found that `Perceive` statistically outperformed the control models in predicting files which would require changing based on a set of existing changes. For example, in Section 6.4.7.2 statistical testing showed that in a use-case designed to mimic a real-world scenario, `Perceive` outperformed both of the control models in terms of precision (0.411 against 0.016 and 0.099) and recall (0.170 against 0.016 and 0.099).

8.3.2 *Research Question 5*

- *RQ 5*: When does the technique outperform syntax-based methods?

Study 2.2, reported in Section 6.5, highlighted a set of cases where a history-based change prediction model can successfully infer a relationship between a pair of files which cannot be found by a static analysis technique, especially between files of different types such as documentation or graphics. Conversely, there are also cases where a history-based technique cannot perform as well as a static technique; for example, there will always be a "training period" for a new file, where insufficient data exist to make predictions based on, or for, that file. This has led to the conclusion that an ideal change prediction model will incorporate elements of both static and history-based techniques.

8.3.3 *Research Question 6*

- *RQ 6*: Can project profiling be used to improve history-based change prediction? ***Proven: Yes***

Study 2.3 (see Section 6.6) showed that it is possible to improve the performance of a history-based change prediction model using data generated by a thematic analysis: adding maintenance activity classifications to the

model increased the performance of `Perceive` by a statistically significant degree on both academic and open source projects. Table 6.16 shows how the improved model, $\texttt{Activity}_2$, has a higher F-Score than the original `Perceive` model on both PuTTY and SEG projects – 27.6% against 26.3% for PuTTY and 28.1% against 27.6% for SEG.

8.3.4 Summary

Change prediction is an important process with a heavy reliance on project comprehension. The less a developer's mental model of a project matches the reality, the more likely they are to fail to fully predict the effects of a change. This research has reinforced existing studies demonstrating the ability of history-based change prediction models to support a developer's project comprehension, and then successfully used thematic analysis data to further improve the performance of the prediction model. Any factor, metric or technique which can provide better change prediction can have a measurable impact on the success of maintenance and development activities.

This case study was designed and conducted to ultimately address the overall research question:

- *RQ 1*: How can data mining of revision control systems be applied to support project comprehension?

By successfully implementing – and then improving – a model of history-based change prediction using RCS repository data, this case study has demonstrated that repository analysis can be used to successfully conduct change prediction, and thus support project comprehension.

8.4 Case Study 3: Software Visualization

As discussed in Section 2.6 the field of software visualization faces a series of problems. Aside from the well-documented technical problems such as scale, usability and performance, there is a documented problem of low

adoption of visualization tools and models, compounded by a lack of empirical studies regarding software visualization tools and techniques. This is in part attributable to a research environment driven primarily by innovation and concept demonstration rather than user-focussed design (Burn et al., 2009). It is true that more mature fields of research have sets of established protocols, designs and data sets which enable experimental research and replication studies, but software visualization is supported by both software engineering and cognitive psychology, two disciplines with strong foundations of empirical research.

Chapter 7 reported a case study which sought to explore whether or not visualization of historical data from RCS repositories could be a useful comprehension tool for student project managers. The following research questions were identified:

- *RQ 7*: Which RCS-based visualization techniques best support student project managers?
- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques?

The following sections discuss how these questions have been answered, and what evidence supports the conclusions.

8.4.1 *Research Question 7*

- *RQ 7*: Which RCS-based visualization techniques best support student project managers? ***Not proven***

In Study 3.1 feedback from the student project managers revealed that the visualization tools saw very low adoption; only the simpler, more familiar tools were used by the managers: the charts and the progress report. Despite being provided with training and documentation the managers did not perceive a benefit to expending time and effort on learning to use new tools; however, the charts and the reports were more familiar; their utility was more apparent and so they were much more readily accepted by the managers.

This research has led to the conclusion that when using visualizations of RCS data to support project comprehension the key factors are familiarity, simplicity and clarity of function – the end-users must be able to use the system rapidly with little to no training and also immediately understand *why* they should do so, and what benefit the system will bring them.

8.4.2 *Research Question 8*

- *RQ 8*: Are student project managers suitable subjects for evaluating software visualization techniques? ***Not proven***

The software visualization case study has demonstrated that while the needs of project managers in industrial and educational domains differ in a number of ways many of the motivations and behaviours exhibited by student managers are the same as those of industrial practitioners: a resistance to adopting new tools or processes, and the need for tools to directly, immediately and clearly address a real problem.

Recognizing these similarities, there arises the fact that students have the potential to form a valuable body of subjects for empirical studies of software visualization tools and techniques. It can be difficult to find industrial practitioners to use as subjects when designing or evaluating visualization tools, whereas students are much more readily employed as test subjects. When designing a visualization tool, students can be used as a "first round" evaluation of usability, utility and uptake, with successful studies leading to industrial case studies which would have a greater impact with industrial practitioners.

8.4.3 Summary

Software visualization is a powerful concept, able to provide and support project comprehension in a wide range of manners. However, the lack of empirical research of visualization has led to a very low uptake. This research has led to the conclusion that the visualization of RCS repository data has

some use to project managers, more involvement of the end-users is required at each stage if the visualizations are to be adopted. Moreover, regardless of who the end-users are, whether industrial practitioners or students, the same mentality is in evidence; students can therefore act as useful subjects when designing and evaluating visualization tools. By doing so, the body of empirical knowledge will grow, and software visualization research may be able to move more towards a user- and evidence-centric design process.

The primary goal of this case study was to address the overall research question of this research:

- *RQ 1*: How can data mining of revision control systems be applied to support project comprehension?

The case study was unable to establish sufficient weight of evidence to determine whether or not software visualization tools are a useful technique for supporting project comprehension. The case study has identified a potential avenue of future research, whereby students are used as substitutes for industry practitioners, and empirical studies to design and evaluate techniques for visualizing project change and repository data have been planned as a follow-up to this case study.

8.5 Case Study 4: Sampling

Considering the scale of modern software projects which can be comprised of millions of lines of code, thousands of source code files and even more numerous assets – such as documentation, data or graphics – it is becoming more and more important to be able to identify a subset of files which is representative of the whole project. For example, a manager wishing to view metrics of a project might have little difficulty with simpler metrics such as lines of code, but calculating more complex, computationally intensive metrics for a large project may be infeasible. Similarly, as described in Section 2.6, software visualization techniques do not always scale well (further

demonstrated by FlowVis in Section 7.2.2.3.4); a common solution is to simply visualize a section of the project, but some visualizations are designed to give an abstracted overview of a project. By being able to select a representative sample of a project, rather than – for example – a single module, it would be possible for the visualization to provide an accurate overview of the whole project.

Section 7.3 reported a case study in which a technique for generating representative subsets of a project was evaluated against a group of human experts, in order to address the following research question:

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling?

8.5.0.1 *Research Question 9*

- *RQ 9*: Can the data contained within RCS repositories be used to automate file sampling? ***Proven: Yes***

As reported in Study 4.1, in the context of selecting files from a group project for assessment, the experts were inconsistent between themselves, showing little overlap in their selections. They also varied in the strategies they used to make their selections, as well as in the degree to which they successfully applied those strategies.

A coverage metric based on the number of revisions and users involved with a set of files was used to empirically assess the coverage samples produced by the experts and by `Perceive`. `Perceive` statistically outperformed the experts, selecting sets of files with greater coverage than those of the experts.

8.5.1 Summary

While the level of project comprehension gained from a subset of a project will obviously be constrained in depth and completeness, it allows for a broader overview or high-level mental model to be constructed more easily. The

sample of files can then act as a starting point from which to explore the project more completely.

The demonstrated success of project sampling has applications in a range of project comprehension activities, including those explored in this research. For example, in Thematic Analysis Study 1.2 (see Section 5.5) the open source projects analysed were too large for complete analyses to be conducted, and so partial sections of the projects were used instead, by using only the earliest revisions before the first milestone release. Rather than naively using a contiguous set of revisions for analysis, a representative sample of the project could be used as the subject, which would allow the results of the thematic analysis to be generalized to the entire project with greater confidence.

Additionally, some of the visualization techniques used in Section 7.2 did not scale well as the number of files increased. Sampling could be used to provide a high-level visualization, which could then be focussed on individual areas of the project to facilitate deeper comprehension.

Finally, history-based change prediction could be made more robust with the use of sampling. By inverting the process, *unrepresentative* files and revisions could be identified and allocated less weight, potentially improving the performance of the prediction technique further still.

In the wider context of this research, this case study is designed to answer the overall research question:

- *RQ 1*: How can data mining of revision control systems be applied to support project comprehension?

By successfully implementing and evaluating a system to select a subset of files from a project for assessment, this case study has demonstrated that RCS repository analysis can be used to perform representative file selection, a process with applications in a number of activities related to project comprehension.

8.6 Future Research

As discussed in each case study, there is significant scope for further developing the research presented here. As with all empirical work there is a need for replication studies to provide a greater weight of evidence to the conclusions drawn, to reveal experimental flaws, anomalous results or to expand the degree to which the conclusions can be generalized.

The thematic analysis case study has several possibilities for extension. Primarily, a greater range of projects need to be analysed to add to the body of data generated in Chapter 5. In cases where an entire project cannot be analysed for reasons of scale, the sampling process discussed in Section 7.3 can be employed to generate more manageable sets of revisions to analyse. A second avenue would be to analyse projects from different domains, such as commercial projects, which would further expand the range of projects to which the results can be applied, and allow a deeper insight into how various project types are conducted and structured. In an academic context, a thematic analysis could form part of the assessment or management process, and feedback from students and assessors could be used to evaluate the effectiveness of the technique in a situation where its impact can be easily identified.

The main conclusion of the visualization case study, that student project managers can be useful substitutes for industrial practitioners, needs to be tested empirically. A study must be performed in which a visualization tool is evaluated by both a body of students and by industrial practitioners. Such a study would enable researchers to identify what aspects of software visualization design and evaluation can be tested with students, and which must be tested with industrial practitioners.

The project sampling quasi-experiment should be replicated to support or counter the findings of the study. While the experimental design was strong, it requires a greater number of projects and experts from a wider range of domains to more accurately evaluate the performance of the models and the consistency of the experts. This will also allow the coverage metric to be

evaluated more thoroughly to ensure that it performs as required.

Finally, the history-based change prediction case study in Chapter 6 has perhaps the greatest potential for further research. As discussed in Section 6.9 a more direct comparison between the performance of syntax-based techniques and history-based techniques is required, which will allow a deeper understanding of how the two models can complement each other.

8.7 *RQ 1*: How Can Data Mining of Revision Control Systems be Applied to Support Project Comprehension?

The overarching goal of this research has been to identify ways in which analysis of transactional, language-agnostic data from revision control systems can be used to support project comprehension. The four case studies reported here have each sought to address this goal by exploring different aspects or applications of project comprehension and evaluating the degree to which the repository analysis has been successful in improving comprehension.

The thematic analysis, which has not been previously applied to revision comment logs, has been shown to be a powerful and viable tool for extracting information from repositories, even incomplete ones. History-based change prediction has been shown to be a viable technique, supporting existing studies; this research has also demonstrated that maintenance activity type data can be used to improve the performance of history-based change prediction to a statistically significant degree, contributing to a growing body of methods of improving prediction discovered by the research community. Visualization of repository data has proven less successful, as student project managers are unlikely to adopt new tools without a clear benefit of doing so. Finally, an automated process of sampling a project to create representative sets of files has been successfully demonstrated in an empirical experiment, with implications on a broad range of project comprehension activities.

In conclusion, there are a number of ways by which repository analysis can be used to support project comprehension. This research has evaluated four techniques, three of which have been proven to support project comprehension and related activities; each of the case studies has implications in the field of project comprehension, especially change prediction where any technique to improve the performance of a prediction model can have an appreciable impact on the cost of software development. The overall goal of identifying techniques based on repository analysis which can be used to support project comprehension has been met; three of the four selected techniques have been proven to successfully support program comprehension.

Appendix A

Structured Literature Review References

Title	Reference	System	Domain
A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction	Wu et al. 2004	CVS	Unspecified
A tool for mining defect-tracking systems to predict fault-prone files	Ostrand and Weyuker 2004	Unspecified	Industrial
An Integrated Approach for Studying Architectural Evolution	Tu and Godfrey 2002	Various	FOSS
Analysis of signature change patterns	Kim et al. 2005	Various	FOSS
Applying Social Network Analysis to the Information in CVS Repositories	Lopez-Fernandez et al. 2004	CVS	FOSS
Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings	Lessmann et al. 2008	NASA MDP	Unspecified
Chianti: a tool for change impact analysis of Java programs	Ren et al. 2004	CVS	FOSS
Comparing Approaches to Mining Source Code for Call-Usage Patterns	Kagdi et al. 2007a	Static	FOSS
Correctness of data mined from CVS	Thomson and Holcombe 2008	CVS	Students
CVSscan: Visualization of Code Evolution	Voinea et al. 2005	CVS	FOSS
Detecting and Visualizing Refactorings from Software Archives	Gorg and Weisgerber 2005	CVS	FOSS

Continued on Next Page...

Title	Reference	System	Domain
Detecting Move Operations in Versioning Information	Van Rysselberghe et al. 2006	CVS	Industrial
Detection of Logical Coupling Based on Product Release History	Gall et al. 1998	PRDB	Industrial
Determining Implementation Expertise from Bug Reports	Anvik and Murphy 2007	CVS, Bugzilla	FOSS
Empirical project monitor: a tool for mining multiple project data	Ohira et al. 2004	Multiple	All
Expertise identification and visualization from CVS	Alonso et al. 2008	CVS	FOSS
Exploring evolutionary coupling in Eclipse	Weissgerber et al. 2005	CVS	N/A
Four Interesting Ways in Which History Can Teach Us About Software	Godfrey et al. 2004	CVS	FOSS
Hipikat: Recommending Pertinent Software Development Artifacts	Cubranic and Murphy 2003	Multiple	Various
How Long Will It Take to Fix This Bug?	Weiss et al. 2007	Jira	FOSS
If your version control system could talk	Ball et al. 1997	ECMS	Industrial
Improving change descriptions with change contexts	Parnin and Görg 2008	Unspecified	Unspecified
Learning by doing:Introducing version control as a way to manage student assignments	Reid and Wilson 2005	CVS	Students
Measuring Developer Contribution from Software Repository Data	Gousios et al. 2008	Multiple	N/A

Continued on Next Page...

Title	Reference	System	Domain
Mining CVS Repositories to Understand Open-Source Project Developer Roles	Yu and Ramaswamy 2007	CVS	FOSS
Mining CVS repositories, the softchange experience	German 2004	CVS	FOSS
Mining repositories to assist in project planning and resource allocation	Menzies et al. 2004	NASA MDP	Unspecified
Mining student CVS repositories for performance indicators	Mierle et al. 2005	CVS	Students
Mining the software change repository of a legacy telephony system	Shirabad et al. 2004	SMS	Industrial
Mining Version Control Systems for FACs (Frequently Applied Changes)	Rysselberghe and Demeyer 2004	N/A	N/A
Mining Version Histories to Guide Software Changes	Zimmermann et al. 2005	CVS	FOSS
Monitoring the Evolution of an OO System with Metrics: An Experience from the Stock Market Software Domain	Girard et al. 2004	Snapshots	Industrial
Predicting Change Propagation in Software Systems	Hassan and Holt 2004	Unspecified	FOSS
Predicting Fault Incidence Using Software Change History	Graves et al. 2000	SCCS	Industrial
Preprocessing CVS data for fine-grained analysis	Zimmermann and Weissgerber 2004	CVS	FOSS
Research Infrastructure for Empirical Science of F/OSS	Gasser et al. 2004	N/A	N/A
Software Evolution Observations Based on Product Release History	Gall et al. 1997	PRDB	Industrial

Continued on Next Page...

Title	Reference	System	Domain
Studying Software Evolution Using Clone Detection	Rysselberghe and Demeyer 2003	Snapshots	N/A
Text is Software Too	Dekhtyar et al. 2004	N/A	N/A
The perils and pitfalls of mining SourceForge	Howison and Crowston 2004	N/A	FOSS
Towards a Theoretical Model for Software Growth	Herraiz et al. 2007	Static	FOSS
Understanding Change-Proneness in OO Software through Visualization	Bieman et al. 2003	N/A	Industrial
Using CVS Historical Information to Understand How Students Develop Software	Liu et al. 2004	CVS	Students
Using version control to observe student software development processes	Glassy 2005	SubVersion	Students
Visual Data Mining in Software Archives to Detect How Developers Work Together	Weissgerber et al. 2007	CVS	FOSS
VRCS: integrating Version Control and Module Management using Interactive 3D graphics	Koike and Chu 1997	RCS/SCSS	N/A
What do large commits tell us?: a taxonomical study of large commits	Hindle et al. 2008	Unspecified	FOSS

Table A.1: Overview of the results of the structured literature review

Bibliography

Bram Adams, Zhen Ming Jiang, and Ahmed E. Hassan. Identifying crosscutting concerns using historical code changes. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 305–314. ACM, 2010. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1806799.1806846.

Samuel Ajila. Software maintenance: An approach to impact analysis of objects change. *Softw. Pract. Exper.*, 25(10):1155–1181, 1995. ISSN 0038-0644.

Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 182–191, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3176-2. doi: http://dx.doi.org/10.1109/ICPC.2008.24.

Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from cvs. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370780.

Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. Software visualization tools for component reuse. In *Second Workshop on Method*

Engineering for Object-Oriented and Component-Based Development at OOPSLA 2004, 2004.

John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.7.

A Bachmann and A Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 62–71. IEEE, 2010. ISBN 978-1-4244-6802-7. doi: http://dx.doi.org/10.1109/MSR.2010.5463286.

T Ball, JM Kim, AA Porter, and HP Siy. If your version control system could talk. In *Workshop on Process Modeling and Empirical Studies of Software Engineering, ICSE*, 1997.

Françoise Balmas. Displaying dependence graphs: A hierarchical approach. *Journal of Software Maintenance and Evolution*, 16:151–185, 2003. ISSN 1532-060X.

Todd Barlow and Padraic Neville. A comparison of 2-d visualizations of hierarchies. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, pages 131–138, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1342-5.

Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

Jon Beck. Using the cvs version management system in a software engineering course. *Journal of Computing Sciences in Colleges*, 20(6):57–65, 2003.

Keith H. Bennett and Vaclav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-253-0.

James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. Understanding change-proneness in oo software through visualization. *International Conference on Program Comprehension*, 0:44, 2003. ISSN 1092-8138. doi: http://doi.ieeecomputersociety.org/10.1109/WPC.2003.1199188.

Sue Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(4):263–279, 2001. ISSN 1532-060X. doi: 10.1002/smr.233.

Alan Blackwell, Kirsten Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15: 95–113, 2001. ISSN 0269-2821.

Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qual Res Psychol*, pages 77–101, 2006.

Greg Breinholt and Christoph Schierz. Algorithm 781: generating hilbert's space-filling curve by recursion. *ACM Trans. Math. Softw.*, 24(2):184–189, 1998. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/290200.290219.

BS ISO/IEC 14764:2006. Bs iso/iec 14764:2006, 2006. Software engineering. Software life cycle processes. Maintenance.

David Budgen, Andrew Burn, Rialette Pretorius, Pearl Brereton, and Barbara Kitchenham. Empirical evidence about the uml: A systematic literature review. *Accepted for publication in Software: Practice and Experience*, 2010. doi: 10.1002/spe.1009.

E.L. Burd and S.A. Drummond. Forging planned inter year co-operation through a peer mentor system for group work projects. In *Proceedings*

of the 3rd Annual Conference LTSN Information and Computer Sciences, 2006.

L. Burd and M. Munro. Research institute for software evolution, 2003. http://www.dur.ac.uk/RISE/.

Andrew Burn. Thematic analysis of group software project change-logs. In *Proceedings of the 9th HEA ICS Annual Conference*, 2008.

Andrew Burn. Thematic analysis of group software project change logs: An expanded study. *ITALICS*, 8, 2009. ISSN 1473-7507.

Andy Burn, David Budgen, Malcolm Munro, and Thomas Ward. Software visualization: The empirical landscape. In *Submitted to ICSE 2009*, 2009.

K. Chen, S.R. Schach, L. Yu, J. Offutt, and G.Z. Heller. Open-source change logs. *Empirical Software Engineering*, 9(3):197–210, 2004.

Ben Collins-Sussman. The subversion project: buiding a better cvs. *Linux J.*, 2002:3, 2002. ISSN 1075-3583.

Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. *Software Engineering, International Conference on*, 0:408, 2003. ISSN 0270-5257. doi: http://doi.ieeecomputersociety.org/10.1109/ICSE.2003.1201219.

S. Cummins, Burd L., and A. Hatch. Improving student engagement with feedback: Using feedback tagging for programming assignments. In *Proceedings of the Higher Education Academy Subject Centre for Information and Computer Science's 2010 Annual Conference*, 2010.

Susan Dart, Alan M. Christie, and Alan W Brown. A case study in software maintenance. Technical report, Software Engineering Institute, Carnegie Mellon University, 1993.

A. Dekhtyar, J. Huffman Hayes, and T. Menzies. Text is software too. In *Proceedings of the 1st International Workshop on Mining of Software Repositories*, pages 22–27, Edinburgh, Scotland, 2004.

S.A. Drummond and M Devlin. Software engineering students' cross-site collaboration: An experience report. In *Proceedings of the 7th HEA ICS Annual Conference*, 2006.

Tore Dybå, Barbara Kitchenham, and Magne Jørgensen. Evidence-based software engineering for practitioners. *IEEE Software*, 22(1):58–65, 2005.

Software Engineering Economics. *Barry Boehm*. Prentice Hall, 1981. ISBN 0138221227.

Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992. ISSN 0098-5589.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/24039.24041.

Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

Milton Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92, 1940. ISSN 00034851. doi: 10.1214/aoms/1177731944.

Harald Gall, Mehdi Jazayeri, Rene Klosch, and Georg Trausmuth. Software evolution observations based on product release history. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 160–170. IEEE Computer Society, 1997. ISBN 0-8186-8013-X.

Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 190–198, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7.

Keith Gallagher. Visual impact analysis. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 52–58. IEEE Computer Society, 1996. ISBN 0-8186-7677-9.

Emden Gansner, Elftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, 2002.

Les Gasser, Gabriel Ripoche, and Robert Sandusky. Research infrastructure for empirical science of f/oss. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 12–16, Edinburgh, Scotland, UK, 2004.

Daniel M. German. Mining cvs repositories, the softchange experience. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 17–21, Edinburgh, Scotland, UK, 2004.

Jean-Francois Girard, Martin Verlage, and Dharmalingam Ganesan. Monitoring the evolution of an oo system with metrics: An experience from the stock market software domain. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 360–367. IEEE Computer Society, 2004. ISBN 0-7695-2213-0.

Louise Glassy. Using version control to observe student software development processes. In *Journal of Computing Sciences in Colleges*, volume 21, pages 99–106, 2005.

Michael Godfrey, Xinyi Dong, Cory Kapser, and Lijie Zou. Four interesting ways in which history can teach us about software. In *In Proceedings of the International Workshop on Mining Software Repositories*, 2004.

Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2254-8. doi: http://dx.doi.org/10.1109/WPC.2005.18.

Georgios Gousios and Diomidis Spinellis. A platform for software engineering research. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 31–40, 2009.

Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *Proceedings of the Fifth International Workshop on Mining Software Repositories*, pages 129–132. Association for Computing Machinery, 2008. ISBN 987-1-60558-079-1. doi: 10.1145/1370750.1370781.

Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7): 653–661, 2000. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.859533.

Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0.

Val Henson and Jeff Garzik. Bitkeeper for kernel developers. In *Proceedings of the 2002 Linux Symposium*, 2002.

Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.31.

Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370773.

Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM. ISBN 0-89791-504-6. doi: http://doi.acm.org/10.1145/143062.143156.

martin Host, Björn Regnell, and Wohlin Claes. Using students as subjects - a comparative study of students and professionals in lead-time impact assessment. *ESE - Empirical Software Engineering*, 5(3):201–214, 2000.

James Howison and Kevin Crowston. The perils and pitfalls of mining sourceforge. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 7–11, Edinburgh, Scotland, UK, 2004.

Christopher Hundhausen. Exploring the potential for conversation analysis in the evaluation of interactive algorithm visualization systems. Master's thesis, Department of Computer and Information Science, University of Oregon, Eugene, OR, 1993.

Christopher Hundhausen, Sarah Douglas, and John Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002. ISSN 1045-926X.

Matthew Hutchins and Keith Gallagher. Improving visual impact analysis. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 294–303, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7.

IEEE Standard 610.12-1999. Ieee standard 610.12-1999, 1999. Standard for Software Maintenance.

M. Jørgensen. An empirical study of software maintenance tasks. *Journal of Software Maintenance: Research and Practice*, 7(1):27–48, 2006.

Magne Jørgensen and Dag I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance*, 14(2):123–146, 2002. ISSN 1040-550X.

Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 20, Washington, DC, USA, 2007a. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.3.

Huzefa Kagdi, Jonathan I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 145–154, Washington, DC, USA, 2007b. IEEE Computer Society. ISBN 0-7695-2860-0. doi: http://dx.doi.org/10.1109/ICPC.2007.28.

S. Kim, E.J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, pages 4–11. IEEE Computer Society, 1999. ISBN 0-7695-0179-6.

Hideki Koike and Hui-Chu Chu. Vrcs: Integrating version control and module management using interactive 3d graphics. In *VL '97: Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, page 168. IEEE Computer Society, 1997. ISBN 0-8186-8144-6.

Jens Krinke. Visualization of program dependence and slices. In *Proceedings of International Conference on Software Maintenance*, pages 168–177, 2004.

Michele Lanza. Program visualization support for highly iterative development environments. In *Proceedings of the 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 62–67. IEEE Computer Society, 2003.

Meir Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124. Springer-Verlag, 1996. ISBN 3-540-61771-X.

Meir M. Lehman and Juan F. Ramil. Effort estimation from change records of evolving software. In *Proceedings of the ICSE*, pages 777–777. ACM Press, 2000.

Meir M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001. ISSN 1022-7091.

Josh Lerner and Jean Tirole. Some simple economics of open source. *The Journal of Industrial Economics*, 50(2):197–234, 2002. ISSN 00221821.

Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Software Eng.*, 34(4): 485–496, 2008.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978. ISSN 0001-0782.

Ying Liu, Eleni Stroulia, Kenny Wong, and Daniel German. Using cvs historical information to understand how students develop software. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 32–36. IEEE Computer Society, 2004.

S. Lohr and J. Markoff. Windows is so slow, but why. The New York Times, 2006.

Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Applying social network analysis to the information in cvs repositories. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 101–105, Edinburgh, Scotland, UK, 2004.

Robert Love. Introducing the 2.6 kernel. *Linux J.*, 2003(109):2, 2003. ISSN 1075-3583.

Matt Mackall. Towards a better scm: Revlog and mercurial. In *Proceedings of the 2006 Linux Symposium*, 2006.

Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software Visualization*, pages 27–36. ACM Press, 2003. ISBN 1-58113-642-0.

Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 83–86. ACM Press, 2001. ISBN 1-58113-508-4.

T. Menzies, J.S. Di Stefano, C. Cunanan, and R. Chapman. Mining repositories to assist in project planning and resource allocation. *IEE Seminar Digests*, 2004(917):75–79, 2004. doi: 10.1049/ic:20040480.

Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. Mining student cvs repositories for performance indicators. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6. doi: http://doi.acm.org/10.1145/1083142.1083150.

Sougata Mukherjea and John Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction*, 1:215–244, 1994.

Paul Mulholland. Evaluating program visualisation systems: An information-based methodology. Technical report, Human Cognition Research Laboratory, Open University, 1993.

M. Ohira, R. Yokomori, M. Sakai, K. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: a tool for mining multiple project data. *IEE Seminar Digests*, 2004(917):42–46, 2004. doi: 10.1049/ic:20040474.

T.J. Ostrand and E.J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. *IEE Seminar Digests*, 2004(917):85–89, 2004. doi: 10.1049/ic:20040482.

Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 80. IEEE Computer Society, 2003. ISBN 0-7695-2027-8.

Harkirat Padda, Ahmed Seffah, and Sudhir Mudur. Investigating the comprehension support for effective visualisation tools: a case study. In *2d International Conference on Advances in Computer-Human Interfaces*, pages 283–288. IEEE Computer Society Press, 2009.

Chris Parnin and Carsten Görg. Improving change descriptions with change contexts. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370765.

Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. In *CASCON '91: Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

Marian Petre, Alan Blackwell, and Thomas Green. Cognitive questions in software visualization. In Marc H. Brown John Stasko, John Domingue

and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 453–480. M.I.T. Press, 1998. ISBN 0-262-19395-7 (hardcover).

Blaine Price, Ian Small, and Ronald Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

Blaine Price, Ronald Baecker, and Ian Small. An introduction to software visualization. In Marc H. Brown John Stasko, John Domingue and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 3–27. M.I.T. Press, 1998. ISBN 0-262-19395-7 (hardcover).

Vaclav Rajlich. A model and a tool for change propagation in software. *SIGSOFT Softw. Eng. Notes*, 25(1):72, 2000. ISSN 0163-5948.

Karen L. Reid and Gregory V. Wilson. Learning by doing: introducing version control as a way to manage student assignments. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 272–276, New York, NY, USA, 2005. ACM. ISBN 1-58113-997-7. doi: http://doi.acm.org/10.1145/1047344.1047441.

Steven Reiss. Consistent software evolution, 2001. White Paper.

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-9. doi: http://doi.acm.org/10.1145/1028976.1029012.

G Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *Proceedings of the 7th IEEE Working Conference on Mining Software*

Repositories, pages 171–180. IEEE, 2010. ISBN 978-1-4244-6802-7. doi: http://dx.doi.org/10.1109/MSR.2010.5463348.

Sebastian Rönnau, Jan Scheffczyk, and Uwe M. Borghoff. Towards xml version control of office documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 10–19, New York, NY, USA, 2005. ACM. ISBN 1-59593-240-2. doi: http://doi.acm.org/10.1145/1096601.1096606.

Filip Van Rysselberghe and Serge Demeyer. Studying software evolution using clone detection. In *Workshop on Object-Oriented Reengineering*, pages 71–75, 2003.

Filip Van Rysselberghe and Serge Demeyer. Mining version control systems for facs (frequently applied changes). In *Proceedings of the International Workshop on Mining Software Repositories*, pages 48–52. IEEE Computer Society, 2004.

H. Sharp, Y. Rogers, and J. Preece. *Interaction Design: Beyond Human-computer Interaction*, chapter 13. John Wiley & Sons, 2002.

J.S. Shirabad, T.C. Lethbridge, and S. Matwin. Mining the software change repository of a legacy telephony system. *IEE Seminar Digests*, 2004(917): 53–57, 2004. doi: 10.1049/ic:20040476.

Ben Shneiderman. Control flow and data structure documentation: Two experiments. *Commun. ACM*, 25(1):55–63, 1982. ISSN 0001-0782.

Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20(6):373–381, 1977. ISSN 0001-0782.

Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *Software reusability: vol. 2, applications and experience*, 2:235–267, 1989.

M.A.D. Storey, FD Fracchia, and HA M
"uller. Cognitive design elements to support the construction of a mental model during software exploration. *The Journal of Systems & Software*, 44 (3):171–185, 1999.

Margaret-Anne Storey, Kenny Wong, and Hausi Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183–207, 2000.

Travis Swicegood. *Pragmatic Version Control Using Git*. Pragmatic Bookshelf, 2008. ISBN 1934356158, 9781934356159.

Christopher Thomson and Mike Holcombe. Correctness of data mined from cvs. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 117–120, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370777.

Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 127–136. IEEE Computer Society, 2002. ISBN 0-7695-1495-2.

Filip Van Rysselberghe, Matthias Rieger, and Serge Demeyer. Detecting move operations in versioning information. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 271–278, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2536-9.

Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: Visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software visualization*, pages 47–56. ACM Press, 2005. ISBN 1-59593-073-6.

Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.13.

Peter Weissgerber, Leo von Klenze, Michael Burch, and Stephan Diehl. Exploring evolutionary coupling in eclipse. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 31–34, New York, NY, USA, 2005. ACM. ISBN 1-59593-342-5. doi: http://doi.acm.org/10.1145/1117696.1117703.

Peter Weissgerber, Mathias Pohl, and Michael Burch. Visual data mining in software archives to detect how developers work together. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 9, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.34.

Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 90–99. IEEE Computer Society, 2004. ISBN 0-7695-2243-2.

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005. ISSN 0163-5948.

Peter Young. *Visualising Software in Cyberspace*. PhD thesis, University of Durham, 1999.

Liguo Yu and Srini Ramaswamy. Mining cvs repositories to understand open-source project developer roles. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 8,

Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.19.

Yu Zhou, Michael Würsch, Emanuel Giger, Harald C. Gall, and Jian Lü. A bayesian network based approach for change coupling prediction. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3429-9. doi: http://dx.doi.org/10.1109/WCRE.2008.39.

Thomas Zimmermann and Peter Weissgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 2–6. IEEE Computer Society, 2004.

Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005. ISSN 0098-5589.