Mining Software Repositories to Assist Developers and Support Managers

by

Ahmed E. Hassan

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2004

©Ahmed E. Hassan 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis explores mining the evolutionary history of a software system to support software developers and managers in their endeavors to build and maintain complex software systems.

We introduce the idea of evolutionary extractors which are specialized extractors that can recover the history of software projects from software repositories, such as source control systems. The challenges faced in building C-REX, an evolutionary extractor for the C programming language, are discussed. We examine the use of source control systems in industry and the quality of the recovered C-REX data through a survey of several software practitioners.

Using the data recovered by C-REX, we develop several approaches and techniques to assist developers and managers in their activities.

We propose *Source Sticky Notes* to assist developers in understanding legacy software systems by attaching historical information to the dependency graph. We present the *Development Replay* approach to estimate the benefits of adopting new software maintenance tools by reenacting the development history.

We propose the *Top Ten List* which assists managers in allocating testing resources to the subsystems that are most susceptible to have faults. To assist managers in improving the quality of their projects, we present a complexity metric which quantifies the complexity of the changes to the code instead of quantifying the complexity of the source code itself.

All presented approaches are validated empirically using data from several large open source systems.

The presented work highlights the benefits of transforming software repositories from static record keeping repositories to active repositories used by researchers to gain empirically based understanding of software development, and by software practitioners to predict, plan and understand various aspects of their project.

Acknowledgements

This thesis would not have been possible without the support of many exceptional people to whom I am grateful.

I am greatly indebted to my supervisor Prof. Richard C. Holt for his support and guidance throughout the years. Ric gave me the freedom to pursue research on many interesting topics while providing encouraging words and advice when needed.

Ric has taught me many valuable lessons about being a good researcher and a caring teacher. I sincerely appreciate his advice concerning my presentations and writings. His counsel has been invaluable in making my work readable and my presentations entertaining. Through out my research, Ric has continued to challenge my findings and question my results as we traveled the world presenting the work in this thesis at several international venues. Thanks Ric for making this a very enjoyable and educational experience! It saddens me that the journey has come to an end. Thanks for being a great teacher, a friendly proponent and a challenging opponent.

I appreciate the time and effort that Prof. Jo Atlee, Prof. Charlie Clark, and Prof. Kostas Kontogiannis put into reading my thesis and their valuable comments. I am grateful to Prof. Dewayne Perry for taking time out of his busy schedule to act as the external for my thesis. His insight and knowledge about prior research has been extremely valuable in strengthening my work, especially in spots where I lacked the hard evidence to support my intuition.

I would like to thank Prof. Michael W. Godfrey for always being like a big brother in offering his views and advice on several aspects of academic life. I also would like to thank Mary McColl for her assistance with various administrative details throughout my degree. I appreciate the lively and engaging discussions with many current and previous students of Ric and members of SWAG, in particular, Prof. Susan Sim, Prof. Bil Tzerpos, and Jingwei Wu (with whom I collaborated on many research papers).

The work in this thesis uses several open source repositories. I gratefully acknowledge the significant contributions of members of the Open

Source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted me in understanding and acquiring these valuable repositories.

I am very fortunate to work with many great people at Research In Motion. I would like to thank them for their friendship and their willingness to listen to a crazy scientist who disturbed them with his ideas and thoughts during their peaceful breaks. In particular, I would like to thank Vi Thuan Banh, Denny Chiu, James Godfrey, and Sean (J. F.) Wilson. Many of the ideas in this thesis, were developed by monitoring some of their work habits (*e.g.* the source sticky notes approach). I am also grateful for Vi Thuan for proofreading parts of this thesis.

Throughout this journey, I was lucky to have many good friends who have always been willing to discuss my research ideas and to disagree with me. Thankfully, they still remain my friends at the end of the journey even though sometimes I was too busy to show my appreciation. I would like to thank Mohammed Abouzour, Andrew Hunter, Prof. Karim Karim and Stephen Sheeler. Thanks guys for always telling me when things just did not make sense, you guys made this journey fun and endurable!

Special thanks to my uncle (Mamdouh Bekhit) for his continuous and caring support. I am also grateful to my family for providing an environment where I could escape in order to relax and regain my strength and sanity.

The work in this thesis would not have been possible if it were not for a truly great and extremely patient person:

> *My mom*, to whom I dedicate this thesis.
> *Thanks Mom!*

Ahmed E. Hassan

on February 2005

To My Mom

Related Publications

The following is a list of our publications that are on the topic of mining software repositories:

1. Exploring Software Evolution Using Spectrographs, Jingwei Wu, Ahmed E. Hassan and Richard C. Holt, Proceedings of WCRE 2004: Working Conference on Reverse Engineering, Delft, 2004

2. Predicting Change Propagation in Software Systems, Ahmed E. Hassan and Richard C. Holt, Proceedings of ICSM 2004: International Conference on Software Maintenance, Chicago, Illinois, USA, September 11-17, 2004.

3. Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution, Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan and Richard C. Holt, Proceedings of IWPSE 2004: International Workshop on Principles of Software Evolution, Kyoto, Japan, September 6-7, 2004.

4. Studying The Evolution of Software Systems Using Evolutionary Code Extractors, Ahmed E. Hassan and Richard C. Holt, Draft, Proceedings of IWPSE 2004: International Workshop on Principles of Software Evolution, Kyoto, Japan, September 6-7, 2004.

5. Using Development History Sticky Notes to Understand Software Architecture, Ahmed E. Hassan and Richard C. Holt, Proceedings of IWPC 2004: International Workshop on Program Comprehension, Bari, Italy, June 24-26, 2004.

6. MSR 2004: The International Workshop on Mining Software Repositories, Ahmed E. Hassan, Richard C. Holt and Audris Mockus, Proceedings of ICSE 2004: International Conference on Software Engineering, Scotland, UK, May 23-28, 2004.Workshop Website: http://msr.uwaterloo.ca

7. Studying The Chaos of Code Development, Ahmed E. Hassan and Richard C. Holt, Proceedings of WCRE 2003: Working Conference on Reverse Engineering, Victoria, British Columbia, Canada, November 13-16, 2003.

8. The Chaos of Software Development, Ahmed E. Hassan and Richard C. Holt, Proceedings of IWPSE 2003: International Workshop on Principles of Software Evolution, Helsinki, Finland, September 1-2, 2003.

Contents

1 Introduction **1**

1.0.1 Prior Research 2
1.0.2 Personal Experience 3
1.0.3 The Open Source Phenomena 4
1.1 Research Hypothesis 5
1.2 Thesis Organization 6
1.3 Thesis Overview 7
1.3.1 Part I: Extracting Information From Software Repositories 7
1.3.2 Part II: Using Software Repositories to Support Developers 10
1.3.3 Part II: Using Software Repositories to Support Managers 11
1.4 Thesis Contributions 13

Part I: Extracting Information From Software Repositories **17**

2 Studying The Evolution of Software Systems Using Evolutionary Code Extractors **21**

2.1 Introduction 22
2.1.1 Organization Of Chapter 23
2.2 Describing Source Code Evolution 24
2.3 The Dimensions Of Source Code Evolution 26

2.3.1 Frequency of Snapshots 27
2.3.2 Data Source 28
2.3.3 The Characteristics of the Source Code 29
2.3.4 Level of Detail 31
2.4 Challenges And Complexity 33
2.4.1 Robustness and Scalability 33
2.4.2 Accuracy 34
2.4.3 The Changing and Unstable Nature of Source Code . 35
2.4.4 Development Time 35
2.5 Previous Work 36
2.6 Conclusion 37

3 C-REX: An Evolutionary Code Extractor for C 39
3.1 Introduction 39
3.1.1 Organization of Chapter 42
3.2 Evolutionary Code Extractors 42
3.2.1 Frequency of Snapshots 44
3.2.2 Data Source 45
3.2.3 The Characteristics of Code 45
3.2.4 Level of Detail 46
3.3 Challenges In Developing C-REX 47
3.3.1 Robustness and Scalability of the Extractor 48
3.3.2 Accuracy of the Extracted Information 48
3.3.3 The Changing and Unstable Nature of Source Code . 48
3.3.4 Time Required to Develop the Extractor 49
3.3.5 Additional Challenges 49
3.4 Schema For The C-Rex Evolutionary Change Data 53
3.5 The Implementation Of C-REX 55
3.5.1 Performance 61
3.6 Limitations Of The C-REX Approach 62
3.6.1 Dependency Analysis 62
3.6.2 Beyond C 63
3.6.3 Beyond CVS 64

3.6.4 More Detailed Change Tracking 64
3.6.5 The Use of Heuristics 64
3.7 Using C-REX In Practice 65
3.7.1 Acquiring Our Guinea Pigs 66
3.8 Related Work 67
3.9 Conclusion 68

4 Source Control Change Messages: How Are They Used And What Do They Mean? 71
4.1 Introduction 72
4.1.1 Organization of Chapter 74
4.2 Study Logistics 74
4.2.1 Study Goals 75
4.2.2 Study Participants 76
4.2.3 Study Design 77
4.2.4 Survey Design 80
4.3 Results For Part 1: Usage and Content of Change Messages 80
4.4 Results For Part 2: Classification of Changes 84
4.4.1 Analysis 1A and 1B of Developers' Classifications . . 87
4.4.2 Analysis 2 of Developers' Classifications 89
4.5 Results For Part 3: Comparing Open Source and Commercial Change Messages 92
4.6 Conclusion 93

Part II: Using Software Repositories to Assist Developers 99

5 Using Development History Sticky Notes to Understand Software Architecture 103
5.1 Introduction 104
5.1.1 Organization of Chapter 105
5.2 The Architecture Understanding Process 106
5.2.1 Propose 108
5.2.2 Compare 109

5.2.3 Investigate 109
5.3 The Software Reflexion Framework 110
5.3.1 A Clarifying Example 112
5.4 Investigating Dependencies - The W4 Approach 114
5.4.1 Three Types of Dependencies 115
5.4.2 Questions Posed During Investigation 116
5.4.3 Source Sticky Notes 117
5.5 Source Control Systems 119
5.5.1 Attaching Sticky Notes to Static Dependencies 120
5.6 Case Study 122
5.6.1 Investigating Removed Dependencies 126
5.6.2 Discussion of Results 127
5.7 Related Work 128
5.8 Conclusion 130

6 Replaying Development History to Assess the Claimed Benefits of Code Maintenance Tools and Strategies 133
6.1 Introduction 134
6.1.1 Organization of Chapter 137
6.2 The Change Propagation Process 137
6.2.1 Information Sources Used to Propagate Changes . . . 140
6.3 Measuring The Performance Of a Tool in Propagating Changes 142
6.3.1 A Simple Example 143
6.3.2 Performance Measures for a Single Change Set . . . 143
6.3.3 Performance Measures for Multiple Change Sets Over Time 147
6.3.4 Relative Performance of a Tool Over Time 149
6.3.5 Relative Stability/Volatility of the Performance of a Tool 150
6.4 The Development Replay (DR) Approach 151
6.4.1 Threats to Validity and Limitations of Results Derived Through the DR Approach 156
6.5 Case Study 161

6.5.1 Enhancing the Performance of FREQ(A) tools 163
6.6 Related Work 166
6.6.1 Change Propagation 166
6.6.2 The use of historical data 170
6.7 Conclusion 173

Part III: Using Software Repositories to Support Managers 177

7 The Top Ten List: Dynamic Fault Prediction 181
7.1 Introduction 182
7.1.1 Organization of Chapter 184
7.2 Motivation 185
7.3 Heuristics For The Top Ten List 188
7.3.1 Most Frequently Modified (MFM) 189
7.3.2 Most Recently Modified (MRM) 189
7.3.3 Most Frequently Fixed (MFF) 190
7.3.4 Most Recently Fixed (MRF) 190
7.4 Studied Systems 191
7.5 Measuring The Performance Of The Top Ten List 193
7.6 The Effects Of a Larger List 197
7.7 Discussion 199
7.7.1 An Accurate Measure of the Performance of a Heuristic 199
7.7.2 Performance of Fault Based Heuristics 199
7.7.3 Determining a Practical Average Prediction Age . . . 200
7.8 Related Work 202
7.9 Conclusion 202

8 Code Development Chaos: a New Perspective on Software Complexity 205
8.1 Introduction 206
8.1.1 Overview Of Chapter 209
8.2 The Code Development Process 210
8.3 Information Theory 213

8.4 The Basic Code Development Model . . . 214
8.4.1 Basic Model . . . 214
8.4.2 Intuition . . . 216
8.4.3 Files As a Unit of Measurement . . . 217
8.4.4 Evolution of Entropy . . . 218
8.5 Extended Code Development Model . . . 219
8.5.1 Evolution Periods . . . 220
8.5.2 Adaptive System Sizing . . . 221
8.6 The File Code Development Model (FCD) . . . 223
8.7 Case Study . . . 226
8.7.1 Building the Statistical Linear Regression Models . . . 227
8.7.2 Measuring and Comparing the Prediction Error for the SLR Models . . . 229
8.7.3 Determining the Statistical Significance for The Difference in Prediction Error between Models . . . 230
8.7.4 Comparing Models . . . 231
8.7.5 Threats to Validity . . . 235
8.8 Related Work . . . 239
8.9 Conclusion . . . 241

Part IV: Conclusion **243**

9 Contributions and Future Work **245**
9.1 Thesis Contributions and Findings . . . 247
9.2 Suggestions for Extending this Research . . . 248
9.2.1 Evolutionary Extractors for C++ or Java . . . 248
9.2.2 Integrating Source Notes into Graphical Browsers . . . 249
9.2.3 Better Change Propagation Techniques and More Realistic Evaluations . . . 249
9.2.4 Commercial Software Systems . . . 250
9.3 Opportunities for Future Research . . . 250
9.3.1 Grokking Through Time . . . 250

9.3.2 Visualizing the Recovered Data from Software Repositories 251
9.3.3 Recovery Of Aspects and Validation of Recovered Aspects 251
9.3.4 Change Distance and Design Quality 252
9.3.5 Discovery of Short Term and Long Term Evolution Patterns 252
9.3.6 Evolution of Clones 253
9.3.7 Standardization of Output 253
9.3.8 Development Decision Support (DDS) Appliances . . 253
9.3.9 Mining Other Repositories and Creating New Repositories 254
9.3.10 Migrating Source Control Repositories 255
9.4 Closing Remarks 255

Bibliography **257**

List of Tables

2.1 Classifying Developer's Replies About a Code Change 26
2.2 Summary of Evolutionary Extractors Design Choices 37

3.1 Size of Source Control Repository vs. Size of C-REX XML file . 52
3.2 Characteristics of the Guinea Pigs 65

4.1 Characteristics of the Participants of the Study 77
4.2 Summary of the Studied Systems 79
4.3 Classification Results for the Intermediate Developers Group vs. the Classifier Program(Analysis 1A) 87
4.4 Classification Results for the Senior Developers Group vs. the Automated Classifier 88
4.5 Kappa Values and Strength of Agreement 89
4.6 Classification Results for the Senior Developers Group vs. the Intermediate Developers Group 89
4.7 Classification Results for the Common Classifications between Both Developers Group vs. the Automated Classifier 90
4.8 Results of the Stuart-Maxwell Test 91

6.1 Classification of the Modification Records for the Studied Systems 156
6.2 Characteristics of the Studied Systems 161
6.3 Performance of FREQ(A) tools for the Five Studied Software Systems 164
6.4 Performance of RECN(M) tools for the Five Studied Software Systems 164

6.5 Summary of Work by Briand, Shirabad, Ying, and Zimmermann *et al.* in relation to our work. 175

7.1 Summary of the Studied Systems 193
7.2 HR, AHR, and APA for the Studied Systems During the 3 Years 196
7.3 AHR and APA for the Exponential Decay Heuristic 201

8.1 Summary of the Studied Systems 226
8.2 The R^2 statistic for all the SLR Models for the Studied Systems 229
8.3 The Difference of Error Prediction and T-Test Results for the $SLR\ Model_m$ and $SLR\ Model_f$ for the Studied Systems 231
8.4 The Difference of Error Prediction and T-Test Results for the $SLR\ Model_m$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$ for the Studied Systems 232
8.5 The Difference of Error Prediction and T-Test Results for the $SLR\ Model_f$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$ for the Studied Systems 234

List of Figures

2.1 Recovering the Evolution of Source Code 26

3.1 Conceptual View of the C-REX Extractors 43
3.2 Source Code Snapshot Frequency Choices 45
3.3 Level of Detail for Evolutionary Analysis 46
3.4 Schema for the Change Data Extracted By C-REX 54
3.5 Schema for the ChangeUnit 55
3.6 Example of a Simple Software System 56
3.7 Steps Needed to Generate the RDE 57
3.8 Contents of the Buckets for Entity *main* in the Simple Software System 58
3.9 Steps Needed to Generate the Evolutionary Change Data 60
3.10 Complexity Ratio Evolution 69

4.1 Analysis of the Classification of Changes by the Senior and Intermediate Developer Groups 86

5.1 Overview of the Architecture Understanding Process 107
5.2 Architecture Understanding Process Using The Software Reflexion Framework 111
5.3 Conceptual View of an Operating System [BHB99a] 112
5.4 Reflexion Diagram for an Operating System 113
5.5 Classification of Dependencies 116
5.6 Conceptual View of the NetBSD Virtual Memory Component . 123
5.7 Reflexion Diagram for the NetBSD Virtual Memory Component 124

5.8 Source Sticky Note for Dependency from *Virtual Address Maintenance* to *Pager* . . . 125

5.9 Source Sticky Note for Dependency from *Pager* to *Hardware Translation* . . . 126

5.10 Source Sticky Note for Dependency from *File System* to *Virtual Address Maintenance* . . . 127

6.1 Model of the Change Propagation Process . . . 139

6.2 Change Propagation Graph for the Simple Example - An edge between two entities indicates that a tool suggested one when informed about changes to the other one. . . . 144

6.3 The Development Replay Infrastructure . . . 152

6.4 Maintaining the Project State Incrementally . . . 154

7.1 Hit Rate For The 4 Proposed Heuristics . . . 194

7.2 Hit Rate Growth As a Function of The Top List Size Using MRM Heuristic . . . 197

7.3 Hit Rate Growth As a Function of The Top List Size Using MFM Heuristic . . . 198

8.1 Flow Of Complexity Between the Facets of a Software Project . 207

8.2 The Entropy of a Period of Development . . . 215

8.3 The Evolution of the Entropy of Development . . . 219

CHAPTER 1

Introduction

Software repositories (such as source control repositories) contain a wealth of valuable information regarding the evolutionary history of a software project. In this research we develop tools to recover such historical data. We present several techniques and approaches that make use of the recovered data to support managers and assist developers working on large software systems. We validate our work empirically using data based on over 60 years of development history for several open source projects.

SOFTWARE systems are continuously evolving and changing. Understanding these complex systems to maintain and enhance them is a challenging task. Managing projects building and maintaining such systems to produce highly reliable software on time and within budget is a difficult goal to accomplish.

In the last decade, software researchers have developed tools, presented methods, and proposed theories to support managers and guide developers as they evolve large software systems. Unfortunately, industrial adoption of such research has been limited. In many cases researchers are not aware of industrial practices and practitioners are often too busy

with their day to day problems to express their concerns to researchers [Gla03b]. Industrial case studies attempt to overcome this quandary, by involving researchers more with industrial problems [BKT03, KPJ+02, PPV00]. Unfortunately, case studies require the commitment of the industrial organization whose practitioners are usually busy trying to meet tight deadlines and are reluctant to participate in such studies. Moreover, the academic experimenter's intervention and monitoring is likely to affect the practitioners' attitude and performance [MWZ03].

Historical information stored in software repositories provides a middle ground that permits researchers to study industrial development processes and products while not interfering with development deadlines. Such repositories contain a wealth of valuable information for empirical studies in software engineering: *source control systems* store changes to the source code as development progresses, *defect tracking systems* follow the resolution of software defects, and *archived communications* between project personnel record rationale for decisions throughout the life of a project. Such data is available for most software projects and represents a detailed and rich record of the historical development of a software system. Moreover, current software engineering research approaches and techniques can benefit from using such historical information. For example, historical information can assist developers in understanding the rationale for the current structure of a software system and its evolutionary history.

1.0.1 Prior Research

Software repositories have primarily been used for historical record keeping activities such as retrieving old versions of the source code or tracking the status of a defect. Several studies have emerged that use this data to study various aspects of software development such as software design or its architecture, development process, software reuse, product reliability, and developer motivation.

Basili and Perricone used historical change data to study the types of software faults, the effort needed to correct them, and the risks of code reuse [BP84]. Perry and Evangelist explored the change data of a large long lived telephony system to determine the root cause of faults [PE85, PE87]. Perry *et al.* used change data along with surveys of developers to gain a better understanding of the type of faults in large software systems and the efforts associated with fixing them [PS93, LPS02]. Research by Gall *et al.* has shown that software repositories can support developers changing legacy systems by pointing out hidden dependencies in a software system [GHJ98, GJK03]. Eick *et al.* studied the concept of code decay and used the modification history to predict the incidence of faults [ELL+92, EGK+01]. Graves *et al.* showed that the number of modifications to a file is a good predictor of the fault potential of the file [GKMS00]. In other words, the more a subsystem is changed the higher the probability it will contain faults. Mockus *et al.* demonstrated that historical change information can support management in predicting bugs and fixing effort to ease the evolution of reliable software system [MWZ03]. Similarly, Khoshgoftaar *et al.* explored using process history to predict software reliability [HAK+96, KAH+98]. Chen *et al.* have shown that historical information can assist developers in understanding large systems [CCW+01]. Eick *et al.* presented visualization techniques to explore change data [ESEES92, SGEMGK02]. These early studies have highlighted some of the benefits of analyzing historical data.

1.0.2 Personal Experience

Working as part of several industrial organizations, such as IBM Research, Nortel, and Research In Motion, the author found himself and other developers examining software repositories (such as source control systems), in an ad-hoc fashion, to clarify many of our concerns and understanding of software systems or to gauge the state of a software project, for example:

1. In the role of a head developer of a project, we were frequently asked for estimates on when a project is ready for release or about the project's expected reliability or concerning the need for testing resources – we adopted very coarse estimates by examining the change history of the software system as stored in the source control repository.

2. In the role of a developer, we faced the daunting task of understanding large complex software systems which were developed by others, enhanced by many and patched frequently to meet tight deadlines or critical emergencies – we found ourselves along with other developers falling back to the initial version of a complex piece of code to understand it. In many cases, the initial cut of a piece of code, which is stored in the source control system, was easier to understand and was cleaner than the current code. In addition, we often investigated prior changes to code segments to gain a better understanding of the rationale for their current complexity or to clarify the design choices.

1.0.3 The Open Source Phenomena

The promising results obtained from prior studies along with our personal industrial experience highlighted to us the potential of software repositories in supporting developers and managers working on large software systems. We recognized the value of transforming such repositories from static record keeping repositories to active repositories used by researchers to gain empirically based understanding of software development, and by software practitioners to predict, plan and understand various aspects of their project. To pursue such research, we needed a large number of projects (*Guinea Pigs*) for which we could analyze the historical development records. With the explosive growth of open source projects, we were able to acquire the source control repositories for several open source projects.

Open source projects keep their repositories accessible online to permit developers around the world to contribute to their project. Furthermore, most of their communication and development documentation is archived online. The large number of available projects and the ease of access to their history permitted us to empirically verify our proposed techniques and approaches, and to interpret our findings.

1.1 Research Hypothesis

Prior research and our informal industrial experience lead us to the formation of our research hypothesis. We believe that:

> *Software repositories contain a wealth of valuable information about the evolution of a software project. By mining such historical information, we can develop techniques and approaches to support software developers and managers in their endeavors to build and maintain complex software systems.*

The goal of this thesis is to show the validity of this hypothesis through studying historical repositories for several open source projects. In particular, we develop several approaches and techniques that make use of the evolutionary history of software project to assist:

- Developers:
  - in understanding legacy code and discovering the rationale behind the current structure of the software system.
  - in ensuring that changes are propagated to the appropriate parts of the code.
- Managers:
  - in predicting faults in a software system.

  - in allocating their limited testing resources to the most appropriate parts of the software system.

This thesis shows that the mining process can be automated to robustly process the historical records from the source control repositories of several large software systems. By automating this process, we can study a large number of systems. By documenting and presenting this process, interested researchers and practitioners can easily apply our proposed techniques. Interested researchers and practitioners can also investigate other possible uses of the recovered data, or they can augment the recovered data using the described techniques to perform additional studies.

1.2 Thesis Organization

Throughout this thesis we demonstrate the value of mining software repositories by studying and formalizing ad-hoc techniques and approaches adopted by practitioners who use historical records as part of their day to day job.

The thesis has three main parts. Each part is geared towards a specific type of reader. Interested readers can focus on the particular parts that would satisfy their interests based on their specific role:

- Part I: This part presents the techniques and approaches that we developed to automate the mining of large historical repositories. Readers of this part are likely to be other researchers interested in the field of mining software repositories.
- Part II: This part presents the techniques and approaches that we developed to assist developers in understanding and changing legacy source code.
- Part III: This part presents the techniques and approaches that we developed to support the management of software systems by

mining the development history. Readers of this part are likely to be software managers who are responsible for allocating resources (such as testing resources) and who must ensure the reliability of a released software system.

To make each part self contained, some repetition may exist between the various parts to permit their reading independently. Nevertheless, repetition is minimized and readers are directed to the appropriate research if they are interested in more details. Related and prior work to each part are examined and studied in the corresponding parts of the thesis.

1.3 Thesis Overview

We now give an overview of the work presented in this thesis.

1.3.1 Part I: Extracting Information From Software Repositories

Researchers interested in mining software repositories need to automate the mining process and to have a good understanding of the quality of the recovered data. In this part we tackle the complexities associated with recovering the historical data and we study the quality of the recovered data.

1.3.1.1 Chapter 2: Studying The Evolution of Software Systems Using Evolutionary Code Extractors

Software systems are continuously changing and adapting to meet the needs of their users. Empirical studies are needed to better understand the evolutionary process followed by software systems. In this thesis we explore the potential of mining software repositories to assist software

practitioners. To perform such studies, we need tools that can analyze and report various details about the software system's history.

We propose evolutionary code extractors as a type of tool to assist in empirical source code evolution research. We present the design dimensions for such an extractor and discuss several of the challenges associated with automatically recovering the evolution of source code.

1.3.1.2 Chapter 3: C-REX: An Evolutionary Code Extractor for C

In this thesis, we focus on mining the data stored by a source control system as an example of a software repository. We discuss C-REX, an evolutionary code extractor for the C programming language. We present our design choices for C-REX, explain the reasoning for these choices, and give an overview of our extraction approach. We also discuss a number of limitations to our approach and show results of using C-REX to recover the evolution of several software systems.

Whereas, most source control systems record changes to the code at the file level, C-REX traces changes to specific source code entities, such as functions, variables, or data type definitions. Then we can track details such as:

- Addition, removal, or modification of a source code entity such as adding or removing a function.
- Changes to dependencies between the modified entities and other source code entities. For example, we can determine that a function no longer uses a specific variable or that a function now calls another function.

Furthermore, C-REX lexically analyzes the content of the change message attached to a modification to automatically classify modifications into three types: Fault Repairing modifications (FR), Feature Introduction modifications (FI), and General Maintenance modifications (GM).

1.3.1.3 Chapter 4: Source Control Change Messages: How Are They Used And What Do They Mean?

Source control systems permit developers to attach a free form message to each committed change. The content of these *change messages* is rarely investigated and little is known about their use by developers while they maintain their code.

We conducted a survey with professional software developers to investigate how developers make use of these messages and what type of information exists in them.We investigated the quality of the classifications done by C-REX. We also asked developers to compare change messages from the repositories of open source software systems to the messages from the repositories of commercial systems.

In particular, we sought to answer questions such as:

- Do developers usually enter meaningful and descriptive change messages? Are they likely to leave the messages empty?
- Do developers monitor such messages and react to their content?
- Do developers make use of these message as they maintain and enhance their software system, or are they ignored?
- Can we automatically determine the type of a change as being a bug fix or a feature?

The findings of our survey suggest that change messages are a valuable resource which practitioners use to maintain and manage software projects. For example, practitioners use change messages to understand the code when they are fixing a bug. Moreover, change messages in open source projects are similar to messages that developers encounter in large commercial projects. An automated approach to determine the purpose of a change using the change message is likely to produce results similar to a manual analysis performed by professional developers. The results of

this survey along with our ability to automate the mining process of software repositories encourage us to investigate techniques and approaches to study and formalize the ad-hoc uses of repositories by practitioners. The following two parts of this thesis study the use of historical data derived from software repositories to support developers and managers.

1.3.2 Part II: Using Software Repositories to Support Developers

Developers maintaining large software projects need tools to assist them in changing the software system and understanding it. The cost of performing incorrect changes to legacy system is very high as it will likely introduce bugs into the software system. In this part, we tackle these issues by using data derived from the source control repositories to assist developers in propagating changes and in understanding the architecture of legacy software systems.

1.3.2.1 Chapter 5: Using Development History Sticky Notes to Understand Software Architecture

Dependency graphs have been proposed and used in many studies and maintenance activities to assist developers in understanding large software systems before they embark on modifying them to meet new requirements or to repair faults. Call graphs and data usage graphs are the most commonly used dependency graphs. These graphs show the present structure of the software system ($e.g.$ In a compiler, an *Optimizer* function calling a *Parser* function). They fail to reveal details about the structure of the system that are needed to gain a better understanding. For example, traditional call graphs cannot give the rationale behind an *Optimizer* function calling a *Parser* function.

We present an approach which recovers valuable information from source control systems and attaches this information to the static dependency graph of a software system. We call this recovered information

– *Source Sticky Notes*. We show how to use these notes along with the software reflexion framework [MNS95] to assist in understanding the architecture of large software systems. To demonstrate the viability of our approach, we apply it to understand the architecture of NetBSD – a large open source operating system.

1.3.2.2 Chapter 6: Replaying Development History to Assess The Benefits of Code Maintenance Tools and Strategies

Practitioners are faced with many tools and methodologies promising to ease their maintenance tasks. Code restructuring methodologies claim to ease software evolution by localizing changes. Development environment tools assert their ability to assist developers in propagating changes. Static source analysis tools (such as lint) promise to point out error prone code. Unfortunately, such claims and promises are rarely substantiated or tested although the cost of adopting such tools and approaches is high and the risks of failures are even higher.

We propose to use the historical information stored in software repositories (such as source control systems) to assess such claimed benefits. We present the *Development Replay* (DR) approach which reenacts the changes stored in the source control repositories using a proposed tool or strategy. We present a case study where the DR approach is used to empirically assess and compare the effectiveness of several not-yet-existing tools which promise to assist developers in propagating code changes. The approach is illustrated through a case study for 5 large open source systems with over 40 years of development history.

1.3.3 Part II: Using Software Repositories to Support Managers

Managers of large software projects are always in search for techniques and approaches to determine the quality of their software system and to

allocate their limited resources wisely. In this part we tackle these issues by using data derived from source control repositories.

1.3.3.1 Chapter 7: The Top Ten List

To assist managers in coping with the challenges of allocating their limited resources effectively, we present an approach (*The Top Ten List*) which highlights to them the ten most susceptible subsystems to have a fault. The list is updated dynamically as the development of the system progresses. Managers can focus testing resources to the subsystems suggested by the list. In contrast to count based techniques which focus on predicting an absolute count of faults in a system over time, or classification based techniques which focus on predicting if a subsystem is fault prone or not, we focus on predicting the subsystem that are most likely to have a fault in them in the near future. For example, even though a subsystem may not be fault prone and may only have a few number of predicted faults, it may be the case that a fault will be discovered in it within the next few days or weeks. Or in another case, even though a fault counting based technique may predict that a subsystem has a large number of faults, they may be dormant faults that are not likely to cause concerns in the near future.

If we were to draw an analogy to our work and rain prediction, our prediction model focuses on predicting the areas that are most likely to rain in the next few days. The predicted rain areas may be areas that are known to be dry areas (*i.e.* not fault prone) or may be areas which aren't known to have large precipitation values (*i.e.* low predicted faults).

We believe that the Top list approach holds a lot of promise and value for software practitioners, it provides a simple and accurate technique to assist them in managing their resources as they maintain large evolving software systems.

1.3.3.2 Chapter 8: Code Development Chaos – a New Perspective on Software Complexity

Using sound mathematical concepts from information theory such as Shannon's Entropy [Sha48], we present a novel view of complexity in software. We propose a complexity metric that is based on the process followed by software developers to produce the code (*the code development process*) instead of on the code or the requirements. We conjecture that:

> *A chaotic code development process negatively affects its outcome, the software system, such as the occurrence of faults.*

We validate our hypothesis empirically through case studies using data derived from the development process history of six large open source projects. Our entropy measurements have statistically significant better accuracy in predicting the occurrence of faults than simply using the number of prior modifications to a subsystem or prior faults in it as predictors of faults. If our complexity metric is used it is likely to assist managers avoid delays and faults in a project over time.

1.4 Thesis Contributions

The conceptual contributions of this thesis center around the development of techniques and approaches to demonstrate the value of mining software repositories in assisting managers and developers in performing a variety of software development, maintenance, and management activities. The technical contributions of this thesis focus on the development of tools and the invention of techniques to robustly automate the mining process for large long lived software systems written in industrial languages such as C. The empirical contributions of this thesis are the application of all proposed techniques and approaches on several long lived large open source projects.

The main contributions of this thesis are as follows:

- The proposal of evolutionary extractors as a central and critical tool for recovering the evolutionary history of software systems. Such a tool enables software engineering researchers to perform empirical studies of software evolution and software repositories mining. Furthermore, the design dimensions for such an extractor and the challenges associated with building them are discussed to assist others interested in building or using such extractors.

- The development of an evolutionary extractor (C-REX) for the C programming language. This extractor can process large long lived projects robustly and within reasonable time. The recovered data forms the empirical basis of the research presented in this thesis.

- The first survey of its kind to examine the usage of source control systems by practitioners. The results of the survey highlight the value of source control repositories in assisting practitioners. The presented results are likely to encourage other researchers to investigate approaches and build better tools which make use of source control data.

- The development of *Source Sticky Notes*, which attach historical information to traditional dependency relations, such as call relations. These sticky notes assist developers in program understanding.

- The development of the *Development Replay* approach which reenacts the development history of a software project using the changes stored in the source control repositories. This approach permits us to empirically asses the effectiveness of not-yet-adopted or not-yet-existing code maintenance tools and strategies.

- The proposal of a new view of software complexity (*Code Development Chaos*) which focuses on the change patterns stored in the source control repositories instead of focusing on the code itself.

- The proposal of a novel technique (*The Top Ten List*) to measure the benefits of a bug predictor in allocating testing resources. The

approach measures the applicability of the predictors results along with the limited resources assigned for testing an application in industry.

Part I

Extracting Information From Software Repositories

Software repositories are available for most large software projects. Yet the data stored in these repositories has rarely been the focus of software engineering research. We believe that this is mainly owing to the following reasons:

- The limited access to such repositories prevented researchers from using them in their work. Until recently researchers could not easily gain access to historical development repositories. Companies in many cases were not willing to give researchers access to such detailed information about their software systems and their evolution history. Another possible source for software repositories and software systems to study is academic systems. Unfortunately, software systems developed in academia tend to have a small number of developers, a short life span, and their development history is not as rich nor as interesting as the history of long lived industrial software systems. The earliest research work in the area of mining software systems were based on the repositories of commercial software systems and were done in cooperation with a few commercial organizations [BP84, PE85, ELL+92, HAK+96, GHJ98, GKMS00, MWZ03, Shi03].

- The complexity of processing large repositories in an automated fashion hindered the adoption and integration of software repositories in other software engineering research. In many cases, software engineering researchers do not have the expertise required nor do they have the interest to recover data from software repositories. Recovering such data requires a great deal of effort and time from researchers, instead they are more interested in gaining convenient access to the recovered data in an easy to process format.

With the advent of open source systems, the accessibility to repositories of large software systems became possible. Researchers now have access to rich repositories for large projects developed by hundreds of developers over extended periods of time. This lead to early research in mining software repositories which was based on open source projects [CCW+01].

This part discusses the challenges and complexities associated with recovering the data stored in software repositories. The part starts off by proposing the need for evolutionary extractors which can recover the evolutionary history of software systems. By recovering the data and representing it in a simple and easy to access format, the data becomes more accessible for researchers to study and investigate. As an example of an evolutionary extractor, we present C-REX an evolutionary extractor for the C language. C-REX recovers the evolutionary history of a software project from the repository of its source control system. We end this part with a survey we conducted with professional software developers to investigate how developers make use of source control systems in practice. We investigated the quality of some of the data produced by the C-REX extractor. We also asked developers to compare change messages in open source software systems to change messages in commercial systems.

This part is likely to be of interest to software engineering researchers, in particular researchers interested in mining software repositions. This part shows that the mining process can be automated through a number of techniques. It also presents an analysis of the quality of the recovered data. Researchers throughout the different areas of software engineering can now focus on analyzing the recovered data from software repositories, instead of spending a large amount of time building tools to recover the data first.

CHAPTER 2

Studying The Evolution of Software Systems Using Evolutionary Code Extractors

Software systems are continuously changing and adapting to meet the needs of their users. Empirical studies are needed to better understand the evolutionary process followed by software systems. These studies need tools that can analyze and report various details about the software system's history.

We propose evolutionary code extractors as a type of tool to assist in empirical source code evolution research. We present the design dimensions for such an extractor and discuss several of the challenges associated with automatically recovering the evolution of source code.

2.1 Introduction

SOFTWARE systems are continuously changing and adapting to meet the needs of their users. A good understanding of the evolution process followed by a software system is essential. This would permit researchers to build better tools to assist developers as they maintain and enhance these systems. Furthermore, it will pave the way for the investigation of techniques and approaches to monitor, plan and predict successful evolutionary paths for long lived software projects.

We could study the evolution of a number of facets of a software project such as its requirements, its architecture, its source code, its bugs reports, or the interactions and communications among its developers. Each facet offers insight on a variety of issues surrounding the evolution of a software system. For example, studying the complexity of the source code or the number of reported bugs over time may give us a better understanding of how bugs are introduced in software systems. It may also assist us in building models to predict bugs and models to guide managers in allocating testing resources where they are needed the most [HH].

To perform such studies a good record of these facets throughout the lifetime of a project is essential. For example, a record of the requirements of a software system since its inception till the current day is needed to study the evolution of its requirements. For some facets such as the requirements of a software system, such records rarely exist and if they do exist they tend to be incomplete or too high level. For other facets such as the features of a software system, they may be well documented in release notes, but it may be challenging and time consuming to recover them. For example, Anton and Potts have manually traced the evolution of features for telephony services [AP01]. Their study focused on telephony services in a single city due to the long time and resources required to manually distill and describe the evolution of features from the phone books.

We should focus on facets for which most projects have good historical records and which can be automatically analyzed with minimal effort. An

empirical approach permits us to generalize our findings instead of associating them to peculiarities of specific systems. Luckily, a large number of software projects store artifacts generated throughout their lifetime in software repositories. For example, the source code and changes to it are recorded in a source control repository. The released versions are usually stored in release archives. Other repositories archive the mailing lists and emails among the project's developers. Bug tracking systems record various details regarding reported bugs and their fixes. These repositories provide a great opportunity for researchers to acquire empirical data to assist them in studying evolution.

To ensure that we can perform our studies on several software systems, we need tools that automatically recover data from these repositories and present the data in a standard format that is easier to process. This would permit researchers to focus on analyzing the recovered data instead of spending a large amount of time building tools to recover the data first.

The source code of a software project can be thought of conceptually as the DNA of the software. The source code encodes the software system's functionality. Studying changes to the various characteristics of the source code will help us understand the evolution of the software system. Moreover, there is a large body of research which demonstrates approaches to recover characteristics of the source code using automated techniques. Hence the source code is a very attractive facet of a software project to study its evolution using automated techniques. In this chapter, we argue the need for tools that could process the source code history of a software project and generate useful data automatically. We call such software tools *evolutionary code extractors*, as they extract the evolution of source code.

2.1.1 Organization Of Chapter

This chapter is organized as follows. Section 2.2 tackles the issue of describing the evolution of source code. We discuss several ways to describe

the same change to the source code. We argue the need to choose descriptions which can be recovered automatically. Section 2.3 overviews the dimensions associated with studying and recovering the evolution of source code. The choices made by researchers along these dimensions influence the techniques used to build evolutionary code extractors. Section 2.4 highlights the challenges and complications that arise based on the choices along the dimensions presented in Section 2.3. Section 2.5 describes prior work which dealt with studying source code evolution. The prior work is presented and the design choices used by their extractors are explored using the dimensions presented in this chapter. Section 2.6 concludes the chapter with parting thoughts about evolutionary extractors and their benefits for studying software systems and validating our understanding of the evolution process followed by software systems.

2.2 Describing Source Code Evolution

Describing the evolution of the source code amounts to describing the changes that occur to it. The simplest way to describe source code changes is by describing their effect on the code size (the addition and removal of lines of code).

We are interested in means to describe source code changes that can be automatically recovered and which are richer than simply describing the addition and removal of lines. For example, even though terms such as perfective, corrective, and adaptive are usually used to describe changes to the code; it is not possible to confidently and accurately describe the evolution of a software system in an automated fashion using these terms. We would require a large number of heuristics, human intervention, and intuition to rank changes to source code accordingly. In short, we seek approaches that provide a balance between the richness of the recovered descriptions and the ease of automating the recovery process.

Consider a developer who is asked about her/his activities in the last few days, a number of replies are possible. Each reply describes the ac-

tivities performed at a specific level of detail and in respect to particular characteristics of the software system. For example, the developer working on a text editor software system may say: "*I added support for saving a text file, I also fixed a bug in the layout engine used by the editor.*" This reply describes change at the feature level.

Instead if we were to ask the developer to elaborate more on her/his changes and their effect on the source code ($i.e.$ to describe her/his changes to the source code), we are bound to get an even larger and more diverse number of replies which describe the same exact change work from different perspectives. The following is a list of possible replies:

1. *I changed 5 lines in the source code.*

2. *I added 3 lines in file main.c. I also commented out 2 lines from file layout.c.*

3. *I added 1 line in the main() function, 2 lines in the init() function, and removed 2 lines from the refreshLayout() function.*

4. *I got the main() function to call function init() and I added a check in the init() function to make sure the filename is set before I call refreshLayout(). Also in the refreshLayout() function, I no longer check if the filename is set.*

The first reply deals with changes to the size of the overall system. The second reply is more specific, it specifies the location (files) of these changes. The third reply is even more specific than the second reply as it maps the changes to the exact function (source code entity) where they occurred. Finally, the fourth reply describes the change using its effect on the call dependencies between the code entities. Table 2.1 summarizes the developer's replies.

Reply #	Characteristic	Level Of Detail
1	Size (LOC)	System
2	Size (LOC)	File
3	Size (LOC)	Function
4	Structural (Call Dep.)	Function

Table 2.1: Classifying Developer's Replies About a Code Change

2.3 The Dimensions Of Source Code Evolution

In the previous section, we showed that a simple change could be described in a number of ways. Each way focuses on a particular characteristic of the source code at varying levels of details. Researchers studying the evolution of source code need to build tools (evolutionary code extractors) to recover and describe this evolutionary process. We believe that there are a number of design dimensions which they should address before they embark on building these tools. In this section we focus on these design dimensions and list the choices associated with each dimension.

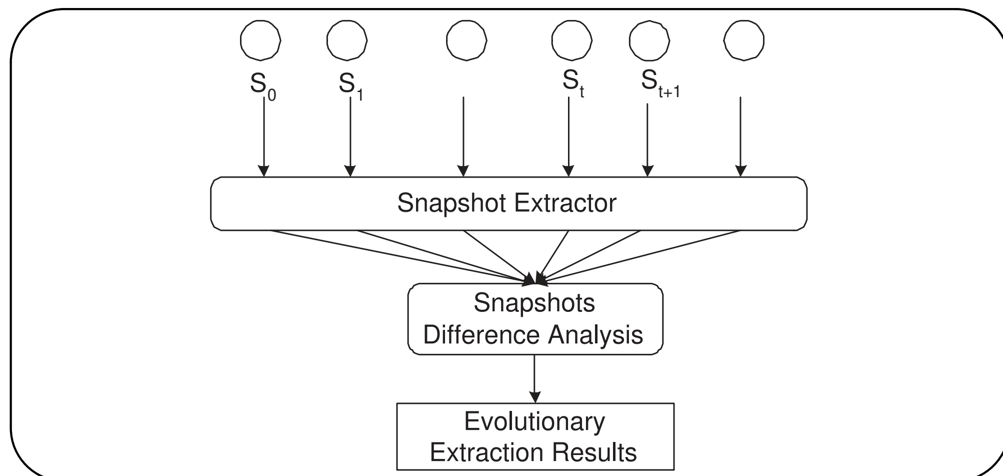

Figure 2.1: Recovering the Evolution of Source Code

2.3.1 Frequency of Snapshots

The source code of a software is continuously changing. We need to determine the frequency at which we should observe the code. Consider Figure 2.1, conceptually to monitor the evolution of the source code we need to decide on a number of historical snapshots of the system's source code. We then need to define some characteristics of these snapshots and study the differences between consecutive snapshots along these characteristics. The frequency of observations/snapshots determines the number of snapshots and their moment in time. Several methods exist to define snapshots:

- **Event based**: Source code progresses through different events during the lifetime of a project. For event based snapshots, we would use project events to determine the snapshots. Examples of these events are:
  - *Change*: A code change is simply the addition, removal or modification of a single line to a software system. Using a change frequency would produce the largest number of snapshots, due to the large number of changes that occur throughout the lifetime of a software system.
  - *ChangeList*: A changelist is the grouping of several code changes to represent a more complete view of a change. For example, a changelist may contain two changes – one change is the addition of a function f2() and the other change is the addition of a call of function f2() in function f1(). These two changes might be required to implement a specific feature or fix a particular bug.
  - *Build*: A build represents the grouping of several features. Builds are usually done to merge the various features that have been developed by the team members. Builds may be requested by the project lead as an indication of achieving development milestones or to track the progress of a project towards the final

release. Nightly builds, release candidate builds, and feature-complete builds are examples of builds.

  - *Release*: A release represents the grouping of a large number of features. The release is sent to customers and users.

- **Time based**: Time based snapshots are independent of the project and source code state. Instead they are done based on calendar time, such as weekly, monthly, and quarterly snapshots.

If we were to build an evolutionary extractor, we would conceptually have to process each snapshot using a *snapshot extractor*. The snapshot extractor would determine the characteristics of each snapshot. Then we would perform a snapshots difference analysis. This analysis would determine changes in the studied characteristics between each pair of consecutive snapshots (see Figure 2.1).

The choice of which snapshot frequency to use is dependant on the type of analysis that will be performed on the recovered data. If we were to study the average number of functions that must be changed to implement a feature then a changelist frequency may be the most appropriate choice.

The choice of the frequency of snapshots determines the number of snapshots which will be studied. If release snapshots are used then we will have a smaller number of snapshots in comparison to using change snapshots. The number of snapshots affects the performance of an evolutionary extractor. The larger the number of snapshots, the more time is required to perform the analysis.

2.3.2 Data Source

When studying the evolution of living creatures throughout time, we are usually limited by the availability of fossils of these creatures. Or if we are able to monitor the creatures as they evolve we are limited by the monitoring frequency. Whereas for studying the evolution of source code,

we have a much richer fossil history. Source code control systems, which are available for many long-lived software systems, store each change to the source code. Hence, we can track the evolution of the source code at a very high frequency (*change frequency*). If we were to draw a parallel to monitoring the evolution of a living creature, the data stored in the source control repository is equivalent to the creature informing the researcher monitoring it each time it is about to evolve/change. This is clearly impossible in living creatures but fortunately possible in source code thanks to the detailed records kept by source control systems.

Alternatively, we can deploy tools to monitor and record the developer activities during code development as done by Sayyad Shirabad [Shi03] and Schneider [SGPP04]. This later approach may be used when source control repositories are not accessible. Or it can be used when additional details, not available in the source control repository, are needed. Furthermore, project release archives which store a copy of released software may be used to study the evolution at the release frequency. km

2.3.3 The Characteristics of the Source Code

As we seek to describe the evolution of source code, we need to define a set of characteristics and techniques to measure these characteristics. We can then describe the evolution of the source code in terms of these characteristics and changes to them. For example, the size of the source code (*i.e.* the lines of source code) is a characteristic which can be measured easily. We can then compare the evolution of the software system from one snapshot to the next.

In this subsection we cover a number of possible characteristics. The choice of characteristics to monitor is dependant on the research performed and the ease of recovering such characteristics from the source code. For example, recovering the number of lines of the source code is easier and less resource intensive than recovering the current dependency structure of the source code. We chose to focus on the static aspect of the

source code instead of its dynamic and behavioral aspects due to the complexity associated with recovering and describing behavioral and dynamic changes to the source code.

We can describe the characteristics of source code using two general approaches:

- **Metric**: Metric approaches define measures to describe the current state of the source code. The simplest measures are metrics such as the Lines Of Code (LOC) or the number of defined functions. Other elaborate metrics such as complexity metrics could be used as well. Using metric approaches we can track changes in the value of the metrics (characteristics) over time.

- **Structural**: Structural approaches describe the current structure of the code. They could describe the dependency structure of the code such as '*function_1* depends on *function_2*', or they could describe the include dependencies such as '*file_1* includes *file_2*'. Using structural approaches, we could track changes in the structure of the source code, such as the addition of new functionality and its effect on the dependencies among the various parts of the source code. For example, we would expect once a function is added, other functions will be changed to call (depend on) it.

Some characteristics are cumulative, such as the number of functions, in the sense that characteristic measures derived based on a high frequency snapshots (such as change frequency) could be combined to study the same characteristic at the release level (*i.e.* the number of functions that exist per release). This is usually not possible for a large number of characteristics such as complexity metrics. It may be beneficial to recover the evolution of source code using the most number of snapshots (change level) then to abstract the data for less frequent snapshots (release level). Using this approach the recovered evolutionary data could be used for a variety of studies based on the desired level of frequency.

2.3.4 Level of Detail

The level of the detail of the characteristics for a snapshot varies. Some snapshot extractors recover information at the system level such as the number of lines of the whole system, whereas other extractors can recover details at the function level such as the number of lines of each function. Or for structural characteristics some extractors recover the interaction between source code files, such as 'file $x.c$ calls file $x.h$'. Whereas other extractors report information at a lower and more detailed level, such as 'function $f1$ calls function $f2$'. Also some extractors detail information about the internals of a function, such as '$func1_for_loop_1$ calls function $f2$'.

The level of details for a snapshot defines the level at which the evolution of the source code can be described. It also limits the type of analysis that could be performed on the recovered data. The more detailed the extracted data, the more complex it becomes to develop a snapshot and an evolutionary extractor to generate this type of information, we believe there are a number of detail levels:

- **System Level**: At the system level a single value is generated for each snapshot of the studied system such as the total number of lines or the total number of files in the software system. Developing snapshot extractor for this level is usually easier than the other more detailed levels.
- **Subsystem Level**: At the subsystem level, the source code is divided into a small number of subsystems. Metric values for each subsystem or structural information about the interaction between these subsystems are generated by the snapshot extractor. For example, the source code of an operating system may contain four subsystems: a Network Interface, Memory Manager, Scheduler, and File System subsystems. A metric approach may track the size of each of these four subsystems. A structural approach may track the dependencies between these four subsystems.

- **File Level**: At this level, the extractor reports changes at the file level, such as the number of functions in a file or the number of lines in it. For example, an evolutionary extractor would detail information such as on Feb 2, 2004 file $x.c$ had 10 lines changed in it: 5 lines added and another 5 removed.

- **Code Entity Level**: At the code entity level, the snapshot extractor describes the snapshot based on code entities such as functions, classes, macros, or data types. For example, it could report the number of lines in a function, or the dependencies between the functions in the source code. This data could be used during the snapshots difference analysis to report the addition of a new call to a function or the removal of a data dependency from another function. At this level of detail, the concepts of a function and data type renaming are possible. For example, it may be expected from an evolutionary extractor to report that a function was renamed. We believe that the detection of renaming of a source code entity versus the addition and removal operation of two sperate entities should be done as a post extraction step using techniques such as the ones described in [TG02].

- **Abstract Syntax Tree (AST) Level**: The AST level represents the lowest level of detail for information produced by an extractor. At this level, the snapshot extractor produces the AST of the source code. The AST is studied during the snapshots difference analysis (see Figure 2.1) to report changes to the internals of entities such as the addition/removal of new fields in a data structure, or if-branches and case statements inside functions.

The level of detail in the extracted information limits the type of analysis possible. It may also complicate the development of the extractor, for example AST level evolutionary extractors are much harder to develop as they require the development of snapshot extractors which can parse the source code and produce very low level details about its characteristics.

In contrast, a system level evolutionary extractor is much simpler to develop as it does not need to perform detailed analysis of the source code snapshots.

2.4 Challenges And Complexity

In an ideal situation, we would develop extractors that would describe the evolution of the source code at the most detailed level, the AST level, and at the highest frequency (change frequency). Unfortunately this is a rather difficult problem and developing such an extractor would be too complex and time consuming.

As researchers approach the problem of building an evolutionary extractor, they must decide on the choices along the dimensions, discussed in the previous section. The benefits and limitations of each decision are weighted using many criteria. The most important criteria we found in our work are the needs of the research for which the extractor is being developed, the time allocated for the project, and the available funding. We have developed several source code extractors for many programming languages in the last few years and we found that this engineering approach is paramount for the success of such projects due to the unsurmountable effort and challenges surrounding the development of the most suitable and practical extractor [HH02]. We cover a few of the challenges associated with developing evolutionary code extractors.

2.4.1 Robustness and Scalability

When studying the structural evolution of source code, we need to develop an extractor that can analyze the source code to determine structural dependencies among source code entities. An approach which is based on a text book grammar will fail to parse legacy systems, due to the disparate dialects of programming languages and the multitude of proprietary compilers extensions. This variety complicates building an extractor. The

developers of snapshot extractors could adopt various approaches to deal with the complexity of parsing legacy software systems. Some developers may choose to have their extractors recover gracefully when such extensions are processed, others may choose to specialize their parsers for such peculiarities using a variety of parsing techniques such as island grammars, robust parsing, and precise parsing [Ste03]. The choice of techniques to use is dependent on the peculiarities of the studied software systems. An evolutionary extractors should be robust and recover gracefully without user intervention to permit the analysis of several snapshots in an automated fashion.

Furthermore, the scalability of an extractor is another difficult challenge, as extractors are expected to analyze large software systems which may contain several million lines of code. This is complicated more by the fact that this analysis must be performed for each snapshot of the code and there could be thousands of snapshots. For example, examining a million line of source code at the change frequency would conceptually require the extraction of the characteristics associated with the source code after each change. This may require that the analysis of a million lines of code is repeated thousands of times, such an approach becomes infeasible and impractical. Instead developers of evolutionary extractors must develop more elaborate techniques to overcome this challenge.

An ideal goal for a snapshot extractor is to confine the length of extraction time to be less than the compilation time of the analyzed software system. In contrast to evolutionary extractors, even meeting this goal would require too much time and would make using an evolutionary extractor infeasible and impractical to study long lived software projects. Incremental extraction techniques similar to incremental compilations may assist in speeding up evolutionary extractors.

2.4.2 Accuracy

Ensuring the high accuracy of the extractor output is another challenging task. Given the size of the software systems extracted and the techniques

used to recover from different system peculiarities, extractors have the tendency to miss some relations (*false negatives*), or in some cases add superfluous ones (*false positives*). Accurate extractors require rather complex language grammars and adopt several elaborate techniques to recover from errors and correctly identify information. Studies have shown empirically the difficulty faced by already well established extractors in ensuring this accuracy [AT98, MNGL98]. Evolutionary extractors would use similar techniques, therefore we expect that they would have to face similar challenges.

To make matters worse, when dealing with extraction over an extended period of time, the adopted approaches have to deal with unrelated entities having similar names appearing and disappearing throughout time.

2.4.3 The Changing and Unstable Nature of Source Code

An evolutionary extractor conceptually performs its work by analyzing each snapshot then comparing the generated information for each snapshot. As pointed out, this may be too time consuming. Furthermore, this would require the source code to be in some compilable stable state to permit the snapshot extractor to process it. This is not possible for example, when studying source code evolution at the change frequency – a developer may add a call to a function before she/he defines the function. Therefore, intelligent approaches are needed to analyze un-compilable and incomplete source code. Alternatively, we may choose to use less frequent snapshots that are more likely to be complete such as nightly builds or code-complete builds. These builds are less likely to cause the snapshot extractors to fail.

2.4.4 Development Time

Another challenge associated with evolutionary extractors is the time needed to develop them. An ideal solution would be to adopt a regular

source code extractor and modify it. In particular, for each change in the project we can rerun the extractor and compare the output of the extractor run before and after the change. Unfortunately, as highlighted in the previous subsection this may not be the optimal solution as the source code may not be compilable. Therefore evolutionary extractors must either be built from scratch or built by adopting regular extractors and enhancing them to deal with many of the aforementioned challenges. Clearly reusing already developed extractors would speed up the development time but may limit the type of analysis and may negatively affect the performance of the evolutionary extractor. On the other hand, building an extractor from scratch may provide the most flexible approach but would require a longer development time.

2.5 Previous Work

In this chapter we advocate the need for evolutionary extractors. We present several critical dimensions based on which such extractors should be designed. A number of evolutionary extractors have been built by many researchers studying the evolution of software systems. Although the term evolutionary extractor was not used by these researchers, the type of analysis performed by them fit well into our definition of an evolutionary extractor. In this section, we overview their work and present it using the design dimensions for evolutionary extractors presented in this chapter.

Work by Lehman *et al.* [LRW+97] tracked the evolution of the size of the source code, to perform such analysis evolutionary extractors that used code metrics at the system level monitored changes to the size of each release. Godfrey and Tu [Mic00] developed evolutionary extractors that use metrics at the system and subsystem level to monitor the evolution for each release of Linux. In addition, Tu and Godrefy [TG02] developed evolutionary extractors that track the structural dependency changes at the file level for each release of the GCC compiler.

Reference #	Snapshot Frequency	Data Source	Characteristics of Code	Level of Detail
[LRW+97]	Release	Release Archives	Metric (LOC)	System
[Mic00]	Release	Release Archives	Metric (LOC)	System/ Subsystem
[TG02]	Release	Release Archives	Structural (Call/Data Depend.)	File
[GHJ98]	Changelist	Source Control	Structural (Co-Change)	File
[ZDZ03]	Changelist	Source Control	Metric (Change)	File
[ZWDZ04]	Changelist	Source Control	Metric (Co-Change)	Function, File
[Ger04a]	Changelist	Source Control	Metric (Change)	Function, File, Subsystem

Table 2.2: Summary of Evolutionary Extractors Design Choices

Gall *et al.* [GHJ98, GJK03] have developed evolutionary extractors that track the co-change of files for each changelist in CVS. Zimmermann *et al.* [ZDZ03] present an extractor which determines the changed functions for each changelist. Table 2.2 summarizes the design choices for each of the extractors developed by other researchers.

Draheim and Lukasz present a software infrastructure to study and visualize the output of evolutionary extractors, in particular they focus on visualizing metric evolutionary extractors using graphs [DP03].

2.6 Conclusion

Software practitioners and researchers have recognized the need to study the evolutionary process of software projects. The source code is an ideal facet of a software project to monitor and analyze as we can easily acquire various snapshots of it as it evolves. Furthermore, we can build tools – evolutionary code extractors – to automatically recover this evolutionary process. This recovered process could improve our understanding of soft-

ware development and assist developers who are maintaining large long lived software systems.

In this chapter, we advocated the need for such evolutionary extractors. We presented the various dimensions along which such extractors could be built. We discussed the challenges and complexities associated with the choices taken along these dimensions. These challenges complicate the development of such extractors, nevertheless we believe that a number of common extractors could be developed and reused within the research community to further empirical based understanding of software evolutionary processes. We also presented previous work which studies the evolution of code and attempted to classify these published extractors using the dimensions and choices we presented herein.

In the following chapter, we give a more concrete example of an evolutionary extractor. We present the implementation of an evolutionary extractor for the C programming language.

CHAPTER 3

C-REX: An Evolutionary Code Extractor for C

We discuss C-REX, an evolutionary code extractor, which we developed to recover information from source control repositories. We present our design choices for C-REX, explain the reasoning for these choices, and give an overview of our extraction approach. We also discuss a number of limitations of our approach and show results for using C-REX to recover the evolution of several software systems. We believe that the discussion presented here is beneficial for others interested in recovering and studying data stored in source control repositories.

3.1 Introduction

SOURCE code encodes the functionality of the software system in terms of modules and source code entities (such as functions and objects) which interact to implement the various features of a software system. Throughout the lifetime of a software system, its source code evolves to

implement various features and satisfy the needs of its users. Source control systems, such as CVS, track changes to the source code over time. They attach to each change additional information concerning the change such as the name of the developer performing the change, the date the change was performed, and a detailed message describing the change.

To overcome the lack of documentation and the pressing need to understand large systems as developers evolve them, developers can use the information attached to changes to understand the current structure of the software system [HH04e, CM03].

The history of changes could be used to determine the benefits of adopting different development techniques and approaches [ABGM99]. It can also be used to measure the effectiveness of development tools and to build better tools more suited to the types of challenges and difficulties faced by developers working on large software systems. For example, we can use the history of source code changes to study change propagation in software systems. As developers change software entities such as functions or variables to introduce new features or fix bugs, they must propagate the effects of these changes to other entities in the software system. Many difficult to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly.

To investigate if there are good indicators such as call graph relations that could assist a developer in determining other entities to change, we could propose several change propagation heuristics and study the performance of each heuristic using historical code changes to software systems [HH04c, Yin03, ZWDZ04, Shi03]. Such a study would require a good record of the dependency structure of the software system throughout its lifetime. Furthermore, we would need to track changes that occur to the entities of the source code and the dependencies among them. Using this information we would know the dependencies in the software system at any given moment in time. We could thereby determine if these dependencies or other factors can explain the propagation of changes.

Unfortunately, source control systems usually treat source code as simple text instead of tracking changes to the code at the level of functions, data types, and dependencies among them. Source control systems track changes at the level of lines changed. For example, the source control system would record that line number four of file *filename.c* was deleted, instead of recording that a line was removed from function *main()*, or that function *main()* no longer calls function *printf()* to be more specific.

To use the change information stored in source control systems, we need to recover this information from the source control systems and transform it. Whereas, source control systems record changes at the file level, our goal is to record changes at the source code entity level (function, macro, variable, or data type definition). Then we can track details such as:

- Addition, removal, or modification of a source code entity. For example, adding or removing a function.
- Changes to dependencies between modified entities and other source code entities. For example, we can determine that a function no longer uses a specific variable or that a function now calls another function.

Using this derived information we associate with each source code change other changes that occurred in other files. We also know the static dependencies between source code entities at the moment in time when a particular change occurred. Furthermore, we have a record of previous entities that changed with the changed entity.

This chapter presents the C-REX extractor and the rationale for its design along with various issues and limitations that we faced during its implementation. C-REX [1] is an extractor which we developed to recover evolutionary information from source control repositories for software systems written in the C programming language. We also demon-

[1]Recovering the Evolution of Code Structure – RECS (REX).

strate some results based on running C-REX to recover the evolution of eleven open source systems.

3.1.1 Organization of Chapter

The chapter is organized as follows. In Section 3.2, we give an overview of evolutionary code extractors and discuss the decision choices we settled on as we built C-REX. In Section 3.3 we explain the complexities and challenges associated with building C-REX. In Section 3.4 we document the schema of the data recovered by C-REX. Section 3.5 gives an overview of the implementation of C-REX and highlights a number of techniques used to speed up the extraction process. Then in Section 3.6 we overview some of the limitations and possible improvements to our approach. Section 3.8 discusses related work. Section 3.7 presents results for using C-REX in practice to demonstrate the feasibility of our approach. In Section 3.9 we conclude the chapter with comments about evolutionary extractors and their benefits for studying software systems and validating our understanding of the development process of large software systems. We also present possible extensions and future directions for C-REX.

3.2 Evolutionary Code Extractors

In the previous chapter, we introduced the concept of an *evolutionary code extractor*. We showed that a change to source code can be described in a number of ways. Each one focuses on a particular characteristic of the source code at varying levels of details. For example, a change to a file can be recorded as a change to the lines of the file, to certain functions that reside in the file, or to dependencies between these functions.

For our purposes we call traditional extractors snapshot extractors. rigiparse [Hau88], CIA [CNR90], and CPPX [CPP] are examples of snapshot extractors. A **snapshot extractor** retrieves information about a

single version of the software system. It would produce facts such as function_1 *calls* function_2, or function_1 *uses* variable_1.

On the other hand an **evolutionary extractor** retrieves information about the history or evolution of a software system over the periods of its development. An evolutionary extractor determines the difference between consecutive snapshots of the software system and produces information such as function_1 *no longer calls* function_2, function_1 *now uses* variable_1.

C-REX is an evolutionary extractor which tracks the evolution of source code using the data stored in source control system.

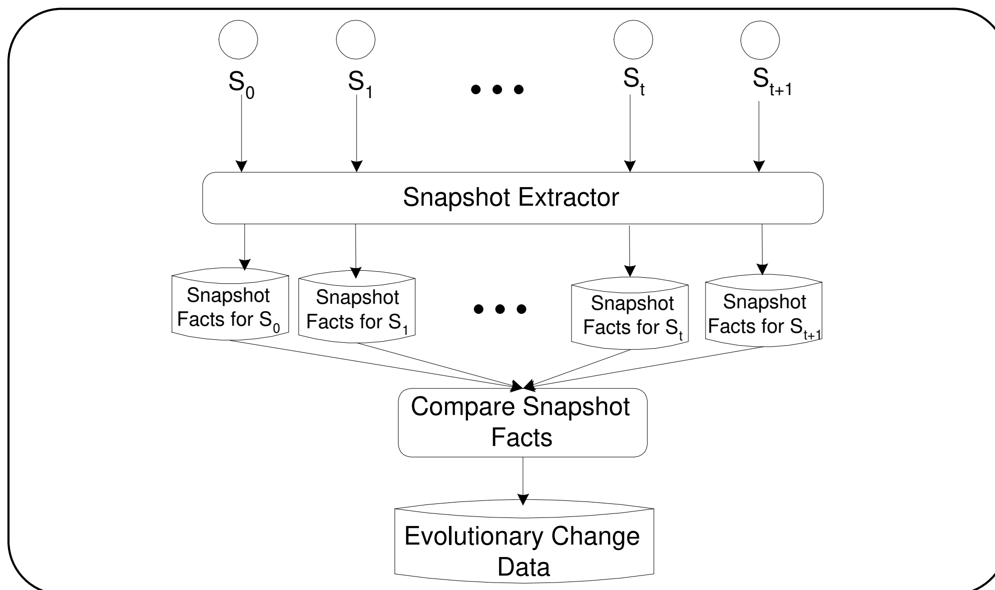

Figure 3.1: Conceptual View of the C-REX Extractors

The C-REX extractor can be thought of as the application of a traditional snapshot extractor on each snapshot of the source code. The traditional (non-evolutionary) code extractor generates information about the defined entities in the source code and the dependencies between them. For example, a snapshot extractor might report that in snapshot S_1 functions $main()$, $print()$, and $help()$ are defined. It might also report that

function $main()$ calls function $help()$ which calls function $print()$. The same snapshot extractor applied on snapshot S_2 might report that functions $main()$ and $help()$ are defined. It might also report that $main()$ calls $help()$. The details for each snapshot are stored in the corresponding snapshot facts database. Once the snapshot extractor processes every snapshot, we compare every two consecutive snapshots (as shown in Figure 3.1) to generate the *evolutionary change data*. Continuing our previous example, this snapshot extraction analysis would generate the following information when comparing snapshots S_1 and S_2: function $print()$ was removed and function $help()$ no longer calls function $print()$.

In the previous chapter, we discussed a number of design dimensions that must be examined before one proceeds to build an evolutionary extractor. Choices along these dimensions influence the type of information generated by an evolutionary extractor and influence the design of such an extractor. We now elaborate on our choices along these dimensions, and discuss the C-REX implementation challenges associated with these choices:

3.2.1 Frequency of Snapshots

This dimension deals with how frequently should the evolutionary extractor recover changes from the source code. Figure 3.2 shows the various possibility for the frequency of snapshots. The evolutionary extractor can compare and report differences between consecutive releases or builds. The extractor could report differences between consecutive code changes done by a developer. It can also track changes between a list of changes (*changelist*) that a developer performed to implement a particular feature, enhance it, or fix a bug.

For C-REX, we chose to track the source code at the changelist level. The choice to use the changelist level is due to the type of analysis we plan on performing on the extracted data, such as studying the change propagation phenomena.

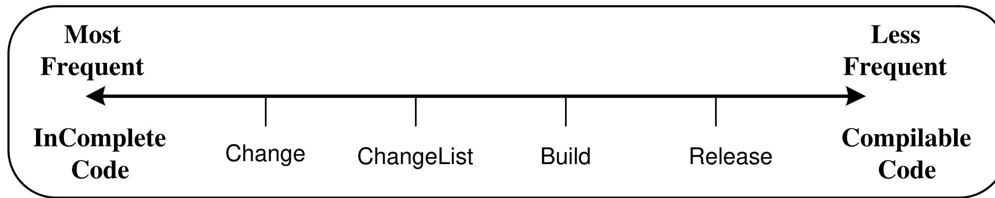

Figure 3.2: Source Code Snapshot Frequency Choices

Due to the large number of changes (hundreds of thousands of changes) that occur to the source code throughout its lifetime we had to develop several techniques to perform the extraction at this frequency level in a reasonable time. These techniques are presented in more detail in Section 3.5. For example, the conceptualization which assumes that each snapshot of the source code is analyzed fully then the facts for each snapshot are compared is not a practical one. Such a technique would take a long time to recover the evolution of a large long lived software system.

3.2.2 Data Source

This dimension deals with the data source used to acquire the snapshots. Modification records stored by source control systems are a good source of changelist information. As a software system evolves, there are many periods throughout its lifetime when the code stored in the source control repository is incomplete, *i.e.* not compilable or parsable. For example, a developer may add calls to a function that is not yet defined or remove the definition of a function without modifying all the functions which called it. Figure 3.2 illustrates that more frequent snapshots (such as changelists) are likely not to be compilable.

3.2.3 The Characteristics of Code

This dimension deals with the characteristics that are monitored for each snapshot of the code and are compared between consecutive snapshots. These characteristics can be metrics such as the size of the source code

or structural characteristics such as dependencies between source code entities.

C-REX monitors and outputs details about the structural changes to the source code. In particular, it tracks the addition, removal, and modification of source code entities, such as functions, macros, and variables. It also tracks the addition and removal of dependencies among these code entities. Using such recovered information, we can study the evolution of source code dependencies and reason about different types of maintenance changes such as refactoring and renaming. For example, if C-REX reports that in one changelist function *f()* is removed, function *newF()* has been added, and all callers to function *f()* have been updated to call function *newF()*, we can conclude that it is very likely that function *f()* has been renamed to *newF()* or replaced by *newF()*.

3.2.4 Level of Detail

This dimension deals with the level of detail for the information recovered by C-REX. The level of detail defines the granularity of code characteristics which are tracked and studied in the analysis of snapshots. Figure 3.3 shows the various levels of detail and points out that the complexity of analysis rises with increased details in the output.

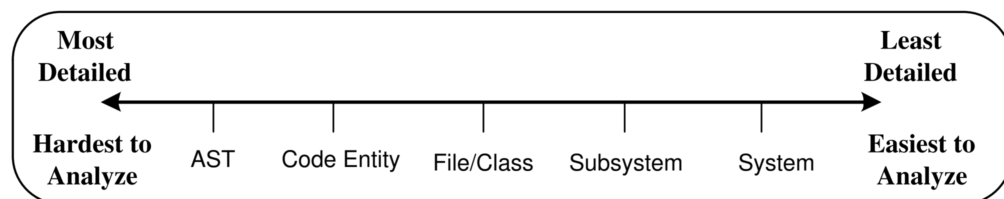

Figure 3.3: Level of Detail for Evolutionary Analysis

For example, an evolutionary extractor could track changes at the level of the whole system (such as changes to the size of the whole system), or it could instead pinpoint changes to the software system entities (changes to particular functions). To obtain the maximum amount of information about the evolution of a software system, we would prefer

that an evolutionary extractor recovers highly detailed information. This could entail tracking changes to the source code at the Abstract Syntax Tree (AST) level. Unfortunately, such an approach is complex to implement. An AST evolutionary analysis done per changelist snapshot needs to deal with extracting incomplete code. This requires many advanced techniques and heuristics to recover from potential errors in such code. Therefore to avoid dealing with such complexities, we chose to have C-REX generate data at the *Code Entity* level (see Section 3.4). This level of detail permits us to develop snapshot extractors which are more robust and which use simpler error recovery techniques than ones needed for AST level analysis.

With our choices along the design dimensions of an evolutionary extractor presented, we now deal with some of the challenges and complexities associated with building such an extractor.

3.3 Challenges In Developing C-REX

Analyzing source code, in particular the source code for legacy systems, is a difficult task. Performing the analysis using tools that are based on text book grammars is susceptible to failure as text book grammars fail to address the variety of dialects of programming languages and the multitude of proprietary compiler extensions. These tools are not capable of reliably processing the legacy source code. They require user intervention to modify the offending source code and restart the tool. For an evolutionary extractor such as C-REX, we need to parse hundreds of thousands of variants of the source code. Therefore C-REX should be able to parse legacy source code without failing, and it should recover in a graceful manner when it processes incomplete source code containing errors.

In the previous chapter, we listed challenges associated with developing evolutionary extractors. We now explain how C-REX deals with these challenges.

3.3.1 Robustness and Scalability of the Extractor

As we started to develop C-REX we decided that the robustness and scalability of C-REX are paramount for our research purposes, as we planned to analyze the evolutionary history of several large long lived software projects. Therefore, we needed an extractor which can process large amount of data and is able to recover gracefully from errors.

3.3.2 Accuracy of the Extracted Information

We acknowledged that ensuring the robustness of C-REX will affect our accuracy and we sought to minimize this negative outcome by focusing on the code entity detail level instead of focusing on the AST detail level.

3.3.3 The Changing and Unstable Nature of Source Code

As for the changing and unstable nature of source code over time, we decided to search for techniques that can parse incomplete source code. A number of approaches exist to analyze source code such as lexical analysis, robust parsing and island grammars [Ste03].

C-REX uses a lightweight extraction technique to robustly analyze incomplete source code for several snapshots. The used technique aims to only locate the start and end of each entity (such as functions, macros, and types) in the source code. By only focusing on the start and end of an entity, we can recover from non compilable code gracefully and still be able to analyze the rest of the file. Once we locate the start and end of each entity we can perform simple string matching to determine if a particular source code entity depends on another entity. For example, if a file contained two functions (*main* and *help*) and we located the start and end of each functions, we can determine the tokens which reside in both functions. Then we can easily determine if function *main* has a token in it named 'help' or if function *help* has a token in it named 'main'. Several enhancements are required to ensure that the results of this approach are

reasonably reliable. For example, we ignore tokens inside of strings and comment blocks.

3.3.4 Time Required to Develop the Extractor

To speed up the time required to develop C-REX, we adopted an open source program called `ctags` [CTA]. `ctags` is a source code tagger which is used by source code editors to parse the source code being edited. Using `ctags`, editors can offer rudimentary source code highlighting and can provide some navigational support for developers (such as moving between function definitions). We believe that the challenges associated with parsing source code while it is being modified by a developer inside an editor are similar to the challenges we have to tackle to parse source code which may not be compilable.

By adopting `ctags` we do not need to consider a large number of peculiarities of legacy source code. For example, we do not need to worry about ANSI and K&R C differences in the source code. `ctags` uses several heuristics to reliably parse source code containing `#if` preprocessor conditional constructs. `ctags` uses a conditional path selection heuristic to resolve complicated choices. It also employs a fall-back strategy when all heuristics fail.

`ctags` has been highly optimized and can parse and tag source code very quickly and efficiently. This is an important consideration given the amount of source code that we need to analyze. In addition, `ctags` offers support for over thirty languages. We hope in the future to use `ctags` multi-lingual support to extend C-REX to deal with repositories which contain source code written in programming languages other than C.

3.3.5 Additional Challenges

More challenges became clear to us as we started working on C-REX. These challenges are mainly due to the design choices we picked for C-

REX. In this subsection we present these additional challenges and discuss the techniques we used to overcome them.

3.3.5.1 Creation of ChangeLists

As C-REX uses changelist snapshots, we needed to get a record of all the changelists in source control modification records. A changelist groups several consecutive changes to a software system as part of a bigger focused change to implement or enhance a particular feature in the system, or fix a bug in it. For example, a developer might change four files to implement a specific feature. Then the changelist would contain details for changes in each of these four files, as the developer will submit all four changes simultaneously (or at least within a few minutes apart of each other) to the source control system. Some source control systems such as Perforce [Per] store the fact that these four changes are related whereas other source control systems such as CVS do not keep track of such information. Therefore for CVS we needed to develop heuristics to recreate this grouping from CVS modification records. An effective heuristic used by C-REX and others [GM03, ZW04] is to group changes in the same changelist if they occur within a short window of time by the same developer.

3.3.5.2 Enriching the Recovered Change Data

Source control systems store in each modification record important information other than just the changes to the source code. For each modification, the system stores:

1. The name of the developer who performed the change.
2. The time the change was performed.
3. An explanation message entered by the developer giving the reason for the change.

We decided to enrich our extracted data and attach this additional information to it. This additional information could be helpful in understanding legacy systems, as we have shown in our previous work (*e.g.* [HH03a, HH04e] – *see* Chapter 5).

To attach the additional information to the extracted data, we first perform heuristic analysis to determine the developer of a change. Although source control systems store the name of the developer who submitted a change to the repository, this name may not be the developer who actually performed the change. Instead in some cases, a limited number of developers are given access to submit changes to the source control system on behalf of other developers. This ensures that code changes are thoroughly reviewed and approved by senior developers before they are integrated into the software system. Luckily in many cases, the name of the actual author(s) is recorded in the explanation message attached to the change. Therefore, we analyzed these messages using several heuristics to locate all possible authors. For example, our heuristics searched for tokens such as the '@' symbol or words such "*Thanks to*" and "*Submitted By*" to locate possible acknowledgement fields and email addresses in the detailed messages. If all our heuristics fail to locate a possible developer then we simply attach to a change the name of the developer as recorded by the source control repository. We also employ heuristics, similar to [GM03], to deal with developers changing their email addresses over time. These heuristics permit us to map a seemingly new developer showing up in the change information to another developer that may have existed previously.

In addition to attaching the explanation message entered by the developer, we performed lexical analysis, similar to [MV00], in order to automatically classify changelists into three types based on the content of the explanation message – Fault Repairing, General Maintenance, and Feature Addition and Enhancement changelists. Such information is used in studying the reliability of a software system [HH03b] (*see* Chapter 8).

3.3.5.3 Performance

Conceptually (as shown in Figure 3.1), we would have to parse the source code of each snapshot. Thus for each change to the software system, we would need to re-parse the source code. This is not a reasonable approach as we would have to parse the source code for the software system a large number of times. Although we use `ctags` to perform a large part of our analysis, the analysis still requires some time and in a long lived software project we may have hundreds of thousands of changes that are stored in the software control repository. To enable the extraction to be done in reasonable time, we use an incremental technique which analyzes only the changed files. We then integrate the analysis results to produce evolutionary change data for the whole software system.

By adopting a mature source code tagger (`ctags`), avoiding the use of complex source code extractors, and focusing our extraction on changed files, we are able to drastically improve our performance. For a large system, such as NetBSD with around ten years of development, C-REX takes over sixteen hours to perform its extraction. This is due to the long history of the project, and the large size of its code base. For smaller systems such as Postgres, C-REX can perform its analysis in just over three hours. We believe this performance is acceptable given the amount of analysis required. Section 3.5 presents the implementation of C-REX and discusses several optimizations that we developed to robustly process large software systems in such a timely fashion.

	NetBSD	FreeBSD	OpenBSD	Postgres
CVS Repository	533MB	367MB	220MB	314MB
C-REX Ouput	96MB	67MB	45MB	11MB

Table 3.1: Size of Source Control Repository vs. Size of C-REX XML file

The evolutionary change data produced by C-REX is represented as an XML file. Table 3.1 summarizes a few of the analyzed systems. It shows that the XML output file is small in comparison to the size of the analyzed

CVS repository. This small size allows the output file to be loaded in memory for fast processing by other tools.

3.4 Schema For The C-Rex Evolutionary Change Data

In the preceding two sections, we outlined the need for an evolutionary extractor, proposed the development of C-REX, specified the type of information C-REX should record about source code changes, and presented additional information which C-REX attaches to the recovered change data.

As we stated above, we store the output of C-REX in XML format. Others may choose to store such output in a database and provide developer APIs to access the output. We decided that the overheard of using SQL in the extraction and analysis steps is too high. We found that a large amount of our analysis involves loading all the data into memory and processing it. Such operation is equivalent to a full table scan, which is inefficient in modern database systems. Furthermore storing the output in a database would require the users of C-REX to perform additional setup work to install a database and configure it. The XML file can be easily processed by any tool without relying on specialized APIs.

The rest of this section presents the structure of the recovered change data stored in the XML file.

Figures 3.4 and 3.5 give the schema for the C-REX output. The schema is divided over two figures to make it easier to read. Figure 3.4 illustrates that each *Change* to the source code has a *Change Type*. The *Change Type* indicates if a *ChangeUnit* was modified, added, or removed. A *ChangeUnit*, as illustrated in Figure 3.5 could be a source code *Entity*, such as a *Function*, a *Data Type*, a *Global Var*, or a *Macro*. A *ChangeUnit* could also be a *Dependency* between source code *Entities*, a *Comment*, or a *Control Structure*. C-REX records the addition or removal of an *if/else* clause, a

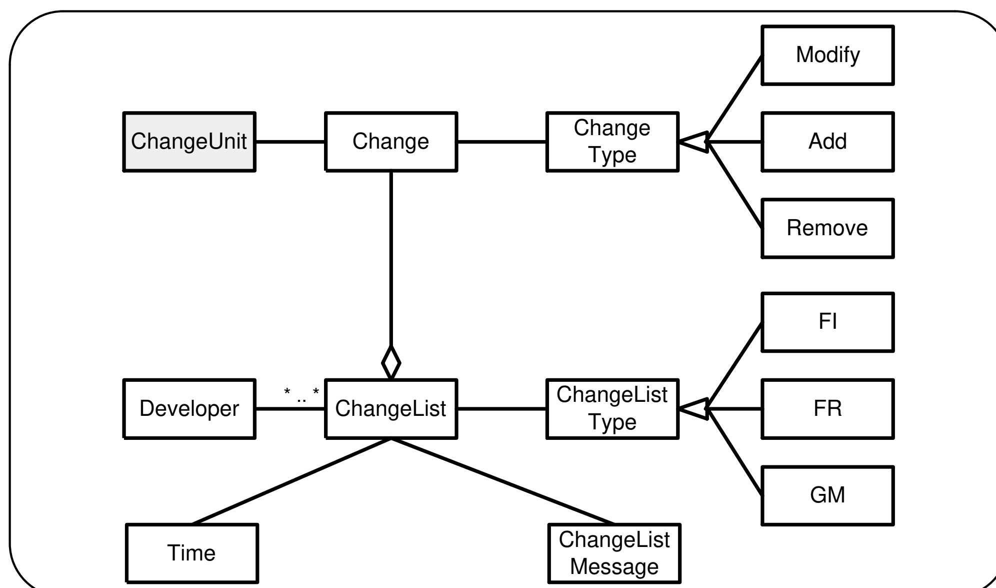

Figure 3.4: Schema for the Change Data Extracted By C-REX

case clause, a *while* statement, or a *for* statement inside of a function. In contrast to an AST level schema, this schema simply records the addition and removal of these control structures, it does not record changes to dependencies inside of these structures. Changes to source code dependencies are tracked only for source entities such as functions.

Each *ChangeUnit* has a *ChangeUnit Location*. For example, a *Dependency* and a *Comment* is located in an *Entity*, whereas an *Entity* is located in a *File*.

Figure 3.4 shows that each *ChangeList* has a number of *Changes* in it. A *ChangeList* has a *Developer* who performed the changes. A *ChangeList* has a *ChangeList Type* which could be a *FI*, *FR*, or *GM* changelist as explained in the previous section. Also for each *ChangeList* we record the *Time* it occurred and the *ChangeList Message* – a message explaining the reason for the change.

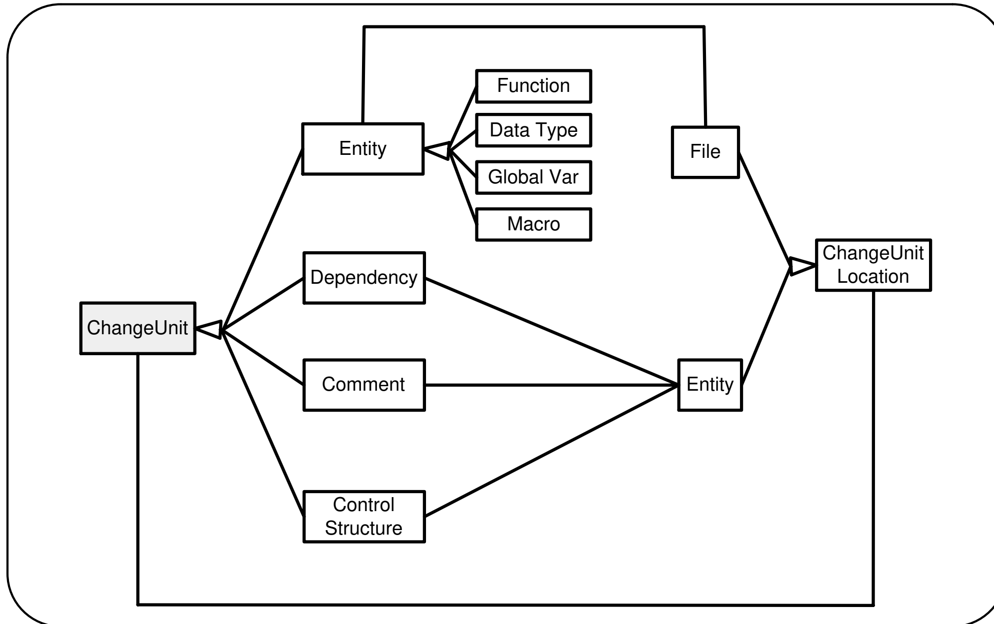

Figure 3.5: Schema for the ChangeUnit

3.5 The Implementation Of C-REX

We now present the implementation of the C-REX extractor. This implementation was influenced by the requirements and challenges outlined in the previous sections.

Our presentation uses an example of a simple software system (shown in Figure 3.6). The software system contains only one file ($main.c$). File $main.c$ has been modified only once. Therefore we have three versions of $main.c$ (rev 0 – initial empty revision, rev 1, and rev 2 – after the modification).

To recover data in the format described in Section 3.4, C-REX performs six steps of iterative analysis for the data and source code stored in the source control repository. In the following subsections, we describe each of these steps.

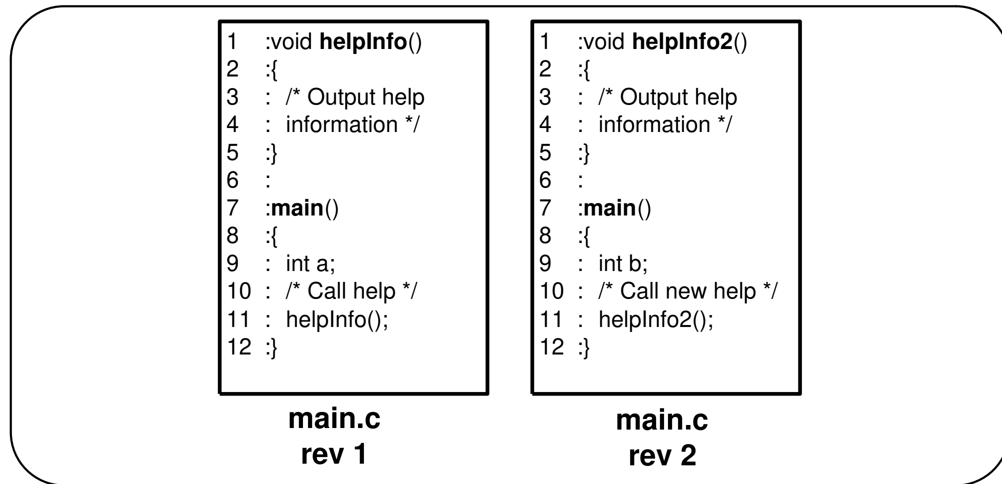

Figure 3.6: Example of a Simple Software System

3.5.0.4 Step 1 – Revision Data Extraction

In the first step, C-REX determines the number of revisions of each file in the software system. For our simple example, C-REX would determine that file $main.c$ has three revisions. CREX stores additional information associated with a revision such as the name of the developer who performed the change and the time the change was done into the *Revision Data* Database.

3.5.0.5 Step 2 – Entity Extraction

In the second step, C-REX identifies all entities, such as functions and data types for each revision of each file. This is done using `ctags` to identify the beginning of a code entity. Then a Perl script is used to locate the end of the entity. At the end of this step:

a. We have a record of each entity ever defined in the lifetime of a project. The record of these entities is analogous to the symbol table used by a compiler when it builds a software system. In contrast to a traditional compiler symbol table, this *Historical Symbol Table* has

all the symbols (entities) that were ever defined in the project's lifetime, not just the symbols defined in a particular compilation. For our simple example, C-REX would produce a historical symbol table that contains the following symbols: *helpInfo*, *main*, and *helpInfo2*.

b. For each revision of an entity in each file, we record its contents (its contained tokens) along with the entity's beginning and ending line. We call this the *Entity Revision Map (ERM)*. For instance in our simple example (see in Figure 3.6), C-REX would determine that the first revision of function $main$ starts at line 8 and ends at line 12 in the first revision of file $main.c$. C-REX also records the contents of $main$ (the tokens from line 8 to 12).

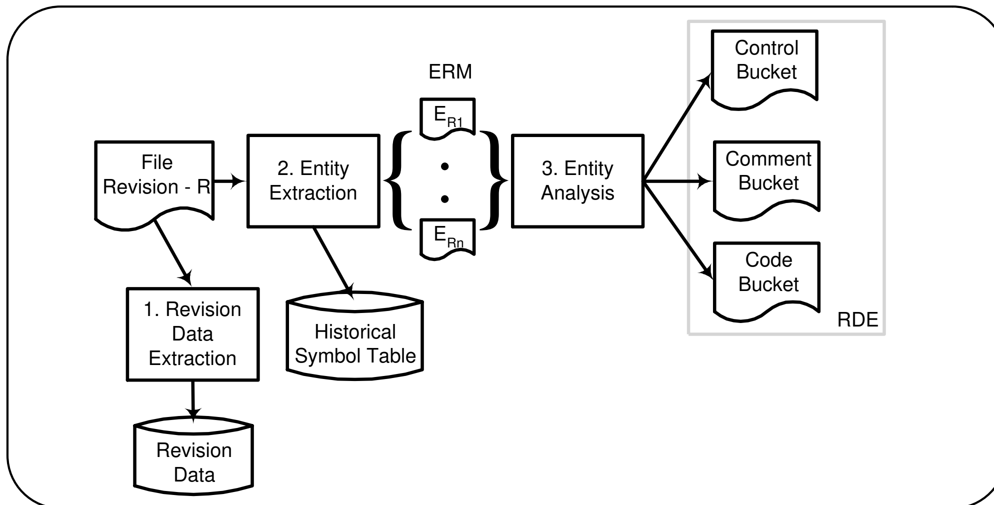

Figure 3.7: Steps Needed to Generate the RDE

3.5.0.6 Step 3 – Entity Analysis

In the third step, we analyze the ERM. For each revision (R) of an entity (E) – E_R, we divide its content into three token buckets. Each bucket contains the tokens and the number of occurrences of each token:

a. The *code bucket* contains all code tokens ($i.e.$ non-commented tokens) in E_R.

b. The *comment bucket* contains all tokens that reside inside of comments in E_R.

c. The C *control bucket* contains all the C-control keywords that are in E_R, such as *for*, *if*, *else*, *while*, and *do*.

We store the results of this step in a Revisions Database for Entities (RDE). The RDE contains three buckets for each revision of an entity: control, code, and comment. These last three steps are shown in Figure 3.7. Figure 3.8 illustrates the contents of the buckets for the revisions 1 and 2 of function $main$ at the end of the third step.

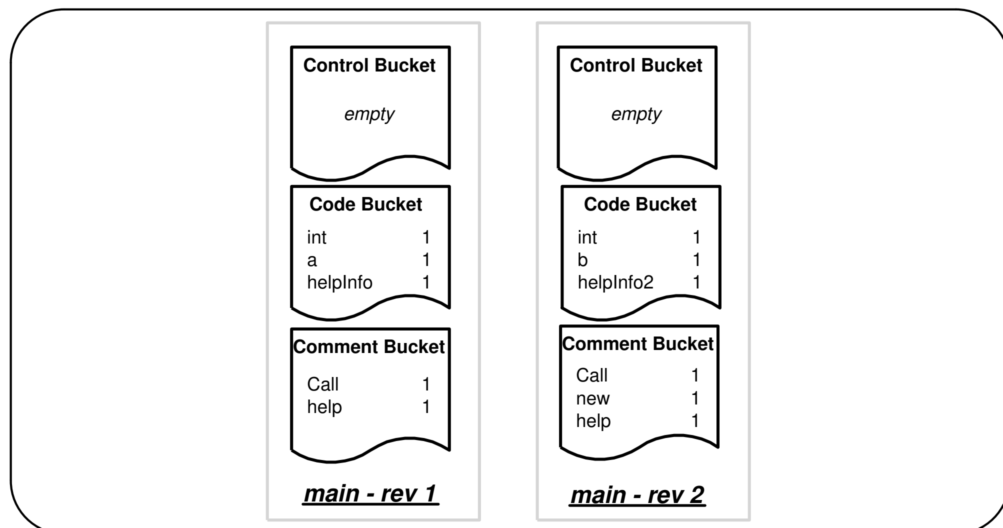

Figure 3.8: Contents of the Buckets for Entity $main$ in the Simple Software System

3.5.0.7 Step 4 – Token Change Analysis

The following three steps are used to generate the *Evolutionary Change Data* and are illustrated in Figure 3.9. In this step, we study the content

of the code, comment and control buckets stored in the RDE. We compare the content of these buckets for each two consecutive revisions of an entity in a file, $i.e.$ R_T and R_{T+1}. The differences are recorded and written to the Difference RDE (*D-RDE*) database. For each pair of consecutive revisions of a file, we have three buckets that store the differences between the pair's code, comment, and control buckets.

For our simple example, looking at the $main$ entity we would have the following information in each bucket, when comparing revisions 1 and 2:

1. **Control Bucket**: empty.
2. **Code Bucket**: helpInfo -1, helpInfo2 1, a -1, b 1. This indicates that code tokens *helpInfo* and *a* no longer exist and that code tokens *helpInfo2* and *b* have been added in revision two.
3. **Comment Bucket**: new 1. This indicates that the comment token *new* has been added in revision two.

3.5.0.8 Step 5 – Dependency Change Analysis

In this step, we examine the code buckets of the D-RDE to recover code dependency relations between entities. We compare the tokens in the code buckets for each pair of consecutive revisions of a file to the contents of the *Historical Symbol Table* generated in second step (*see* Figure 3.7). We remove all tokens from the code buckets that do not exist in the *Historical Symbol Table*. We call the code buckets after this step *Dependency Buckets*. For our simple example, we would remove tokens *a* and *b* as they do not exist in the Historical Symbol Table, which contains the tokens $main$, $helpInfo$ and $helpInfo2$. We now know that from revision 1 to revision 2, entity $main$ *no longer* depends on entity $helpInfo$, instead it *now* depends on entity $helpInfo2$.

The technique used in this step ensures that we do not miss dependencies where an entity is used before it was defined in a later change. For

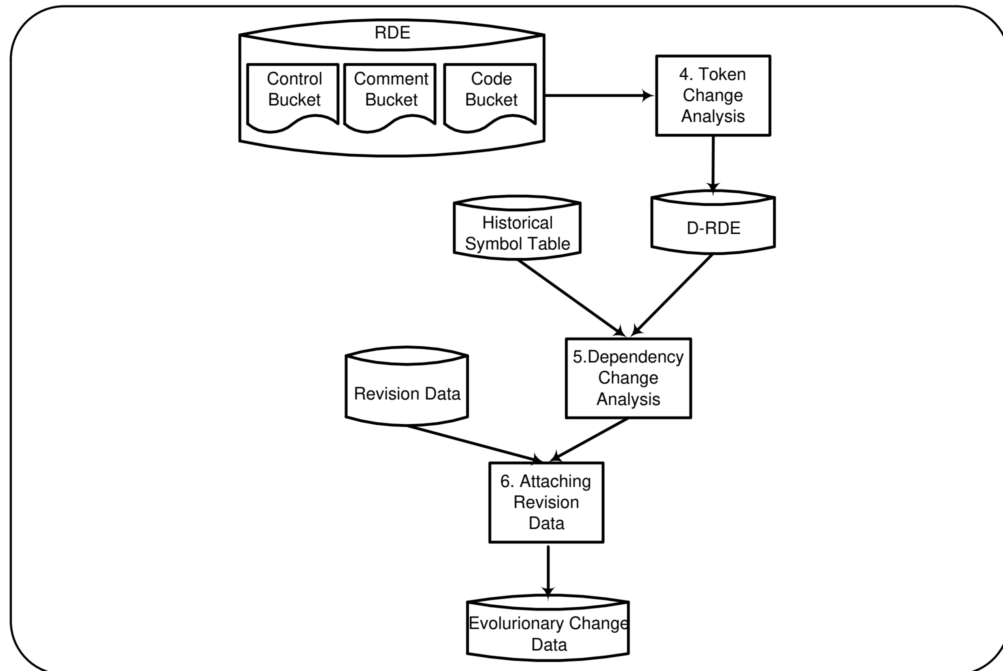

Figure 3.9: Steps Needed to Generate the Evolutionary Change Data

example, if version 1 of $main$ called $helpInfo2$, our approach would recognize that $helpInfo2$ is a function that will later be defined, as $helpInfo2$ would exist in the Historical Symbol Table. The technique also tracks the number of times a dependency exists between entities as it uses the token occurrence counts stored in the code buckets.

3.5.0.9 Step 6 – Attaching Revision Data

The output of the previous step contains a record of changes throughout the lifetime of a project and their effects on the source code dependencies, the comments, and the control keywords. It is still missing additional information stored in the source control system such as the name of the developer who performed the change, the changelist each change is part of, and other important information.

In this final step, we combine the D-RDE with the *Revision Data* Database created in the first step of the extraction. The output of this extraction process is stored in the XML *Evolutionary Change Data* file.

3.5.1 Performance

The sequence of steps described above are a simplification of the actual technique used. In particular, disk access is minimized. The different steps try to pass data through memory and preprocess data for later steps, instead of writing data to disk then rereading it again. The approach still takes as much as 16 hours for large projects with over 10 years of development. To further speed up this extraction process, three optimizations are possible.

First, the disk of the system on which the extraction is being done can use a RAID configuration to speed up the reading of the data considerably, as data is striped across different drives. We have used this technique in some of our extractions and the improvements in speed were substantial around 20-30% reduction in extraction time for large systems.

Second, the extraction process as described is done sequentially file by file. This does not need to be the case. Instead files can be analyzed concurrently on different machines. Each machine can populate the code, comment, and control buckets for its entities in parallel. The results could be combined later for further processing. The bucket generation (RDE building in step 3) is the most time consuming step in the extraction process.

Finally, the C-REX system described in this paper has been used to study software systems stored in CVS. To gain access to each revision of a file, we use APIs provided by CVS. By using these APIs we are able to later extend the C-REX extractor to other source control systems, for example we are currently working on adding support for ClearCase to investigate some commercial software systems. Unfortunately, these APIs require many disk I/O operations. Alternatively, we could load up the

CVS database for a file in memory and operate on its content in memory. This is a much faster alternative but it limits the extensibility of the extractor to other source control systems and requires good knowledge of the internals of the source control repository.

We believe that performance of our current C-REX implementation is reasonable for recovering evolutionary change information about large software systems. Nevertheless we believe that the implementation could be optimized further to permit us to perform more detailed tracking of changes throughout the lifetime of long lived software projects.

3.6 Limitations Of The C-REX Approach

Based on our use of the approach as described in Section 3.5 to recover the historical evolution of several large software systems, we discovered a number of limitations to our approach. In this section, we discuss these limitations and propose techniques to overcome them.

3.6.1 Dependency Analysis

The dependency analysis technique used by the C-REX extractor assumes that all files in the source control are part of the same executable. This is not the case for many software systems, such as multi-platform operating systems, which may support several hardware platforms and provide several implementations for the same function for each supported platform. The determination of which function to use is done using makefiles at compile time by setting some flags and using C preprocessor directives. Our approach does not analyze makefiles to determine the values of these flags and may analyze code blocks that are never executed or code blocks that are executed in some configurations but not in others. This is a limitation of our approach but this limitation exists for a large number of traditional code extractors which do not track the build process as part

of the extraction steps [TG01]. It may be possible to reduce this limitation by getting the user to assign specific files/directories to analyze their history instead of analyzing all source code stored in the source control repository. Also the user can define flags to determine if blocks of code should be analyzed or if they should be ignored.

Furthermore, the use of the historical symbol table may cause the creation of incorrect dependencies. For example, suppose a function uses a local variable then several years later in the future a global variable with the same name is defined. This will create an incorrect dependency. To help overcome this limitation we can ensure that dependencies are created only to entities which have already been defined before the source code of an entity is changed. Unfortunately, this will prevent us from detecting situations where an entity is used before it was defined as the developer has not written the code for it yet. To solve this problem, we could adopt a time window approach. A dependency to another entity is created only if that entity has been defined before the entity being analyzed or after its definition within some time-period – a day or two seem to be a reasonable time windows. Currently C-REX does not use a time window approach but an analysis of the output of C-REX to detect the usage of entities before their declaration by two days shows that such situations are luckily almost non existent.

3.6.2 Beyond C

We believe the technique used by C-REX, in particular the dependency change analysis algorithm, can be extended to any procedural programming language. Unfortunately, this technique does not scale directly to object oriented languages and would require more elaborate analysis to deal with object oriented language peculiarities such as polymorphism.

3.6.3 Beyond CVS

Currently C-REX only supports the analysis of data stored in CVS source control repositories. We would like to extend it to support other source control systems as well.

3.6.4 More Detailed Change Tracking

Using the extracted data in our analysis, we discovered that we lacked a piece of useful information about the evolutionary changes in a software system. In many cases, we needed to determine if the signature of a function was changed. For example, it would be useful to know if a new parameter was added or the return type was changed. The current C-REX output does not offer such information. We have extended the C-REX output to support tracking of function signature changes. We expect to add tracking for additional change information as we try to understand the evolutionary process followed by software systems. Moreover, C-REX supports analysis for only one branch in the source control repository at a time instead of analyzing all branches simultaneously.

3.6.5 The Use of Heuristics

C-REX uses heuristics to create changelists as some source control systems (*e.g.* CVS) do not store changelists. Other source control systems, that record the content of each changelist, such as Perforce or ClearCase will not require such heuristics. The technique used to create changelists is similar to techniques used by others [ZW04, Ger04b]. We have manually investigated the quality of our changelist recovery technique by inspecting by hand a random number of changelists and the corresponding code change from the Postgres CVS repository. The performance is very good, none of the inspected changelists had incorrect or missed changed entities. Nevertheless we are considering adding a confidence rating to

each created changelist to assist users of the extracted data in determining the quality of our extraction.

We have as well investigated manually the quality of our heuristics to determine the authors of a change and found it to be able to perform extremely well (over 95% of authors determined by the C-REX heuristics were the correct authors for the Postgres repository). Furthermore, we conducted a study with commercial developers to determine the quality of our lexical based classification of changes. The study showed that our automated classifications match closely the classifications done manually by developers [HH04d] (*see* Chapter 4).

3.7 Using C-REX In Practice

Application Name	Application Type	Start Date	End Date	ChangeLists
NetBSD	Operating System	Mar 93	Jan 03	38,391
FreeBSD	Operating System	Jun 93	Dec 02	26,178
OpenBSD	Operating System	Oct 95	Jan 03	14,147
Postgres	DBMS	Jul 96	Nov 02	6,199
GCC	C/C++ Compiler	May 91	Apr 99	8,602
Apache 2.0	Web Server	Jun 99	Dec 03	5,696
APR	Cross Platform Library	Aug 99	Dec 03	3,230
Ruby	Language Interpreter	Jan 98	Jan 04	2,494
GSL	Scientific Library	Jul 96	Sep 03	1,955
XCONQ	X Windows Game	Apr 99	Dec 03	935
Sylpheed	Mail Client	Jul 00	Dec 03	1,197

Table 3.2: Characteristics of the Guinea Pigs

C-REX is built to analyze large long lived projects in a timely fashion without manual intervention. We acquired local copies of the CVS repositories for several open source project (*Guinea Pigs*) to verify the scalability and feasibility of the C-REX approach. Using C-REX, we recovered the evolutionary history of the main CVS development branch of these

projects. These repositories permitted us to detect and fix a number of limitations in C-REX and improve it. Table 8.1 shows descriptive statistics for our *Guinea Pigs*. C-REX is able to analyze the largest repository (NetBSD) in around sixteen hours and produces a small size output as shown in Table 3.1. The output of C-REX has been used by us in previous studies to examine the change propagation phenomena [HH04c] and to assist developers in understanding large legacy systems [HH04e].

3.7.1 Acquiring Our Guinea Pigs

To acquire these guinea pigs for our studies, we used four different techniques:

1. We asked the developers of some projects for access to their source control repositories. We acquired local copies of the repositories to avoid taxing the project's servers during our analysis and development.

2. Several projects offered support for `CVSup` [CVSb] – a tool used to mirror source control repositories. For these projects, we used a CVSup client to acquire our own copy of the source code repository.

3. Other projects offered support for `rsync` [rsy] – a protocol used to mirror data repositories. We used an rsync client to mirror the source control repositories for these projects.

4. Finally for projects that offered none of the aforementioned options and their development team was inaccessible, we used `CVSsuck` – a tool which uses the CVS protocol itself to mirror the CVS repository. The CVS protocol which is not designed for mirroring, therefore CVSsuck is not efficient. We used CVSsuck as a last resort to acquire a repository due to its inefficiency.

3.8 Related Work

Other researchers have recovered data from source code repositories to explain and validate their ideas. In contrast to prior approaches which recover information at the line or file level, the C-REX approach performs a more detailed analysis by mapping changes to source code entities and dependencies between them. Furthermore C-REX performs its analysis for each change/changelist in the repository instead of performing the analysis for each build/release as done by most approaches. To ensure that such a detailed analysis can be done, we could not use traditional source code extractors and we had to develop techniques to analyze code that may not be compilable. We now review some of the prior recovery approaches.

In [GHJ98, GJK03], visualization techniques are used to show the historical logical coupling between files in a software system. To perform such studies, an evolutionary extractor that produced high level change data (*i.e.* at the file level) was used. A similar extractor was used by Graves *et al.* to show that the number of modifications to a file is a good predictor of the fault potential of the file [GKMS00]. German [Ger04b] and Liu [LS03] use a similar extractor to visualize changes at the file level for open source and student projects.

Chen *et al.* presented a case study for a source code searching tool that makes use of the explanation message associated with each change to the code [CCW+01]. The extractor associates the messages entered by developers to particular lines of code. These comments are used to index the source code and provide more accurate search results, when developers search for the location where specific features are implemented in the code.

Zimmermann *et al.* present an extractor which maps changed lines to their containing entity (such as functions) in the code [ZW04]. Their extractor is similar to our extractor in being able to map changes to a particular entity but it does not track other types of changes, such as changes to dependencies and comments. Maletic *et al.* present a system

which uses island grammars to analyze differences between non compilable code [MC04].

3.9 Conclusion

In this chapter, we presented a new type of source code extractor, namely, a source code extractor that goes beyond extracting facts from a single version of a software system and instead extracts facts and compares them across the whole evolutionary history of a project. We believe that such evolutionary extractors are important in advancing research in software engineering and empirical software evolution in particular by facilitating access to such evolutionary data buried inside source control systems.

This chapter gave an overview of evolutionary code extractors and presented C-REX, our implementation of such an extractor for the C programming language. We presented the schema for the data generated by C-REX and discussed the challenges and rationale associated with its implementation. We then explained the implementation of the extractor. Finally, we highlighted limitations of the extractor and proposed techniques to overcome them. C-REX and evolutionary extractors form the empirical basis for the research presented in this thesis.

In the following chapter, we examine the use of source control systems by professional software developers. We also examine the accuracy of the automated classification technique used by C-REX.

Appendix

We show brief results to demonstrate the usefulness of C-REX. Whereas previous evolution studies typically focused on monitoring either the LOC [Mic00] or the number of modules (files) [LRW+97], we can, thanks to C-REX, monitor evolution at the function or dependency level. For example, we can measure the degree of connectivity of the dependency graph in

comparison to the maximum possible connectivity on a quarterly basis. Such a metric acts as a better indicator of the evolution in complexity in the dependency graph than LOC or module counts [LRS01]. To measure the degree of connectivity, we divide the number of dependencies in a software system by the maximum number of possible dependencies [2].

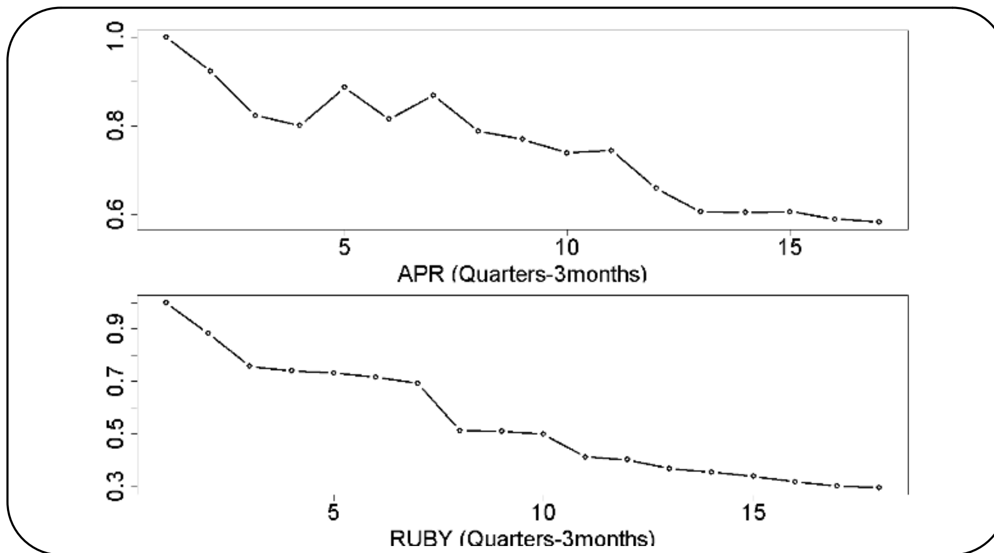

Figure 3.10: Complexity Ratio Evolution

We present graphs for only two systems (*see* Figure 3.10). Both graphs show the ratio[3] dropping over time while remaining stable over short periods of time with rare rises. The same pattern holds for all our guinea pigs. Investigating anomalies (*e.g.* rare rises) in these graphs reveal interesting events in the project's history. For example, the seesaw (between quarters 5 and 7) in the APR data corresponds to the period between an initial alpha release and the first beta release of the project. During this period, a large amount of refactoring and clean-up changes occurred, as determined from reading the change messages in CVS. Using an extractor which performs its analysis at the release level, such variations to the

[2]The square of the number of entities in the software system

[3]The shown ratio is standardized between the values 0 and 1 by dividing by the maximum ratio value for the studied periods.

value of the metric would not have been as visible, as the variations occurred between two releases. Furthermore, determining the reasons for variations would have required us to manually search through the source control system and other historical information from the project. Such manual search is not needed as C-REX already extracts and attaches to the dependency graph the messages describing changes within each quarter. For example, using C-REX we can investigate if refactoring changes are more likely to be done at the beginning of a release development cycle or if such changes are evenly distributed throughout the release cycle. The following parts of this thesis will investigate uses of the data recovered by C-REX.

CHAPTER 4

Source Control Change Messages: How Are They Used And What Do They Mean?

Source control systems permit developers to attach a free form message to every committed change. The content of these change messages is rarely investigated and little is known about their use by developers while they maintain their code.

We present the results of a survey we conducted with professional developers. The purpose of this survey was to investigate how developers make use of these messages and what type of information exists in them. We also investigated the possibility of using automated techniques which examine change messages and determine their purpose, for example that a change was done to fix a bug or to indent the code. We also asked developers to compare change messages in open source software systems to change messages in commercial systems.

The findings of our survey suggest that change messages are a valuable resource used by practitioners to maintain and manage software projects, for example to understand the code when they are fixing a bug. Moreover, change messages in open source projects are similar to messages that developers encounter in large projects. An automated approach to determine the purpose of a change using the change message is likely to produce results similar to a manual analysis performed by professional developers. Researchers should investigate techniques and approaches to improve the quality of the change messages and to make them more accessible for developers as they evolve software systems.

4.1 Introduction

SOURCE control systems such as CVS [CVSa], ClearCase [Cle], and Perforce [Per] are used nowadays by most large software projects to control and manage their source code [Roc75, Tic85]. As a software system evolves, changes to its code are stored in the source control repository. The repository of a source control system contains detailed information about the development history of a project. The repository stores the creation date of every file, its initial content and a record of every modification done to a file. A modification record stores the date of the modification, the name of the developer who performed the changes, the numbers of lines that were changed, the actual lines of code that were added or removed, and a *change message* entered by the developer usually explaining the reasons for the change.

In this chapter, we focus on these *change messages* which are attached to every modification record. Such messages have been rarely used by researchers to build tools or study approaches to assist developers in maintaining long lived software systems. Chen *et al.* presented a case study of a source code searching tool that makes use of these change messages [CCW+01]. The tool uses the messages to index the source code to provide more accurate search results, when developers search for the location

where specific features are implemented in the code. Mockus and Votta presented a lexical analysis technique to classify the type of a change based on the content of the change message [MV00]. This classification is then used to monitor the evolution of a software product and to gauge its reliability. A similar approach has been used by us in [HH03b] (*see* Chapter 8). We also mined change messages and attached them to the software architecture to assist developers in investigating the gaps between the actual architecture of the software system and the documented architecture that is rarely up to date [HH04e] (*see* Chapter 5). Perry *et al.* used change messages along with surveys of developers to gain a better understanding of the type of faults in large software systems and the efforts associated with fixing them [PS93, LPS02].

The aforementioned approaches have demonstrated the value of change messages in assisting developers. Yet, the quality of the change messages and the current use of change message by developers in industry have never been investigated. This chapter addresses these issues through a survey conducted in participation with six professional software developers. The responses to the survey were analyzed and studied thoroughly to arrive to our results.

We are interested in answering questions such as:

- Do developers usually enter meaningful and descriptive change messages? Are they likely to leave the messages empty?
- Do developers monitor such messages and react to their content?
- Do developers make use of these message as they maintain and enhance their software system, or are they ignored?
- Can we automatically determine the type of a change as being a bug fix or a feature?

With the widespread of open source software systems, their repositories have been used by researchers instead of relying on the repositories of

commercial software systems which are usually harder to acquire. For example, work by Chen *et al.* [CCW+01] and us [HH03b, HH04e] was conducted on open source systems; whereas work by Mockus and Votta [MV00] was conducted on industrial telephony systems. We investigated the differences between change messages entered by open source developers and change messages entered by developers of commercial software systems. By studying the differences between change messages in both types of systems, we can understand better the applicability of research findings to commercial software systems when the research is done using open source change messages.

4.1.1 Organization of Chapter

This chapter is organized as follows. In Section 4.2, we discuss our study design and explain the motivation behind its design and the approach used to develop our survey. Our study consisted of three main parts. A separate survey part was done for every part. In Section 4.3, we present the results of the first part of our survey and address the issues pertaining to change messages and their usage by software developers. Then in Section 4.4, we discuss the results for the second part of our survey, which focuses on the ability of an automated technique to accurately determine the purpose of a change by lexically examining the change message. In Section 4.5, we examine the results for the final part of our study which focuses on the differences between the change messages in open source systems and large industrial software systems. Finally in Section 4.6, we summarize our results, propose future research directions, and examine challenges based on our survey findings.

4.2 Study Logistics

In this section, we introduce the goals of our study. We then present our study participants and elaborate on the design of our study.

4.2.1 Study Goals

We would like to understand how developers use change messages in modification records as they maintain and enhance a software system. Given that such a message is optional, we want first to determine if developers are likely to enter meaningful and descriptive information in that message. Moreover, we want to examine the type of information that developers are likely to record in a change message. For example, are developers likely to specify the rationale for the change, alternative designs, or limitations of the current change.

Furthermore, we would like to determine if an automatic classification of a change message would agree with a manual classification performed by industrial developers. For many software projects, source code repositories are the only source of historical records about the project. Bug reports are commonly not archived. To perform studies to gauge the reliability of a software system, we can use the source code repositories (*e.g.* [HH03b]). We can use a lexical based approach, similar to [MV00], to classify modification records into three types based on the content of the change message. The classification approach would determine if a modification was done to fix a bug, to add features, or to perform general maintenance activities such as updating copyright notices or indenting the source code.

We would like to compare change messages in open source systems to change messages in commercial software systems. Due to the accessibility of open source code repositories, researchers are more likely to use such repositories in their studies. By comparing the difference between open source and commercial change messages, we can determine the suitability of using open source change messages in case studies and the applicability of any findings to commercial systems.

4.2.2 Study Participants

To perform our study, we used a survey technique. We surveyed experienced software developers to gain better insight into how professional software developers working on large industrial software systems are likely to write and use change messages. We picked a small number of participants which are accessible to us so we can easily interview them to clarify their survey replies if needed. We asked six software developers to participate in the survey. The developers worked in companies in the following software domains: security, telecommunication, graphics, and databases. The developers were chosen so they would represent two groups: an intermediate and a senior group. We hoped that this grouping would uncover if there are any noticeable variations in the survey responses that may be attributed to experience or managerial differences between both groups of developers:

- The first group consists of 3 intermediate software developers with at least 7 years of software development with no experience of managing other software developers.
- The second group consists of 3 experienced software developers with at least 9 years of experience developing industrial software systems and who have previously managed or are currently managing other software developers.

Table 4.1 gives background information about the developers participating in our study. At the time of the study, the developers worked at a number of different companies. All the developers had used source control systems for most of their professional career, this is likely due to the fact that source control systems are widely used and adopted in industry. The participants, where source control was not used for part of their professional career, were asked to elaborate on the reasons. Their replies indicated that this occurred in their earliest jobs and it was usually because they worked as part of a small group ("*we were a 2 man shop*"),

Dev. #	Development Experience (years)	Source Control Experience (years)	Avg. Team Size	Team Lead
I1	7	5	5	No
I2	7	7	5	No
I3	7	7	30	No
S1	9	5	5	Yes
S2	15	12	8	Yes
S3	9	9	5	Yes

Table 4.1: Characteristics of the Participants of the Study

or because they had a strict gatekeeper person who reviewed each every change ("*we had a dictatorship*"). Participants added that looking back they regret not using a source control system even though they were part of a small group. They recounted cases where they lost work due to erasing their source code by mistake and not using source control systems – "*I accidently erased all my stuff once at ... because I was trying out UNIX's recursive file removing feature :) My supervisor was not impressed.*". In short, even though the size of the development group size may be small and using source control systems may not be needed to coordinate development, professional developers think it is still beneficial to adopt a source control system as it offers a safety net that can protect developers from losing their work.

4.2.3 Study Design

Since the developers participating in the survey were easily accessible to us, we broke our survey into 3 major parts. Every part was given to the developers independently of the other parts. We hope that this technique encouraged the developers to focus on answering the questions in the current part of the survey without being influenced by other parts of the survey. The parts of the survey were given to the developers over a period of two months. Once a part was completed, the following part was given to the developers within 2-3 weeks.

The survey consisted of the following three parts (*see* this chapter's appendix for the full survey):

Part 1 - Usage and Content of Change Messages: In this part, the developers' background was assessed and they were asked to answer a number of questions about source control change messages. Every question in this part was formulated in such a way that its answer was a number from 0 (rarely) to 10 (most of the time). For example, one question asked the following: "*Q1. When you commit a change to a source control system, how often do you enter a change message?*". Developers were encouraged to add remarks of interest for every question. Also, for questions where several alternatives were listed an additional "*other*" alternative was listed to encourage developers to add additional reasons that may have been missed by us. For example, when asked "*How often do you use/read previous change messages when you are: (a) Doing Design, (b)Writing new Code, ...*" a final item was "*(k) Other, Specify*". We chose to use a scale of 0 to 10 instead of a smaller scale such as a 5 or 7 point scale, which are usually used in opinion surveys (e.g. like vs. dislike), since questions in our survey ask developers for time frequency estimates. A time frequency is easier to map to our chosen scale. In a 0 to 10 scale, a 100% of the time maps well to 10 and 80% of the time maps well to 8, that is not the case for smaller scales.

Part 2 - Classification of Changes: In this part, a list of 18 change messages from several open source projects were presented to every developer. Every developer was asked to allocate 10 points to four categories. Three categories represented the possible purpose of a change: bug fixing, feature introduction, and general maintenance. A fourth category was "Not Sure" – Developers were asked to use this category when the change message did not have sufficient information for them to confidently classify the change into one of the other three categories. We limited the number of change messages in the survey to 18 messages so that the professional software de-

velopers would finish the survey and classification in a timely and accurate fashion without interfering with their busy schedules.

Application Name	Application Type	Start Date	Programming Language
NetBSD	OS	March 1993	C
FreeBSD	OS	June 1993	C
OpenBSD	OS	Oct 1995	C
Postgres	DBMS	July 1996	C
KDE	Windowing System	April 1997	C++
Koffice	Productivity Suite	April 1998	C++

Table 4.2: Summary of the Studied Systems

The 18 change messages were selected from the repositories of six large open source systems (NetBSD, FreeBSD, OpenBSD, Postgres, KDE, Koffice - Table 8.1 lists details of these projects). Every change message in these repositories is already classified as either a bug fixing, feature introduction or general maintenance change using an automated classifier described in Section 4.4. We randomly picked 18 modifications from the repository of every project: 6 bug fixing, 6 feature introduction, and 6 general maintenance modifications (for a total of 108 changes). We then randomly chose half of these modifications (54 changes) and broke them into three disjoint sets of 18 modifications. We followed this selection procedure for modification to guarantee that the mathematical analysis performed later does not suffer from any bias resulting from the type of the change messages or their sources. Every set was classified by a member of the intermediate group and a member of the senior group. Each group classified the three modifications sets. No two developers in the same group classified the same set of modifications.

Part 3 - Comparing Change Messages: In this part, developers were asked to compare the change messages from open source projects

that they had classified in part 2 to the change messages that they usually encounter at work. They were asked to point out main differences between open source change messages and commercial ones.

4.2.4 Survey Design

To ensure that the survey questions were meaningful and not confusing to developers, we used the following technique in creating the survey questions:

- Questions were formulated and successively refined by us into a preliminary versions of the questions.
- This preliminary version of the questions was then presented to an intermediate professional developer who was asked to answer the survey. This developer is not one of the six developers and his replies were not used in our analysis. This developer's replies were examined and the developer was interviewed to determine if the questions were clear. Comments by this developer were incorporated into the final version of the survey which was given to the participating developers.

In the following sections, we analyze the results of the three parts of the survey. Our analysis focuses on finding if there are major trends in the developers' replies that would indicate with high confidence that a particular property is true. Furthermore, we examine if there are variations between the replies of the intermediate developers group versus the replies of the senior developers group.

4.3 Results For Part 1: Usage and Content of Change Messages

The first part of the survey focused on the content of change messages and how developers make use of such information as they maintain a

software system. In the following paragraphs, we give the purpose of every question and discuss the replies of the participating developers. All questions in this part were to be answered by giving a reply from 0 to 10 with a scale of 0 (rarely) to 10 (most of the time). We present the average of the replies of the participants. We expand on the cases when the average for the senior group differs from the intermediate group or when the reply of a particular developer differs a lot from the rest of the replies.

Q1. **Purpose:** To determine how often developers are likely to enter a change message when submitting a change to the source control system. **Results:** All developers, except one senior developer, indicated that they enter a change message most of the time (9 out of 10 is the average reply). When that senior developer was asked about the reason for not always entering a change message, he indicated that he does not enter change messages during the early parts of a project when he is developing new code in his own private code branch as the code is not yet stable and he is mainly using the source control system for backup. Yet, when maintaining source code, he always enters a change message. This reply highlighted two styles for using source control systems: one for backup and another for coordinating development in a large team.

Q2. **Purpose:** To determine how often developers monitor changes to the code base they are working on. **Results:** Results indicate that intermediate developers rarely (1.7 out of 10) monitor changes by others. In contrast, senior developers monitor changes most of the time (7.6 out of 10). We believe this is due to the managerial role of senior developers who use source control changes to keep up with the progress of the project. This hypothesis is supported by the replies to the third question of our survey as discussed in the following paragraph.

Q3. **Purpose:** To determine the main reasons for which developers monitor the changes. **Results:** The replies by the intermediate developers showed that they rarely monitored changes. These findings match the replies in the previous question where intermediate developers indicate

that they rarely monitor changes. As for senior developers, we found that they mainly use source control messages to keep abreast of the project progress and of changes to the APIs. They also use change messages to monitor the validity of changes. A senior developer pointed out that he monitors changes to the source code to ensure that no one is changing code that he/she should not be changing, indicating that source control systems are in some cases used as guards to enforce code ownership policies.

Q4. **Purpose:** To determine the suitability and clarity of the change messages. In particular, we were interested to know if developers felt that change messages were meaningful and useful in assisting them understand changes to the code and design of a software system. **Results:** The intermediate developers indicated that change messages tend to be useful most of the time (7 out of 10). In contrast, senior developers did not feel that these messages were that useful (2.6 out of 10). We believe the variation may be due to the fact that senior developers are more likely to monitor a larger number of changes to source code for which they are not as familiar. Moreover, the change message may rarely describe effects of a change at the high level big picture (the focus of a manager), but usually provide detailed low level explanations (the focus of a developer).

Q5. **Purpose:** To gauge the likelihood of developers reading the code associated with a change. **Results:** The replies indicate that both senior and intermediate developers sometimes (6.3 out of 10) read the code.

Q6. **Purpose:** To investigate the reaction of developers when an empty change message is attached to a code change. **Results:** The results of the survey indicate that senior developers (8.6 out of 10) are likely to examine the code. In contrast, intermediate developers would sometimes (5.2 out of 10) examine the code when the change message is empty. We believe this variation is due to the tendency of senior developers to use source control systems to monitor the progress of a project and enforce code ownership and quality policies.

Q7. **Purpose:** To examine the circumstances associated with a change

that cause a developer to investigate the code associated with a change.
Results: The results of the survey indicate that both intermediate and senior developers will most of the time (10 out of 10) examine the changed code when the change prevents their code from compiling. Both groups are likely (8 out of 10) to examine the changed code if it was in a subsystem that they recently worked on, or if the changed code was in a subsystem they depend on. The survey results show that developers are likely (7 out of 10) to examine code changes close to a release date. This results indicates that changes to the source code performed close to a release date are more likely to be examined than other changes. This may be due to the criticality of such last minute changes and the fact that developers would like to ensure that these last minute changes do not introduce bugs to their code which may depend on the recently changed code. Another interesting finding was that senior developer examine most of the time (9 out of 10) the changed code when a change was done by particular developers. This is due to the knowledge acquired by senior developers about the quality of the code produced by specific team members. This could be considered as a very primitive and intuitive quality monitoring technique. We were surprised to find that developers are not as concerned (2.8 out of 10) with examining changes when a large number of files are changed together as part of the same modification record. This may be due to the fact that such changes are usually simple changes that are well documented in the change message such as updating the copyright information in all files. Also it may be due to the unwillingness for developers to spend a large amount of the time investigating such large changes. The tendency of developers to voluntarily review small changes is encouraging as it shows that code is being peer reviewed. On the other hand the unwillingness of developers to voluntarily review large changes (as they may be too time consuming) is alarming, as large changes are likely to introduce bugs [GKMS00]. Formal change review process must be instilled in a development organization instead of simply relying on voluntary reviews.

Q8. **Purpose:** To determine the type of information most likely to ex-

ist in a change message. **Results:** The participating developers indicated that the rationale for the change is usually the most (6 out of 10) found type of information in a change message. Indication of the limitations of the change or alternatives are rarely found in the change message.

Q9. **Purpose:** To understand when developers are likely to use old change messages. **Results:** The results of the survey indicate that change messages are used most of the time (7.8 out of 10) during code reviews and code integration between different source control branches. They are also used (6 out of 10) to understand old code and during bug fixing.

We can summarize the results of the first part of the survey as follows. Developers will most of the time enter a change message. Whereas intermediate developers are not concerned about empty messages, senior developers tend to examine the code associated with such changes closely. Change messages are used by senior developers to enforce code ownership, and to gauge the quality of a change. Change messages are used by all developers during software maintenance and integration as the change messages are likely to specify the rationale for the changes.

4.4 Results For Part 2: Classification of Changes

In the second part of the survey, we were concerned with the feasibility of automatically classifying changes using the content of the change message attached to modification records in open source software systems. Results by Mockus and Votta show that 61% of the time, their automatic classifier and the developer who performed the change agree on the classification of changes to a large commercial telephony systems [MV00].

For our study, we developed an automated classifier program that reads every change message and classifies its modification record as one of the following three types:

Fault Repairing modifications (FR): These are the modifications which are done to fix a bug. Our automated classifier labels all

modifications which contain terms such as *bug*, *fix*, or *repair* in the change message as FR modifications.

General Maintenance modifications (GM): These are modifications that are mainly bookkeeping modifications and do not reflect the implementation of a particular feature. Examples of such modifications are updates to the copyright notice at the top of source files, re-indentation of the source code by means of a code beautifier (pretty-printer). Our automated classifier labels all modifications which contain terms such as *copyright/update*, *pretty/print*, or *indent/code* in the change message as GM modifications.

Feature Introduction modifications (FI): These are the modifications that are done to add or to enhance features. Our automated classifier labels all modifications that are not FR or GM modifications as FI modifications.

Every participating developer was shown the message associated with a modification and asked to allocate a total of 10 points to four categories. Three of the categories mirrored the automated classification categories (FR, GM, and FI). A fourth category was "Not Sure" (NS). Developers were asked to use the NS category when the modification message did not have sufficient information for them to confidently classify the modification into one of the other three categories. For the senior developer group only one out of 54 modifications was ranked as NS. For the intermediate developer group, out of 54 modifications three modifications were ranked as such. For our analysis, we considered modifications classified as NS to be FI modifications. The automated classifier uses FI as the default classification when it cannot determine the type of a modification; therefore, we chose to use the same default rule for the manual classification done by developers.

Developers had the option to allocate points between different categories but our automated classifier only assigns a single category to a modification. We chose to classify modifications based on the highest

ranked category, to permit us to compare manual classifications to the automated ones. When there were ties, we used the following tie breaking priority order: FR, GM, then FI. For example, if a developer allocated 5 points to the FR category and 5 points to the FI category, we would consider this modifications to a be an FR modification. This tie breaking priority order was used for only two ranked modifications. This order rule was followed as it is the same rule followed by the automated classifier. The automated classifier tends to be more pessimistic through counting modifications by ensuring that modifications that may be a combinations of fault repairing and feature introduction are counted as fault repairing modifications to get a more complete count of repaired faults in the software system.

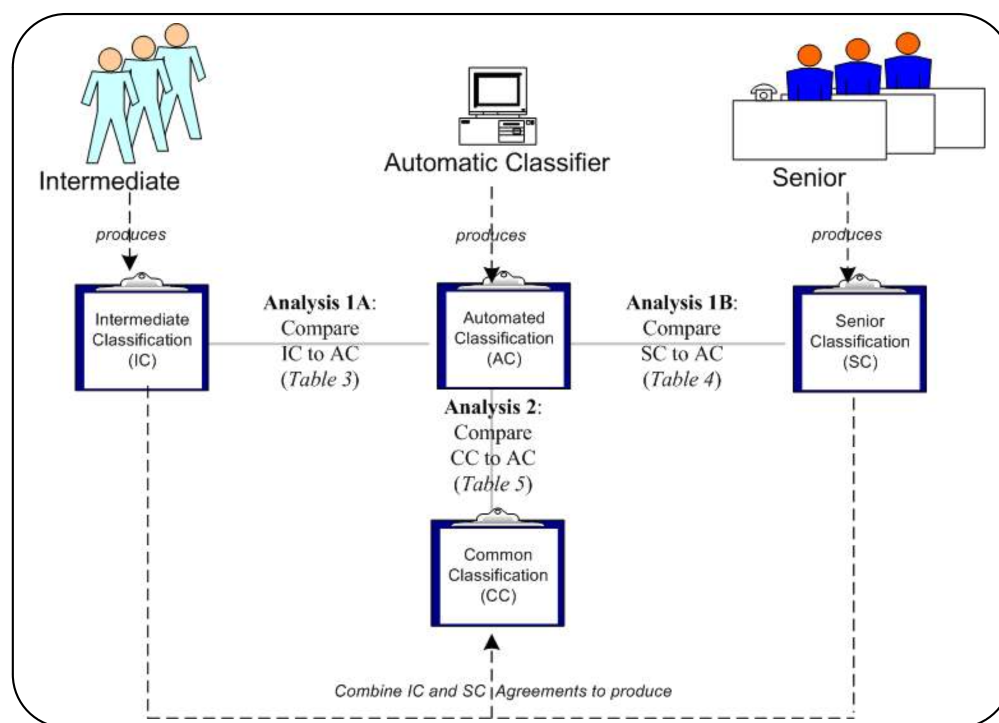

Figure 4.1: Analysis of the Classification of Changes by the Senior and Intermediate Developer Groups

The two groups of developers were given the same 54 change mes-

sages to classify. Every developer in a group was given a disjoint set of 18 messages to classify. We then combined the classification by every developer to arrive to a classification for the whole group (Intermediate and Senior classifications). The same 54 change messages were classified using our classifier program. Figure 4.1 summarizes the types of analysis we performed on the data. We performed two types of analysis:

- In the first analysis we compared the intermediate group classification to the automatic classifier (Analysis 1A) and the senior group classification to the automated classifier (Analysis 1B).
- In the second analysis (Analysis 2), we combined the classification done by the senior and intermediate groups to create a common classification. We then compared this common classification to the classification done by the automated classifier.

We now present the results of the two types of analysis.

4.4.1 Analysis 1A and 1B of Developers' Classifications

	Automated Classifier			
Manual Classifier	GM	FR	FI	Total
GM	15	2	3	20
FR	4	14	7	25
FI	0	0	9	9
Total	19	16	19	54

Table 4.3: Classification Results for the Intermediate Developers Group vs. the Classifier Program(Analysis 1A)

Table 4.3 and 4.4 summarize the results for analysis 1A and 1B. The last row in both tables shows the distribution of modification types as classified by the automated classifier. The automated classifier categorized the 54 modifications into 19 GM, 16 FR, and 19 FI modifications.

The last column of both tables shows the results of the manual classification which differs between the two groups of software developers. Table 4.4 shows that our automated classifier has classified 16 changes as FR changes. By comparison column 2 of Table 4.4 shows that the senior developers have classified 15 out of these 16 modifications as FI and one of the modifications as GM.

	Automated Classifier			
Manual Classifier	GM	FR	FI	Total
GM	15	1	4	20
FR	3	15	4	22
FI	1	0	11	12
Total	19	16	19	54

Table 4.4: Classification Results for the Senior Developers Group vs. the Automated Classifier

The diagonal of both tables lists the number of times the developers and the automated classifier agreed on their classifications. Summing the diagonal values in both tables shows that:

- For Table 4.3 the intermediate developers agreed 38 (15 + 14 + 9) times with the automated classifier. The intermediate group agreed ($\frac{38}{54} = 70\%$ of the time with the automated classifier.
- For Table 4.4, the senior developers agreed 41 (15 + 15 + 11) times with the automated classifier. The senior group agreed ($\frac{41}{54}$) $= 76\%$ of the time with the automated classifier.

We calculated Cohen's Kappa (κ) coefficient for both groups of developers [Coh60]. The Kappa coefficient is a widely adopted technique to measure the degree of agreement between two raters, in our case: the automated classification technique and the developers participating in our experiment. The Kappa for the senior group and the automated classifier is 0.64. The Kappa for the intermediate group and the automated classifier is 0.56. According to the Kappa thresholds values proposed by El

Emam [Ema99] (see Table 4.5), the agreement between the automated classifier and the group of senior developers is *substantial*. The agreement between the automated classification and the group of intermediate developers is high *moderate*. These results are similar to the ones determined by Mockus and Votta who found *moderate* agreement between an automated classification and a manual classification using the El Emam classification. In brief, the results indicate that automated classification techniques are likely to achieve similar classifications to ones done manually by professional software developers.

Kappa Value	Strength of Agreement
< 0.45	Poor
$0.45 - 0.62$	Moderate
$0.63 - 0.78$	Substantial
> 0.78	Excellent

Table 4.5: Kappa Values and Strength of Agreement

4.4.2 Analysis 2 of Developers' Classifications

	Intermediate Classifier			
Senior Classifier	GM	FR	FI	Total
GM	17	2	1	20
FR	2	19	1	22
FI	1	4	7	12
Total	20	25	9	54

Table 4.6: Classification Results for the Senior Developers Group vs. the Intermediate Developers Group

For the second analysis, we combined the classifications done by both the senior and intermediate developer groups to create a common classification. We removed from the common classification change messages which both intermediate and senior developers disagreed in classifying. We felt that since both human classifiers could not agree on the classification of a message, then we should not expect an automated classifier to de-

termine the correct classification of that message. Table 4.6 summarizes the classification results for the senior and intermediate developers. Out of 54 change messages, the senior and intermediate developers disagreed on the classification of 11 change messages. The Table indicates an 80% overall agreement between both developer groups and a Kappa of 0.68, corresponding to a *substantial* agreement. A closer look at the degree of agreement between classifiers for each change type reveals that there is an 85% agreement for GM changes, 81% agreement for FR changes, and 68% agreement for FI changes. In short, developers are likely to agree more on classifying GM or FR changes, than on classifying FI changes. This is likely due to developers using specific keywords to classify GM and FR messages such as "*bug*" or "*fix*".

We used the agreed on classifications to create a common classification for the remaining 43 change messages. We compared the common and the automatic classification (see Table 4.7). The Kappa for the common classification is 0.71. Using the Kappa thresholds values shown in Table 4.5, we note that the agreement between the automated classification and the common classification is *substantial*. The table indicates that the automated classification and the common classification agree 81% of the time.

	Automated Classifier			
Manual Classifier	GM	FR	FI	Total
GM	14	1	2	17
FR	2	14	3	16
FI	0	0	7	7
Total	14	15	12	43

Table 4.7: Classification Results for the Common Classifications between Both Developers Group vs. the Automated Classifier

In addition to performing the Kappa analysis on the classifications, we used the Stuart-Maxwell Test. Whereas Kappa examines the agreement between classifiers, the Stuart-Maxwell Test examines the disagreement between classifiers. In particular, the Stuart-Maxwell Test tests

the marginal homogeneity for all classification categories [BS77, AE70, AA55, MH]. One reason classifiers disagree is because of different tendencies to use classification categories. The Stuart-Maxwell Test determines if classifiers have biases towards specific classification categories or if they do not. A small probability value P implies that there is an association between both classifiers and that no bias exists. Table 4.8 summarizes the results for the Stuart-Maxwell Test for the classification tables. The Stuart-Maxwell Test holds for all classification tables at above 90%. These Stuart-Maxwell Test results agree with the Kappa analysis performed above.

	Maxwell Test	
Classification Table	Chi-Squared	P
Intermediate vs. Automated	10.494	0.0053
Senior vs. Automated	6.786	0.0336
Common vs. Automated	5.238	0.0729

Table 4.8: Results of the Stuart-Maxwell Test

The results of analysis 1A, 1B, and 2 indicate that an automated classification technique for modification records for open source systems, using the change message attached to the records, is likely to produce results that are substantially similar to classifications done manually by professional developers. These results are encouraging as they permit us to recover automatically from open source system a historical overview of the bug fixes applied to the system. These bug fix modifications could be used, for example, to study the quality of open source systems and to analyze the benefit of adopting different techniques to improve the quality of software systems in general [HH03b, HH] (*see* Part III).

4.5 Results For Part 3: Comparing Open Source and Commercial Change Messages

In the last part of the survey, we asked the participants to compare the change messages they had classified in part 2 to the change messages they usually find at work. The participants indicated that the change messages they ranked are as descriptive as (5 out of 10) change messages they find in industrial software systems. Nevertheless developers commented on differences between the classified changes and the ones they find at work. A number of participants indicated that as the size of the development group increases, they are more likely to write longer and more descriptive change messages to avoid other developers asking them for clarifications for their changes. They indicated that open source change messages are as descriptive as the ones they usually encounter in large projects. They also indicated that whereas open source change messages list details concerning a change (such as a bug description), there is a tendency for change messages in commercial systems to have less text and to reference a bug or a feature request number. The details for such a bug or feature request are usually stored in a separate database (by means of a bug database or feature tracking system). Some of the intermediate developers preferred the fact that such information was stored in a separate system and not repeated; whereas all the senior developers preferred having the details of the bug/feature request attached to the change message instead of having to access another system to retrieve such information. These results suggest that bug or feature tracking systems need to be tightly integrated with source control systems to offer developers easy access to all information related to a change while avoiding duplicating this information in multiple systems.

4.6 Conclusion

In this chapter, we investigated an artifact of software development that is rarely studied; namely, the change messages attached to every modification committed to a source control system. We used a survey technique in which intermediate and senior developers were asked a number of questions about these messages. Although we surveyed a small number of developers, we believe that their replies are representative of industrial software developers, since they worked at different companies spanning various domains and they have several years of industrial experience. Nevertheless, it is desirable to investigate that our survey findings hold using a larger number of participants.

We focused on understanding the type of information that exists in change messages. We found that developers are more likely to record the rationale for a change than to list alternative implementations or limitations. We discovered that developers make use of information in change messages to help understand legacy code and to fix bugs. Our survey results indicate that senior developers use source control systems to keep abreast of the progress of a software project and use the change messages as a quality monitoring facility to detect potential bug prone changes or to ensure that their code is not touched by others who do not own it (code ownership). Moreover, our results indicate that change messages in open source projects are as descriptive as change messages in industrial systems. We as well investigated the possibility of classifying changes automatically into bug fixing, bookkeeping and feature introduction changes. Our results indicate that automated classifications agree over 70% of the time with classifications done manually by software developers.

The findings of our survey suggest that change messages are a valuable resource that is used by practitioners to maintain and manage software projects. We conclude that researchers should investigate techniques and approaches to improve the quality of the change messages and to make them more accessible for developers as they evolve software systems. In the following parts of this thesis, we present techniques and

approaches which attempt to formalize some of the intuition of practitioners about monitoring changes to source code, and the ad-hoc uses of software repositories by practitioners.

Appendix

In this appendix we show the three parts of the survey given to the participants in our study.

Survey About: Messages Entered into Source Control Systems

Thank you for participating in this survey. The purpose of this survey is to gain a better understanding of how professional software developers use source control systems such as Perforce and CVS. In particular, we are interested in the message (the Source Control Change Message), entered when a developer commits a change to the source control. The survey should take you under 20-25 minutes in total. In any question, feel free to add remarks of interest.

Part 1. General

Your Background

i. How many years of development experience do you have? ____

ii. How many years have you used a source control system for? ____

iii. What is the average team size for the projects where you used a source control system? ____

iv. Have you ever lead a team of developers? ____

Source Control Change Messages

In the following questions please give a number from 0 to 10, according to this scale: (0 – Rarely, 5 – Sometimes, 10 – Most of the time).

Q1.When you commit a change to a source control system, how often do you enter a change message? (0 to 10) ____

Q2. How often do you monitor other developer's submissions to source control? (0 to 10) ____

Q3. How often do you monitor other's submissions to source control (0 to 10)

a. To keep abreast of the project's progress (recent features, recent bug fixes) ____

b. To check the correctness of their submission ____

c. To know the effects of their submission on the code and APIs ____

d. Other ____
Specify ____________________________

Q4. When you read the change message entered by other developers, how often do you find them meaningful and sufficient to get an idea of what changed? (0 to 10) ____

Q5. After reading a change message, how often do you examine the changed code? (0 to 10) ____

Q6. If a change message is not meaningful or empty, how often do you examine the source code? (0 to 10) ____

Q7. How often do you examine the changed code when (0 to 10):

a. The change is done by a junior developer ____

b. The change is done by a particular developer ____

c. The change is done to a file/subsystem you worked recently on ____

d. The change is done to a file/subsystem that your code depends ____

e. The change is done to a file/subsystem that depends on your code ____

f. The change is done in off working hours (weekend/night) ____

g. The change is done close to a release date ____

h. The change message indicates it is a bug fix ____

i. The change affects a large number of files (how many on average?): ____

j. The change prevents your code from compiling ____

k. Other ____
Specify ___________________________

Q8.When reading a change message, how often do you find (0 to 10):

a. The reason/rationale for the change: ____

b. A detailed description of the change and its effect on other parts of the system: ____

c. A description of alternative designs/implementations: ____

d. Warnings about limitations: ____

e. Indication of possible future enhancements of the changed code (todo list): ____

f. Un-meaningful or empty descriptions: ____

g. Other ____
Specify ___________________________

Q9. How often do you use/read previous change messages when you are

a. Doing design: ____

b. Writing new code: ____

c. Testing: ____

d. Understanding old code: ____

e. Fixing bugs: ____

f. Adding new features to old software: ____

g. Improving performance: ____

h. Reviewing and inspecting code: ____

i. Enhancing an old feature: ____

j. Monitoring the progress of the project: ____

k. Other ____
Specify ________________________

Part 2. Classifying Changes

The following are 18 actual change messages, which have been randomly picked from several large software projects. For every change, please classify the change message as

BF: A bug fix.

FE: A feature enhancement/addition

BK: A bookkeeping change such as merging of source control branches, updating copyright dates, indentations, spelling corrections, etc.

NS: Not sure. The change message does not give enough details to classify the change.

Please allocate 10 points among the 4 classes (BE, FE, BK and NS). For example, if you feel confident that a change is a bug fix then assign all 10 points to BF. If you feel a change is likely a bug fix and a feature enhancement then you could assign 5 points for BF and 5 points for FE. If you are not sure how to classify the message then assign all 10 points to NS. For example:

0. “fix error in hyperzont.c”

   BF. `_10_` FE._____ BK._____ NS._____

Here are the change messages that you are to classify.

[*Personalized Generated List of Change Messages for Every Participant*]

Using the Data from this survey

Can we acknowledge you when we report these results? (yes/no, I would like to remain anonymous) ____

Part 3. Comparing Change Messages

1. From 0 (less descriptive) to 10 (more descriptive): Would you consider the change messages you saw in that survey to be more or less descriptive than the ones you see in your daily work? ____

2. What are the main differences you see between the change messages in this survey and the ones you see at work? Any things that strike you as different? ____

Part II

Using Software Repositories to Assist Developers

Developers working on large software systems are usually faced with many challenges as they work on evolving the source code to meet the changing needs of the customers. Developers require tools and approaches to understand the current structure of a software system and to accurately propagate changes throughout the software system.

This part deals with both of these issues by presenting two pieces of research work:

- **Source Sticky Notes**: We present an approach which recovers valuable information from source control systems and attaches this information to the static dependency graph of a software system. We call this recovered information – *Source Sticky Notes*. These notes along with the software reflexion framework [MNS95] could assist developers in understanding the architecture of large software systems. [Chapter 5]
- **Development Replay Approach**: We present the *Development Replay* (DR) approach which reenacts the changes stored in the source control repositories using proposed tools or strategy. The proposed tool benefit is measured using changes done to the source code by professional software developers over an extended period of time. This approach permits us to empirically asses the effectiveness of not-yet-adopted or not-yet-existing code maintenance tools and strategies. [Chapter 6]

This part is likely to be of interest to developers working on large software systems. This part shows that historical changes stored in source control repositories could assist in understanding large software systems, and could be used to get rough estimates on the benefits of adopting tools and approaches.

CHAPTER 5

Using Development History Sticky Notes to Understand Software Architecture

Maintenance of evolving software systems has become the most frequently performed activity by software developers. A good understanding of the software system is needed to reduce the cost and length of this activity. Various approaches and tools have been proposed to assist in this process such as code browsers, slicing techniques, etc. These techniques neglect to use a central and vital piece of data available – the historical modification records stored in source control systems. These records offer a rich and detailed account of the evolution of the software system to its current state.

We present an approach which recovers valuable information from source control systems and attaches this information to the static dependency graph of a software system. We call this recovered information – Source Sticky Notes. We show how to use these notes along with the software reflexion framework to assist in understanding the architecture of large software systems. To demonstrate the viability of our

approach, we apply it to understand the architecture of NetBSD – a large open source operating system.

5.1 Introduction

THE primary business of software is no longer new development; instead it is maintenance [LS81, Gla92] and a good understanding of the software system is needed to reduce the cost of maintaining it. Software understanding tasks represent fifty to ninety percent of the maintenance efforts [Sta84].

Good documentation can significantly assist in software understanding tasks. Unfortunately software developers commonly do not document their work. Documentation rarely exists and if it does it is usually incomplete, inaccurate, and out of date.

Faced with the lack of sufficient documentation, developers choose alternative understanding strategies such as searching or browsing the source code. The source code in many cases represents the only source of accurate information about the implemented system [Sim98]. Developers search the code using tools such as `grep`. They browse the code using text editors or cross-reference code browsers such as LXR, which permit them to navigate the static dependencies of the software system. For example, developer can track variable/function usage and locate their declarations. The usefulness of this code browsing technique is limited by the size of the software system and the amount of information a person can keep track of while jumping around the source code [vMV95, SCH98].

To overcome the lack of documentation and the pressing need to understand large systems as developers evolve them, we propose to speedup the understanding process by using knowledge acquired from mining the historical modification records stored in source control systems. Source control systems track the evolution of source code. Throughout the lifetime of a projects, source code is changed to add new features, enhance

old ones, or fix bugs. All these code changes are stored in the source control system. Along with the code changes other valuable information are kept by the source control system. For example, the source control system stores a message for each change. This message is entered by the developer performing the change. This message offers us an opportunity to gain some insight about the change rationale. For example, a developer may indicate that a change was done to fix a recently discovered bug in the field or to add a new feature.

This rationale message along with other change details stored by the source control system provide a valuable source of information about the software system and the complex interaction between its components, the same way that history can guide us to understand the current state of the world, as noted eloquently by David C. McCullough, a president of the Society of American Historians:

> *"History is a guide to navigation in perilous times. History is who we are and why we are the way we are."*

In this chapter we propose to attach these valuable change details (such as the rationale message) to the dependencies between the entities of a software system. Specifically for each change we determine its affect on the software's dependency graph, such as the addition of a call to a function. Then we attach these change details to the corresponding edges in the graph. We call this recovered change details – *Source Sticky Notes*, as they are attached to the dependency edges to remind developers of valuable information which may assist them in understanding the system at hand.

5.1.1 Organization of Chapter

The chapter is organized as follows. Section 5.2 presents a process for understanding the architecture of a software system and breaks the process into three major steps. These steps are repeated by developers until they

have a sufficient (good enough) understanding of the part of the system they are interested in. Then in Section 5.3, we overview the software reflexion framework which has been proposed by Murphy *et al.* to assist in understanding the structure of software systems. In Section 5.4 we outline the key questions that developers pose during their investigation of the results of the reflexion framework. Furthermore, we demonstrate the benefit of using the source control data to address these questions. We introduce the idea of *Source Sticky Notes* – which augment static dependencies between source code entities and permit us to attach information derived from the source control data. In Section 5.5, we describe the data stored in source control repositories and present the techniques we use to recover such data to build Source Sticky Notes. Then we demonstrate the viability of our proposed approach through a case study on the NetBSD operating system in Section 5.6. In Section 5.7, we describe related works and compare them to our approach. In Section 5.8, we summarize our findings and draw conclusions.

5.2 The Architecture Understanding Process

The architecture of a software system describes the structure of the system at a high level of abstraction. Individual functions and even modules are not described in detail; instead, they are abstracted into higher level constructs such as subsystems. Subsystems and interactions between them are shown in an architecture document. A well documented architecture provides a good understanding of the entire software system and eases the understanding of the design decisions involving interactions among its subsystems. Unfortunately, software architectures are rarely documented. Therefore developers attempt to understand the architecture using the source code as the definitive guide.

The architecture understanding process followed by developers can be broken into three major steps: Propose, Compare, and Investigate (see Figure 5.1). These steps are repeated in an iterative manner by develop-

ers At first the developer *proposes* a conceptual breakdown of the software system – conceptual architecture. The conceptual breakdown defines the major components of the system and the interactions between them. This proposed conceptual breakdown is based on the developer's assumptions and intuition. In the following step, the developer *compares* her/his proposed conceptual breakdown with the actual source code.The developer *investigates* the results of the comparison. New knowledge is acquired from the source code and the developer updates her/his understanding of the software system. The developer would then propose an updated conceptual breakdown based on the newly acquired knowledge. This process is repeated till the developer has acquired sufficient understanding of the architecture of the software system. The developer now moves on to tackling other challenges such as adding functionality or fixing bugs. This process is a simplification and abstraction of software understanding processes that were derived from our experience studying and working with large software systems [BHB99a, HH00] and research by others based on observing the process performed by developers in industry to understand complex software systems [vMV94].

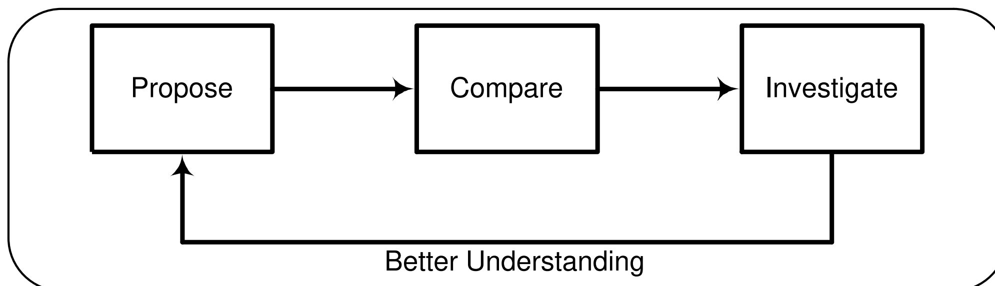

Figure 5.1: Overview of the Architecture Understanding Process

We now discuss each of the steps in the architecture understanding process in detail.

5.2.1 Propose

In the propose step, the developer approaches a software system with a set of assumptions and preconceived ideas about its architecture and the interaction between its various subsystems. These assumptions are usually based on any available documentation for that system and the developers' previous interactions with that system or other similar systems. Unfortunately, the documentation for software systems rarely exists and if it does it is rarely up-to-date. Instead a developer relies on her/his current knowledge about the internals of the system, the knowledge she/he acquired from interviewing other developers (in particular senior ones) on the team, and her/his knowledge of the architecture of similar systems (*i.e.* the reference architecture) to form his assumptions. Influenced by these assumptions, the developer proposes an initial conceptual breakdown of the software system.

For example, a developer working on enhancing features in an operating system, might begin by proposing a conceptual breakdown of the operation system which consists of five conceptual subsystems: *File System*, *Memory Manager*, *Network Interface*, *Process Scheduler*, and an *Inter-Process Communication*. The developer might also assume that these subsystems interact in a particular fashion to implement specific features. For example, the *File System* would depend on the *Network Interface* to support networked file systems such as NFS. Or the *Memory Manager* would depends on the *File System* to support swapping of processes to disk when the system runs out of physical memory. These assumptions form the conceptual view of the software system and are influenced by the reference architecture of an operating system, descriptions of operating systems in text books, and available documentation about the system [BHB99a].

5.2.2 Compare

The proposed conceptual breakdown of the software system is influenced by many assumptions. These assumptions must be verified. In the Compare step, these assumptions are compared against the actual implementation to either refute or support them. Several approaches and tools have been proposed to assist developers in the compare step. The software reflexion framework is an example of such approaches [MNS95].

Once the developer has compared her/his conceptual breakdown with the actual implementation, she/he gains a more accurate view of the structure of the software system. Unfortunately, she/he are left with many unanswered questions about the interactions between the software's subsystem. The developer may find unexpected dependencies that indicate, for example, that the *Network Interface* uses the *Memory Manager*. The developer may find unexpected dependencies or may realize that expected dependencies are missing. These dependencies form the gaps between the conceptual understanding and the actual implementation. The developer needs to investigate the reasons for such gaps.

5.2.3 Investigate

The Investigate step of the understanding process is the most time and resource intensive step. The developer needs to determine the rationale behind the dependencies that caused the gaps. For example, given an unexpected dependency, the developer may need to determine if there are any good reasons for such a dependency to exists, or if the dependency is due to the misunderstanding of the developer who introduced it.

Research in recovering the software architecture has focused primarily on assisting developers in creating conceptual views of software systems and comparing them to the source code. Yet the process of investigating the results of the comparison has been neglected and it depends on ad-hoc methods such as reading source code, browsing documentation and newsgroup postings; and asking senior developers for clarifications about

the current state of the system. For example puzzled by the unexpected dependency between the *Network Interface* and the *Memory Manager*, a developer may contact a senior developer to uncover the rationale behind such dependency.

Unfortunately uncovering this rationale may be difficult, as the senior developer may be too busy or may not recall the rationale for such dependency, the developer who introduced the dependency may no longer work on the software system, or the software may have been bought from another company or its maintenance out-sourced. Therefore the developer may need to spend hours/days trying to uncover the rationale behind such unexpected dependency. In some cases the rationale for an unexpected dependency may be justified due to, for example, optimizations or code reuse; or not justified due to developer ignorance or pressure to market.

The goal of our work is to support developers in the time consuming Investigate step. In the following section, we present the software reflexion framework which can be used to guide developers as they to understand the structure of large complex software systems. We then show how to integrate our approach (*Source Sticky Notes*) with the software reflexion framework to reduce the time needed by developers to understand a software system.

5.3 The Software Reflexion Framework

The software reflexion framework assists developers in understanding the structure of their software system. In particular, it provides support for the Propose and Compare steps of the architecture understanding process described in the previous section. Figure 5.2 illustrates the architecture understanding process based on the software reflexion framework:

1. Developers use their acquired knowledge about the software system to:

a) *propose* several conceptual subsystems and dependencies between these subsystems. (*conceptual subsystems and dependencies between subsystems*)

b) *propose* a mapping from the implementation the system (*i.e.* the source code in files/directories) to these conceptual subsystems. (*mapping source entities to subsystems*)

2. Developers *compare* their proposed conceptual system breakdown and the extracted concrete dependencies from the source code. Gaps such as missing expected dependencies or unexpected dependencies are noted.

3. Developers *investigate* the discovered gaps.

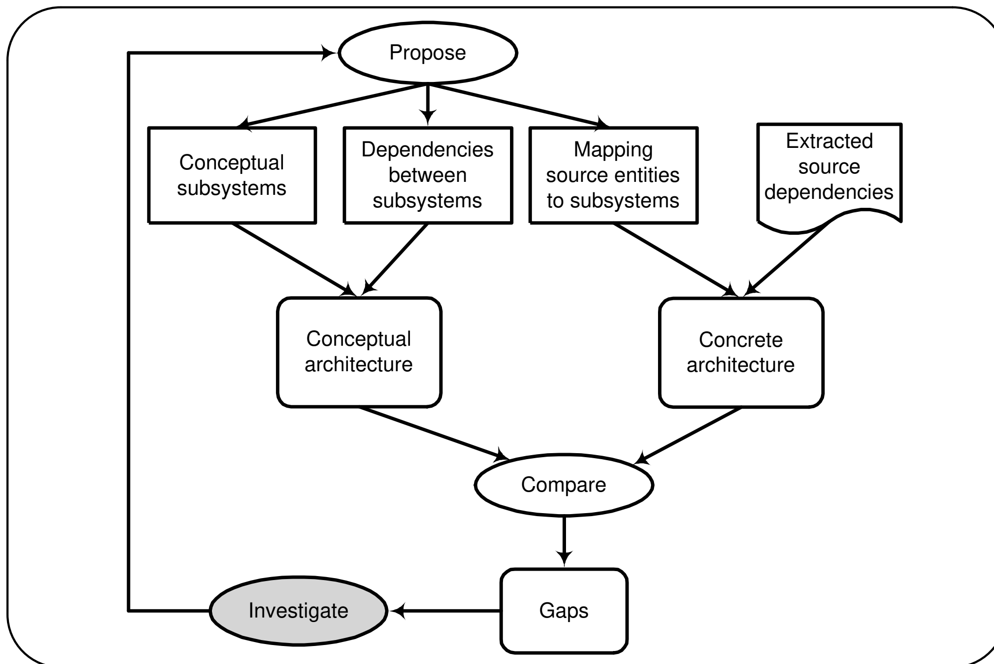

Figure 5.2: Architecture Understanding Process Using The Software Reflexion Framework

Once the gaps are investigated, the developers have a better understanding of the software system. They may choose to update their pro-

posed conceptual breakdown.

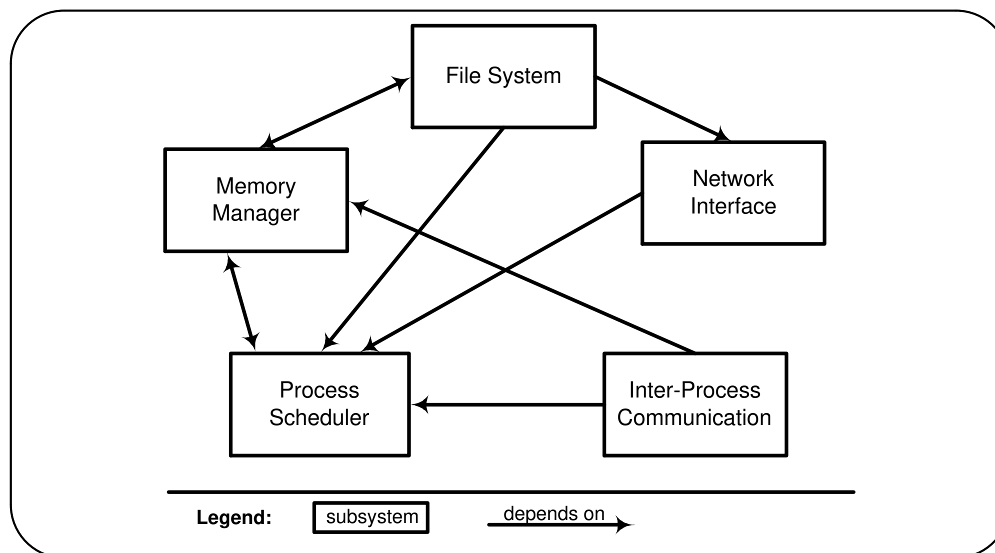

Figure 5.3: Conceptual View of an Operating System [BHB99a]

5.3.1 A Clarifying Example

In this subsection, we give an example of using the software reflexion framework to understand the architecture of an operation system. For the first step in the reflexion framework, the developer proposes conceptual subsystems and dependencies between these subsystems. This proposal constitutes the conceptual architecture of the software system. Figure 5.3 shows a proposed conceptual architecture of an operating system, which a developer may derive based on her/his knowledge of the reference architecture of traditional operating systems and other documentation [BHB99a]. Next, the source code files are mapped to the conceptual subsystems. For example, all files in the “src\sched” directory may be mapped to the *Process Scheduler* subsystem, similarly all files in the “src\ipc” directory may be mapped to the *Inter-Process Communication* subsystem.

In the second step, dependencies between these conceptual subsystems are derived using a source extractor which parses the source code

to recover concrete dependencies. For example if a file in “src\ipc“ calls a function defined in another file in “src\sched“ then this is considered to be a dependency relation between the *Inter-Process Communication* and *Process Scheduler* subsystems. These extracted dependencies along with the proposed mapping between files and conceptual subsystems form the concrete architecture of the software system. Now the concrete architecture is compared against the proposed conceptual architecture. Figure 5.4 shows a reflexion diagram which highlights the differences (gaps) between the proposed and the actual extracted dependencies among the subsystems. In this case all expected dependencies existed in the software system. There are two unexpected dependencies; these are the dashed lines in Figure 5.4.

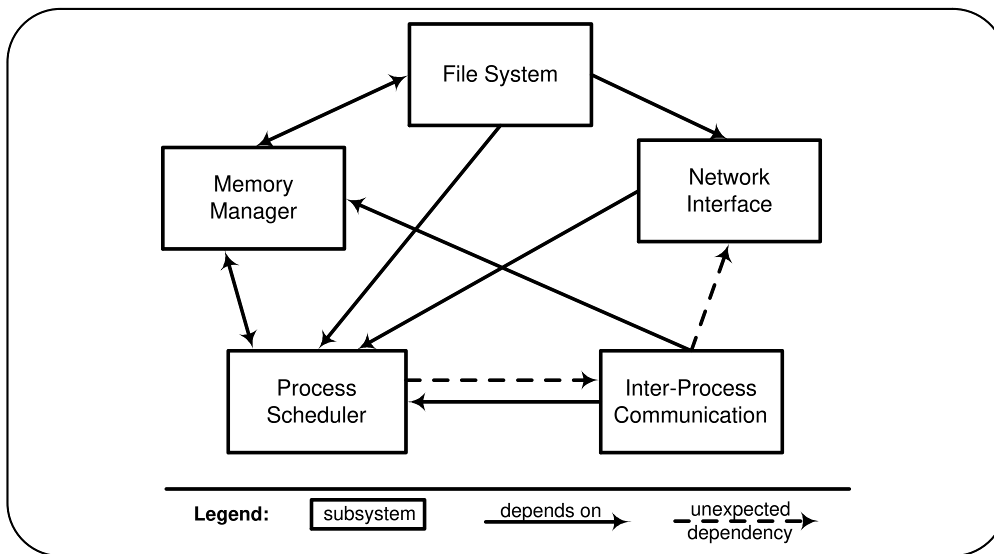

Figure 5.4: Reflexion Diagram for an Operating System

In the third step, the developer investigates the discovered *gaps* between her/his conceptual view and the concrete (*as implemented*) view of the system. In particular for the example shown in Figure 5.4, she/he needs to uncover the reasons for:

- The *Process Scheduler* to depend on the *Inter-Process Communica-*

tion and for

- the *Inter-Process Communication* to depend on the *Network Interface*.

Investigating such gaps is a challenging and time consuming task with no support provided by the reflexion framework. Ad-hoc methods such as interviewing senior developers, reading through project documentation or archived project communications are used to assist in the investigation. In the following section, we focus on the Investigate step (the grey oval in Figure 5.2). We categorize the types of dependencies highlighted by the reflexion diagram. Then we outline several types of questions posed by developers as they investigate the gaps. By carefully studying what is being investigated – the gaps – and how it is being investigated – the questions – we hope to understand better the needs of developers throughout this step. This should assist us in developing techniques to assist them.

5.4 Investigating Dependencies - The W4 Approach

As pointed out in the previous sections, the Investigate step is the most time consuming step in the architecture understanding process, with little support by software engineering research. In this section, we introduce the concept of *Source Sticky Notes*. These notes are derived from the source control system and can be used to assist developers in this step. Using these notes developers can gain insight about the rationale for gaps between their conceptual understanding of the software system and the actual implementation. But before we introduce these notes, we present two important aspects of the investigate step: the type of dependencies and the questions posed during investigations. These aspects will help us define the contents of the Source Sticky Notes proposed at the end of this section.

5.4.1 Three Types of Dependencies

The software reflexion framework focuses on identifying gaps between the conceptual understanding of the software system and its actual implementation. As developers investigate these gaps, they can classify the dependencies that appear in the reflexion diagram into the three types illustrated in Figure 5.5:

- **Convergences:** These are dependencies that exist in the software system as expected by the developer. It is possible that the reason for the existence of such dependencies does not match the rationale the developer had in mind. Yet, they are rarely investigated. Instead most of the focus of the investigation step is on the *Absences* and *Divergences*. These two latter types represent the gaps between the conceptual understanding and the actual implementation.

- **Absences:** These are *missing* dependencies that the developer expected to find in the software system but the concrete architecture revealed that they do not exist. Absences could be due to lack of knowledge of the developer investigating the system, changes in the architecture of the system, or removal of features. For example an operating system may no longer provide network support, therefore the Network Interface subsystem may not exist. Based on our experience of studying several large software systems, absences occur rarely.

- **Divergences:** These are *unexpected* dependencies that exist in the implemented software system. Divergences may be due to undocumented features, pressure to market, developer laziness, etc. For example, the operating system may have undocumented features such as supporting special hardware devices, or the source code may have been optimized by means of unusual or messy dependencies. Or during a tight release cycle a developer may have decided to bypass clean design principles to fix a bug or add a feature in a short time. Based on our experience, there are many divergences in software

systems. In some extreme cases, we found systems in which almost every subsystem depends on every other subsystem. This poses a great challenge for developers as they would have to investigate a large number of divergences. Any tool support to assist them in the investigation would be appreciated and valuable.

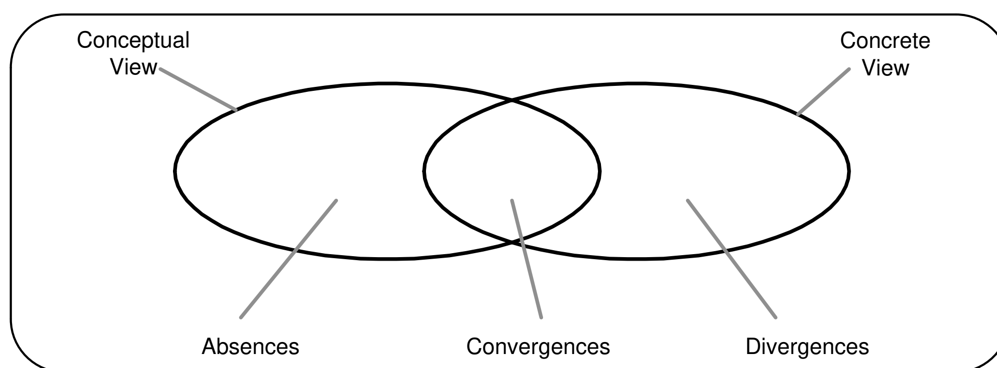

Figure 5.5: Classification of Dependencies

5.4.2 Questions Posed During Investigation

As developers investigate these dependencies, they pose various questions. The goal of these questions is to uncover the rationale for the missing and unexpected dependencies which in turn represent the gaps in understanding. We can classify these types of questions into four types. We call them the W4 questions – *Which? Who? When? Why?* We discuss these questions in detail.

- *Which?* Which concrete source code entities are responsible for these unexpected dependency in the concrete view? Based on the names of the entities involved in the dependency or their source code, the developer may be able to deduce the reason for the existence of such dependency. Unfortunately, this is not usually the case. Thus developers find themselves asking several other questions.

- *Who?* Who introduced an unexpected dependency or removed a missing dependency? A knowledge of this person gives developers hints and assists them in understanding the reasons for such gaps. A gap due to a change made by a novice developer may suggest that the developer is at fault and the change must be fixed. On the other hand, the change may have been performed by a senior developer with a well established record for producing high quality code. In that case, the investigating developer should have a good reason to believe that the senior developer introduced it for good reasons. Therefore, the investigating developer may consider adjusting her/his conceptual view of the system.

- *When?* When was the unexpected dependency added or the missing dependency removed? Even though a dependency being investigated had been introduced by a senior developer, one may want to ensure that this dependency was not introduced just to fix a critical bug under a tight release schedule and should be reworked. In that case, one may need to determine if the dependency was modified in the few days/hours before a release, hence suggesting it may be a hack just to get the product out of the door or if it is a justified dependency that the investigating developer should expect.

- *Why?* Why was this unexpected dependency added or why was an expected dependency missing? A knowledge of the rationale for the investigated dependency may be key in explaining the gap and would improve the developer's conceptual understanding of the system.

5.4.3 Source Sticky Notes

In the previous two subsections, we gave an overview of the types of dependency gaps highlighted by the reflexion diagram and the types of questions posed by developers investigating these gaps. We noted that in large software systems, divergences are the most common type of gap

highlighted by the reflexion diagram. We also noted that developers seek answers to several questions regarding these gaps. Since the reflexion diagram is based on static dependencies, it provides little support for developers who are searching for clues to uncover the rationale for the highlighted gaps.

Static dependencies are only capable of giving us a current static view of the software system without details about the rationale, the history, or the people behind the dependency relations. Such details are vital in assisting developers through the understanding process.

To overcome the shortcomings of static dependencies, we propose to augment dependencies by attaching *Source Sticky Notes* to them. These notes specify various attributes for each dependency – such as the name of the developer, the rationale behind the addition or removal of a dependency, and the date the dependency was modified. These notes would make the job of the developer easier as they could help answer the W4 questions (*Which? Who? When? Why?*) posed by developers while investigating dependencies. In the fast paced world of software development with tight schedules and short time to market, manually recording such attributes for each dependency is neither possible nor practical, for the following reasons:

1. For established software projects, it would be a time consuming and error prone task for developers to go through each dependency in the software system and attach notes to it. In many cases the developer may no longer recall the reasons for the dependencies and in most cases won't recall the details for the other attributes such as the date it was modified.

2. For new projects, we would have to ensure that developers annotate each created dependency. Again this is extra work which most developers would not be interested in doing.

We conclude that attaching Source Sticky Notes to static dependencies would assist developers in improving their understanding of software sys-

tems, yet developing such sticky notes manually is a rather cumbersome and impractical option. To overcome this quandary, we propose using the historical modification information stored by source control systems. In the following section we give an overview of source control systems and present an approach to recover information from source control system to create Source Sticky Notes and to attach them to static dependencies.

5.5 Source Control Systems

As a software system evolves to implement the various functionality required to fulfill customers requirements and stay competitive in the market, changes to its source code occurs. These changes are done incrementally over the lifetime of a project by its various developers. Source control systems as CVS or Perforce record the history of changes to the source code of the software system.

The source code of the system is stored in a source repository. For each file in the software, the repository records details such as the creation date of the file, modifications to the file over time along with the size and a description of the lines affected by the modification. Furthermore, the repository associates for each modification the exact date of its occurrence, a comment typed by the developer to indicate the rationale for the change, and in some cases a list of other files that were part of the change described by the developer's comment.

This detailed description of the history of code modification permits us to automatically build Source Sticky Notes for each dependency. Luckily, such data is already being entered by developers as part of their routine development process, thus generating these notes doesn't require any more time commitment by the developers.

Source control systems store the details of the modification at the line level of a file, which is not at the right level of detail for generating Source Sticky Notes. Therefore, we need to map source code changes to appropriate source code entities (*i.e.* functions, macros or data types). Once

mapped we can determine if a change caused the addition or removal of a dependency. We can then associate modification attributes (developer, rationale, and date) to the modified dependencies between these mapped source code entities.

5.5.1 Attaching Sticky Notes to Static Dependencies

To automate the attachment of sticky notes to static dependencies, we use a two pass approach to analyze the source control repository data:

1. In the first pass, each revision of a file is parsed and all defined entities ($i.e.$ functions, macros or data types) are identified. In particular, we record their name, and their content. For example, file A may have two revisions: an initial revision containing four functions, and a second revision in which one of these functions is removed and another one added. By parsing each revision and identifying all the entities that were defined for all files throughout the development history of a project, we can generate the equivalent of a symbol table for a software system. In contrast to a traditional symbol table, this *historical symbol table* has all symbols (entities) that were ever defined in the project's lifetime.

2. Using this historical symbol table, we re-analyze each revision of each file. We locate for each entity in a revision which other entities it depends on in the historical symbol table. This produces a snapshot of the dependencies between all the entities of a software system at the exact moment in time of each revision of a file. Since the source control system stores a modification record for each revision of a file, we are able to attach a Source Sticky Note to new or removed dependencies for a revision. The Source Sticky Note contains the data recorded by the source control system for the corresponding modification record. Each Source Sticky Note has four subsections which can be used to answer the four types of questions posed in the W4 approach for investigating gaps: *Which? Who? When? Why?*

> As a results of parsing each revision for each file, we have a *historical dependency graph*. This historical dependency graph is composed by successively combining snapshots of dependency graphs for all revisions of all files throughout the lifetime of a software project. A detailed description of the approach used to recover the historical dependency graph is available in Chapter 3.

The historical dependency graph is then used to assist developers to investigate dependency gaps. Each dependency in the software system has attached to it Source Sticky Notes for each change that has affected that dependency. Thus a developer can read all the Source Sticky Notes attached to any dependency.

We observe that the order of the Source Sticky Note can speed up the understanding process. For an unexpected dependency, the first attached Source Stick Note to that dependency has usually enough information to uncover the rationale for such a dependency. This note corresponds to the first change that introduced this dependency in the software system. As for a missing but expected dependency that may have existed in the past, we found that the last Source Sticky Note attached to that dependency usually has enough details to uncover the rationale for such a dependency. To summarize for unexpected dependencies, we recommend reading the Source Sticky Notes in chronological order. As for expected but missing dependencies, we suggest reading the Source Sticky Notes in reverse chronological order.

The method of attaching Source Sticky Notes to static dependencies described in this subsection is a simplification of our actual implementation. A more detailed explanation is presented in [HH04a] (*see* Chapter 3). Several optimizations are done to avoid re-parsing the revisions of files and to speed up the identification of dependencies. For a large system, such as NetBSD with around ten years of development, building the historical dependency graph takes over twelve hours. This is due to the long history of the project, the large size of its code base and the I/O intensive nature of our sticky notes recovery approach. Luckily, this process

needs to be done only once with the results stored in an XML file which is reused throughout the investigation process. As the software system evolves, only the new revisions in the source control system need to be analyzed to attach sticky notes corresponding to new changes to modified dependencies. The new sticky notes are appended to the previously generated XML file. By keeping the Source Sticky Notes up to date developer can use them during the development to understand the rationale behind the interactions among the various entities in a software system.

5.6 Case Study

To validate the usefulness of our approach we carried out a case study on *NetBSD*. We chose NetBSD as our case study for two reasons:

- NetBSD is a large long lived complex software system. It is being developed by a large number of developers and runs on over thirty hardware platforms.
- In addition, NetBSD (in particular the virtual memory component) was used by Murphy *et al.* as a case study in [MNS95] to demonstrate the usefulness of the reflexion framework. By using the same case study system, we can reuse the published conceptual view with its same mapping of source file to conceptual subsystems. This allows us to focus on showing the benefits of our approach in speeding up the investigation of gaps and improving the understanding of large software systems.

Figure 5.6 shows the conceptual view of the virtual memory component in the NetBSD operating system. In contrast to the figure shown in [MNS95], we focus only on the six main subsystems and we show a dependency between two subsystems if they use a function, macro, data type or a variable defined in another subsystem. Following the steps described by reflexion framework (see Figure 5.2, we create the reflexion diagram shown in Figure 5.7

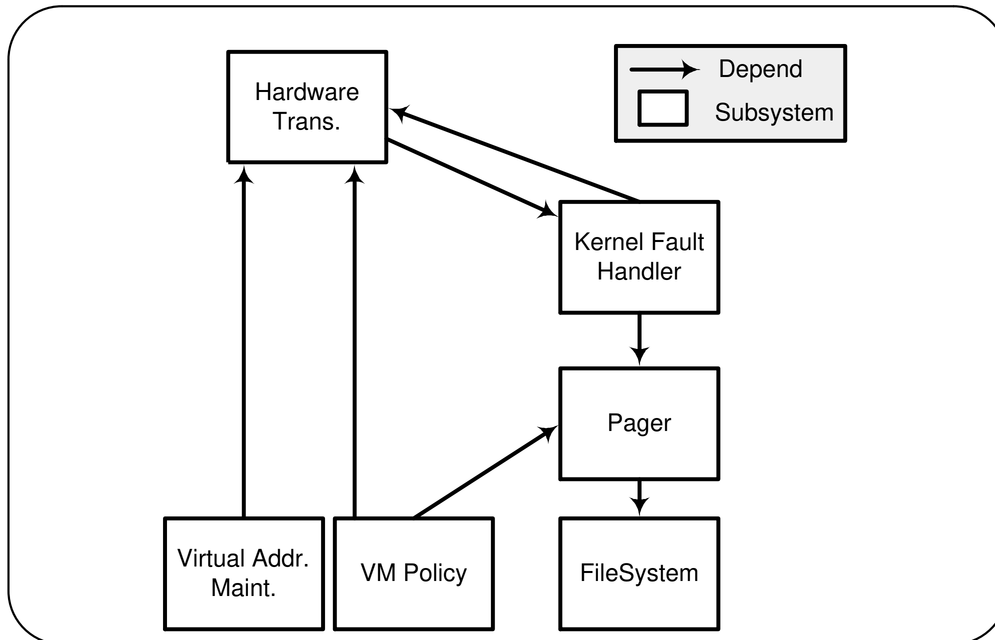

Figure 5.6: Conceptual View of the NetBSD Virtual Memory Component

We begin by observing that there are no *absence* dependencies, which is a common situation in most systems we have studied. It is a very rare case to find missing expected dependencies, instead the more common case is to find a large number of *divergences* - such is the case for NetBSD. We find that we have eight unexpected dependencies - the dotted arrows in Figure 5.7.

To understand the rationale for each of these dependencies, it would seem that we need to study the source code and consult members of the development team. This would be a time consuming task, due to the size of the source code and the size of the development team which is scattered throughout the world. Instead we use the historical dependency data with its sticky notes to speed up the process and to focus on the most troublesome dependencies. We start by investigating *when* did these dependencies appear in the source code. To our surprise, all of the dependencies except two have been in the source code since day one. Thus, we consider

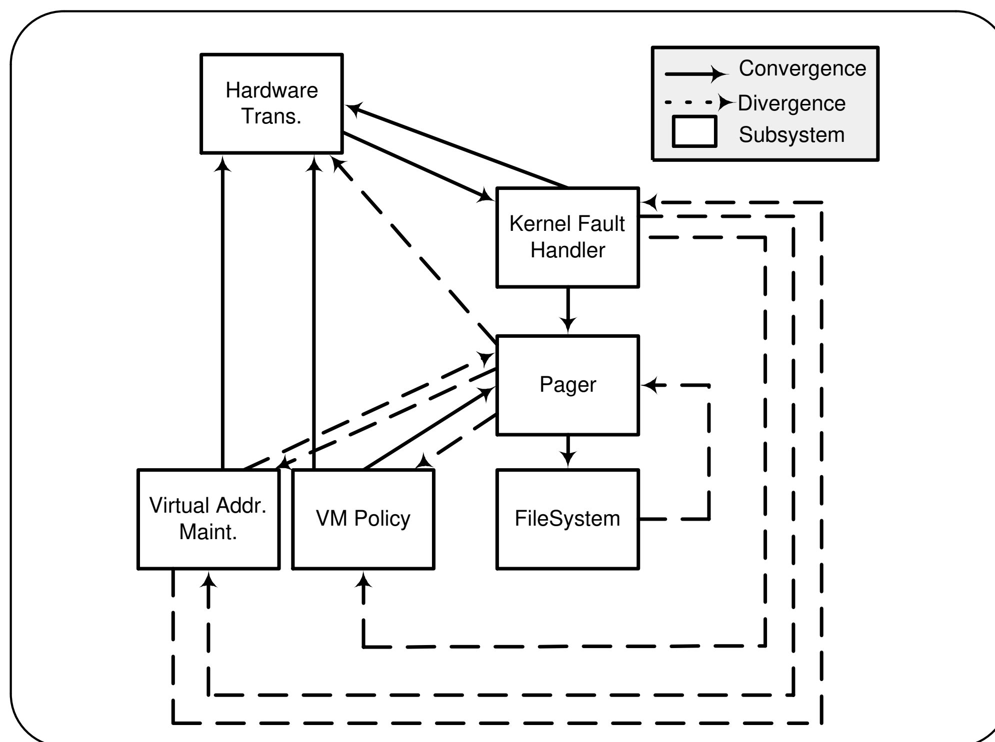

Figure 5.7: Reflexion Diagram for the NetBSD Virtual Memory Component

these seven dependencies not to be as critical, as they have apparently been part of the original code and have not been introduced due to decays in the design. It may be the original implementation had weaknesses but for now we focus on the two unexpected dependencies that were added after the start of the project, they are:

- The dependency from *Virtual Address Maintenance* to *Pager*.
- The dependency from *Pager* to *Hardware Translation*.

Investigating the dependency from the *Virtual Address Maintenance* to *Pager*, we ask *what* is the reason for the creation of such dependency. Given this is an unexpected dependency we look at the attached Source Sticky Notes in chronological order. We look at the first Source Sticky

Note (shown in Figure 5.8). The note shows the source code dependency *which* caused the dependency between these two subsystems. The note also records the name of the developer *who* introduced the dependency and *when* it was introduced. Furthermore, the note displays the comment entered by the developer when the change was performed. This comment gives the rationale (*why*?) for this dependency.

Which?	vm_map_entry_create (in src/sys/vm/Attic/vm_map.c) *depends on* pager_map (in /src/sys/uvm/uvm_pager.c)
Who?	cgd
When?	1993/04/09 15:54:59 Revision 1.2 of src/sys/vm/Attic/vm_map.c
Why?	from sean eric fagan: it seems to keep the vm system from deadlocking the system when it runs out of swap + physical memory. prevents the system from giving the last page(s) to anything but the referenced "processes" (especially important is the pager process, which should never have to wait for a free page).

Figure 5.8: Source Sticky Note for Dependency from *Virtual Address Maintenance* to *Pager*

We conclude that this dependency was added to prevent the system from deadlocking under special circumstances. We can investigate other Source Sticky Notes attached to the dependency between these two subsystems if needed.

We now focus on the other unexpected dependency – the dependency from the *Pager* to *Hardware Translation* subsystem. Since this is another unexpected dependency, we read the Source Sticky Notes attached to the dependency in chronological order. The first Source Sticky Note (shown in Figure 5.9 uncovers the rationale for such dependency. The dependency was introduced to fix a bug on multiple process (MP) systems.

In this subsection, we have shown how we can easily and rapidly investigate unexpected dependencies. A large number of unexpected depen-

Which?	uvm_pagermapin (in src/sys/uvm/uvm_pager.c) *depends on* pmap_kenter_pgs (in src/sys/arch/arm26/arm26/Attic/pmap.c)
Who?	thorpej
When?	1999/05/24 23:30:44; Revision 1.17 of src/sys/uvm/uvm_pager.c
Why?	Don't use pmap_kenter_pgs() for entering pager_map mappings. The pages are still owned by the object which is paging, and so the test for a kernel object in uvm_unmap_remove() will cause pmap_remove() to be used insteadof pmap_kremove(). This was a MAJOR source of pmap_remove() vs pmap_kremove() inconsistency (which caused the busted kernel pmap statistics, and a cause of much locking hair on MP systems).

Figure 5.9: Source Sticky Note for Dependency from *Pager* to *Hardware Translation*

dencies have been in the source since the start of the project. For these initial dependencies, we can use the same approach presented in this subsection. For example, investigating the reason for the unexpected dependency from the *Hardware Translation* to the *VM Policy* subsystem, the first Source Sticky Note does not reveal much about the rationale for the dependency other than saying that the project has commenced. We examine subsequent Source Sticky Notes to discover that this dependency is due to the same reasons as the investigated unexpected dependency from the *Pager* to the *Hardware Translation* subsystems.

5.6.1 Investigating Removed Dependencies

In the NetBSD case study, we did not find any expected dependencies that were missing in the implementation of the system. A study of the history of NetBSD shows that some dependencies existed at some point in time but are no longer there. Examples of such dependencies are:

- *Filesystem* to *Virtual Address Maintenance*.

- *Hardware Translation* to *VM Policy*.

Examining the Source Sticky Notes attached to the missing dependencies, we can discover the rationale for the removal of a dependency. We read the last Source Sticky Note attached to a removed dependency as it corresponds to the change that removed the dependency and would ideally give us the rationale for removing the dependency. For the first case, we see that, the dependency was removed as it was the result of a fix to a previous incorrect change (see Figure 5.10).

Which?	mfs_strategy (in.src/sys/ufs/mfs/mfs_vnops.c) *depends on* vm_map (in src/sys/vm/Attic/vm_map.h)
Who?	thorpej
When?	2000/05/19 20:42:21; Revision 1.23 of src/sys/ufs/mfs/mfs_vnops.c
Why?	Back out previous change; there is something Seriously Wrong.

Figure 5.10: Source Sticky Note for Dependency from *File System* to *Virtual Address Maintenance*

As for the *Hardware Translation* to *VM Policy* dependency, the last sticky note attached to that dependency indicates it was removed as part of a clean up and re-organization of the include files in the software system.

5.6.2 Discussion of Results

In this case study, we have shown the benefits of using historical data stored in source control systems to understand the dependencies between the subsystems of a large software system. The approach is highly dependent on the quality of comments and notes entered by developers when they perform changes to the source code. Luckily for many large software systems (in particular open source systems [CCW+01]), these comments are considered as a means for communicating the addition of new features and narrating the progress of the project to the other developers.

Hence developers are willing to put effort into entering correct and useful comments. This may not be the case for other systems. For these other systems where developers do not enter useful comments in the source control system, the source code remains the definitive and only option for investigating dependencies.

Throughout the investigation, we found ourselves performing three types of operations. Given a particular dependency, we wanted to retrieve the initial, last or all Source Sticky Notes attached to it. These operation are performed very fast (*interactively*) in contrast to building the historical dependency graph which requires many hours to generate. In the current implementation the system is text based but integrating such a system with a graphical interface would be beneficial. It would permit developers to simply right click on an unexpected dependency and a number of relevant Source Sticky Notes could pop up in a floating window.

This chapter and case study focused on using Source Sticky Notes to enhance the understanding of the architecture of software systems. Throughout the architecture understanding process the source code of the software system does not change, instead the main emphasis is on improving and enhancing the conceptual understanding of the developer so the conceptual understanding and the concrete implementation no longer have gaps between them. Another possible application for Source Sticky Notes is for architecture repair. The architecture repair process focuses on understanding the architecture of a software system, and on performing changes to either the conceptual understanding or to the system implementation to bridge the gap [TGLH00]. Source Sticky Notes can assist the developer in performing the changes to the source code during the architecture repair process as well.

5.7 Related Work

Several researchers have proposed the use of historical data related to a software system to assist developers in understanding their software

system and its evolution. Chen *et al.* have shown that comments associated with source code modifications provide a rich and accurate indexing for source code when developers need to locate source code lines associated with a particular feature [CCW+01]. We extend their approach by mapping changes at the source line level to changes in source code entities, such as functions and data structures, and the dependencies between them. Furthermore, we map the changes to dependencies between source code entities.

Murphy *et al.* argued the need to attach design rationale and concerns to the source code [BMS03, RM02]. They presented approaches and tools to specify and attach rationale to the appropriate source code entities. The processes specified in their work are manual and labor intensive, whereas our approach uses the source code comments and source control modification comments to automatically build a structure to assist developers in maintaining large code bases. Since our approach is automated, we avoid the problem of trying to get developers to specify, attach, and maintain this rationale.

Bratthall *et al.* have shown the significance of design rationale in assisting developers perform code changes for some software systems [BJR00]. Our approach provides a tool to recover some of the rationale automatically. Keller *et al.* suggested the recovery of patterns from the source code as a good indicator of decision rationale [KSRP99].

Design rationale includes: the issues addressed, the alternatives considered, the decision made, the criteria used to guide the decision, and the debate developers went through to reach such decision [BD00]. Our approach assumes that the text entered by the developer performing a change will cover some of these points, hence it will be useful in recovering part of the rationale. Richter *et al.* offer support to recover the full design rationale [RSA99]. They propose a tool to capture discussions and drawings during architectural meetings. These captured meetings should ideally contain enough information to assist in recovering the rationale of a system. Their system provides no benefit for legacy systems where such

meetings have not been captured.

Lastly, Cubranic and Murphy presented a tool which uses other types of captured project discussions such as bug reports, news articles, and mailing list posting to suggest pertinent software development artifacts [CM03]. The suggestions by their tool could be used to uncover the rationale for various architecture decisions. Compared to our approach, the information returned by their tool are numerous and are not as detailed as ours. Their tool may be beneficial when our approach is not able to return sufficient results, or if developers would like to gain more details about particular decisions. For example if an unexpected dependency has always existed since the beginning of the project our approach won't be able to provide the rationale for its existence as there won't be any modification records in the source control for it. Hence, using other types of captured project discussions may assist the developer in recovering the rationale for that unexpected dependency.

5.8 Conclusion

Much of the knowledge about the design of a system, its major changes over the years and its troublesome subsystems lives only in the brains of its developers. Such live knowledge is sometimes called *wet-ware*. When new developers join a team, mentoring by senior members and informal interviews are used to give them a better understanding of the system. Such basic understanding is rarely enough to maintain a software system. Therefore developer spend long periods of time hypothesizing about the state of the software system, comparing their hypotheses/assumptions with the actual implementation, and investigating any gaps they discover between their understanding and the actual implementation.

Static dependencies give us a current fixed view of the software system without details about the rationale, the history, or the people behind the dependency relations. Data stored in source control repositories provides a rich resource to assist developers in understanding large and complex

software systems. Using this data, we are able to automatically attach *Source Sticky Notes* to static dependencies. These notes record various properties concerning a dependency such as the time it was introduced, the name of the developer who introduced it, and the rationale for adding it.

Source Sticky Notes assist developers as they investigate dependencies in large software systems, by annotating the current structure of the software system with valuable information. This information links implementation entities to higher level constructs and provides a historical record of the evolution of the system and its rationale.

Although our concentration in this chapter has been on using Source Sticky Notes to understand software architecture, the benefits of these notes are abound. They can assist in other tasks such componentization, repairing decaying structures, or large scale refactoring. By distilling the pearls of wisdoms stored deep inside source control systems, we can assist developers understand the state of their project and plan confidently for its future.

In the following chapter, we continue on studying the use of the historical information stored in source control systems to support software developers. We use such historical information to gauge the benefits of adopting new maintenance tools or strategies.

CHAPTER 6

Replaying Development History to Assess the Claimed Benefits of Code Maintenance Tools and Strategies

Nowadays practitioners are faced with many tools and methodologies promising to ease their maintenance tasks. Code restructuring methodologies claim to ease software evolution by localizing changes. Development environment tools assert their ability to assist developers in propagating changes. Static source analysis tools (such as lint) promise to point out error prone code. Unfortunately, such claims and promises are rarely substantiated or tested although the cost of adopting such tools and approaches is high and the risks of failures are even higher.

We propose to use the historical information stored in software repositories (such as source control systems) to assess such claimed benefits.

We present the Development Replay (DR) approach which reenacts the changes stored in the source control repositories using a proposed tool or strategy. We present a case study where the DR approach is used to empirically assess and compare the effectiveness of several not-yet-existing tools which promise to assist developers in propagating code changes. The approach is illustrated by a case study of five large open source systems with a total of over 40 years of development history.

6.1 Introduction

NEW tools, languages, strategies and approaches are being continuously proposed by researchers and industry. Software developers need to determine if status quo is the best option or if they should consider adopting such novel ideas. Such tools and approaches need to be investigated using careful, rigorous software engineering experimentation before they can be adopted by practitioners [FPG94, PPV00, KPJ+02, Gla03a].

Laboratory experiments are usually not able to simulate real life industrial settings and tend to be run for a short period of time, while industrial studies usually require a long and costly commitment by the practitioners. An ideal approach should strike a balance between the low cost, short duration, fast results of laboratory experiments which permit the analysis of a variety of tools or approaches; while limiting the costs and time needed for industrial studies. Such an ideal approach would expedite studying the effectiveness of specific tools or strategies over an extended period of time. In this chapter, we propose an empirical approach that attempts to strike such a balance.

Consider the release of a new programming language which promises to get rid of memory leaks by performing automatic garbage collection. Software developers would like to determine the potential benefits of migrating their code to such a language. Clearly there may be other benefits

but if the promise of getting rid of memory leaks is the main driver to adopt such a language, then developers should gauge the potential benefits. One thought is to consider prior faults during the lifetime of the project, if none of the prior faults were memory leaks then the potential benefits of adopting such a programming language are likely to be minimal. It may be the case that the tools, processes, techniques, and expertise in place already at their organization are able to deal successfully with memory tracking issues. For instance techniques such as code reviews, or specialized memory tracking libraries may be able to prevent memory leaks and no new programming languages are needed.

Software repositories such as source control repositories, bug tracking repositories, and archived email communications track the evolutionary history of a software project. We could use the information stored in these repositories to assess the claimed benefits of new tools and approaches. A number of claims and approaches cannot be studied through the historical information of a project such as the ability of a tool to reduce developer's stress or the ability of a tool to assist developers in gaining a better understanding of the source code. Nevertheless, several types of claims and approaches can be easily examined, for example:

- **The benefits of code restructuring strategies on localizing changes to the source code**: Refactoring strategies, object oriented technologies, and aspect oriented techniques aim to restructure the source code to assist developers in understanding the code better and to ease future changes by localizing changes. Using historical information, we could study the potential benefits of adopting such restructurings on localizing changes. Prior changes to the source code could be used to study the locality of changes using the newly proposed restructuring. For example, if all or most prior changes occurred within a small number of files, then the benefits of a restructuring on localizing changes will be minimal.
- **The benefits of static source code analysis tools on pointing out error prone code**: Static source code analysis tools such as lint

perform static analysis on the source code and mark areas that are likely to have faults in them. Using historical information, one could study the potential benefits of adopting such tools. For example, we could run the tool on prior versions of the source code then examine the project's history to determine if warnings issued by the tools correlated with actual faults in the source code or if they did not.

- **The benefits of development environments in assisting developers to propagate changes**: It is essential that changes are propagated correctly to the appropriate locations in the code, otherwise bugs are likely to occur due to inconsistencies in the source code. There exists different heuristics and tools to propagate changes. Using historical information, one could study the potential benefits of adopting such tools. We could compare the performance of several tools which use various techniques to propagate changes by studying their performance using old changes. In this chapter, we give an example for this particular case.

In this chapter, we present the *Development Replay (DR)* which permits us to replay the history of a software project since its inception till any moment in its history using information recovered from its source control repository. We can determine at any moment the *state of the software project*, such as the current developers that worked on or are currently working on the project, the cooperation history between these developers, and the structure of the dependencies between the source code entities (such as functions and variables). We can also recover *change sets* from the source control system. These change sets track the source code entities that were modified together to implement or enhance features, or to fix bugs. Using this historical information (the change sets and the state of the software project), we present an example of comparing the benefits of several enhanced dependency browsers in assisting developers as they propagate changes. Such dependency browsers could be integrated in development environments to support developers who are maintaining and evolving large software systems.

6.1.1 Organization of Chapter

The chapter is organized as follows. In Section 6.2, we discuss the change propagation process and explain how it could be used to measure the benefit of adopting new tools. Then in Section 6.3, we present several metrics to assess some of the claimed benefits of adopting new change propagation tools. In Section 6.4, we give an overview of the DR approach and the software infrastructure we developed to permit the assessment of software maintenance tools and strategies. We present a critical analysis of the limitations of the results derived through the DR Approach. In Section 6.5, we present a case study which applies the DR approach to measure the effectiveness of several tools that assist developers to propagate changes in large software systems. Section 6.6 discusses related work. Section 6.7 concludes the chapter with comments about the benefits and limitations of the DR approach.

6.2 The Change Propagation Process

The dangers associated with not fully propagating changes have been noted and elaborated by many researchers. Parnas tackled the issue of software aging and warned of the ill-effects of *Ignorant Surgery*, modifications done to the source code by developers who are not sufficiently knowledgeable of the code [Par94]. Brooks cautioned of the risk associated with developers losing grasp of the system as it ages and evolves [Bro74]. Misunderstanding, lack of experience and unexpected dependencies are some of the reasons for failing to propagate changes throughout the development and maintenance of source code. Mis-propagated changes have a high tendency to introduce difficult to find bugs in software systems, as inconsistencies between entities (such as functions) increase.

Work by Atkins *et al.* [ABGM99] and surveys by Sim *et al.* [SCH98] indicate that software developer would like to have tools to assist them in performing changes to the source code. These tools could guide developers by either informing them about code entities (such as functions) to

change, or assisting developers in performing the actual change (such as tools that automate the code refactoring process). For our work, we focus on the tool's ability to inform developers which entities to change. A desired software development tool is one that would ensure that a developer changing a particular source code entity (such as a function) is informed of all relevant code entities to which the change should be *propagated*.

We now examine the process of change propagation and give a breakdown of the various steps it involves. In the following section, we present metrics to measure the performance of a tool in propagating changes.

We define *change propagation* as the changes required to other entities of the software system to ensure the consistency of assumptions in a software system after a particular entity is changed. For example, a change to a function that writes data to a file may require a change to propagate to the function that reads data from file. This would ensure that both functions have a consistent set of assumptions. In some cases no change propagation may be required; for example when a comment is updated, the indentation of the function text is changed, the internal logic of a function is reworked, a locally scoped variable is renamed to clarify its use, or local optimizations are performed. Though developers have to tackle the problem of change propagation and locate entities to change in a software system to ensure its consistency on a daily basis, this problem and its surrounding challenges are not clearly understood. Nevertheless several development tools have been proposed to assist developers in propagating changes.

In Figure 6.1, we propose a model of the change propagation process. Guided by a request for a new feature, a feature enhancement, or the need to fix a bug, a developer determines the initial entity in the software system that must change. Once the initial entity is changed, the developer then analyzes the source code to determine if there are other entities to which the change must be propagated. Then she/he proceeds to change these other entities. For each entity to which the change is propagated the propagation process is repeated. When the developer cannot locate

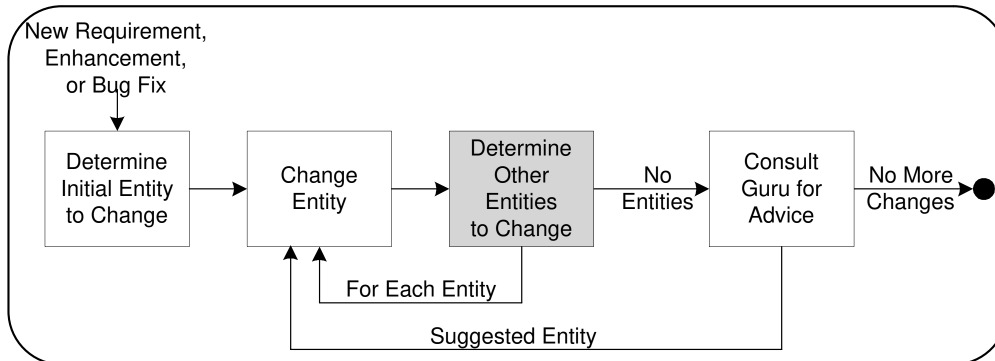

Figure 6.1: Model of the Change Propagation Process

other entities to change, she/he consults a *Guru*. If the Guru points out that an entity was missed, then it is changed and the change propagation process is repeated with that entity. This continues until all appropriate entities have been changed. At the end of this process, the developer has determined the *change set* for the new requirement at hand. Ideally all appropriate entities should have been updated to ensure consistent assumptions throughout the software system.

The Guru could be a senior developer, a test suite, or even a compiler. Usually consulting the senior developer is not a practical option, as the senior developer has limited time to assist each developer. Nor is it a feasible option for long lived projects where such knowledgable developer rarely exists. Moreover, complete and extensive test suites rarely exist. Therefore, developers find themselves forced to use other forms of advice/information such as the advice reported by a development tool to perform the activities in the grey highlighted box in Figure 6.1. Ideally developers would like to minimize their dependence on a Guru. They need software development tools that enable them to confidently determine the need to propagate changes without having to seek the assistance of gurus which are not as accessible and may not even exist.

6.2.1 Information Sources Used to Propagate Changes

Program dependency relations, such as *call* and *use* have been proposed and used as indicators for change propagation. For example, if function A *call*s function B, and B was changed then function A is likely to change as well. If function A were to change then all other functions that call A may need to change as well. This ripple effect of change progresses until no more changes are required to the source code [YNTL88].

In search of other entities to propagate a change, developers depend on a number of information sources to assist them in locating other entities that should change. Builders of development tools depend on these information sources as well to suggest entities to which a change may propagate. We now describe some of the possible sources of information:

6.2.1.1 Entity Information

The change propagation process may depend on the particular changed entity. For example, a change may propagate to other entities interdependent on the changed entity according to relations such as:

- A *Historical Co-change* records that one entity changed at the same time as another entity. If entity A and B changed together in the past, then they are related via a Historical co-change relation and are likely to change together in the future.
- A *Code Structure* relation records static dependencies between entities. *Call*, *Use*, and *Define* relations are some possible sub-relations:
  - The *Call* relation records that a function calls another function or macro.
  - The *Use* relation records that a function uses a variable.
  - The *Define* relation records that a function defines a variable or has a parameter of a particular type. For example F *Define* T, means F defines a variable of type T.

- A *Code Layout* relation records the location of entities relative to classes or files or subsystems in the source code. Containers such as files and classes are good indicators of a relation between entities, and related entities tend to change together.

6.2.1.2 Developer Information

The change propagation process may be dependent on the fact that the same developer changed other entities recently or frequently. This is based on the observation that over time developers gain expertise in specific areas of the source code and tend over time to modify a limited set of entities in their acquired areas of expertise [BH99].

6.2.1.3 Process Information

The change propagation process may be dependent on the process employed in the development. For example a change to a particular entity may propagate changes to other frequently or more recently changed entities independent of the specific entity that changed, as these recently changed entities may be responsible for a specific feature being modified.

6.2.1.4 Textual Information

The change propagation process may be dependent on the fact that change propagates to entities with similar names, as the similarity in naming indicates similarities in the role of the entities and their usage, as suggested by [AL98] who used such information to improve automatic clustering of files. It may be also dependent on the fact that entities have similar tokens in common in their comments [Shi03].

Other sources of information may exist. Also the aforementioned information sources can be combined and extended in various ways. For example, another possible source is the co-call information, where A and B both *call* C. A and B may implement similar functionality and a change to

A may propagate to B. Developers use such information sources to assist them propagate changes to the appropriate part of the code. Development tools use this information as the basis of heuristics that are used to suggest entities to the developers to assist them in the change propagation process.

Developers spend a considerable amount of time to correctly propagate a change to other entities. This is a labor intensive task that is error prone. Change propagation is a central aspect of software development and developers are always in search for tools to assist them in this process. In the following section, we investigate a number of metrics that could be used to measure the performance of tools such as dependency browsers that are provided by modern software development environments.

6.3 Measuring The Performance Of a Tool in Propagating Changes

Developers seek tools that can assist them to perform changes quickly and accurately. By quickly, we mean tools that would reduce the time needed to perform the change. By accurately, we mean tools that would ensure that a change to a source code entity is propagated to all relevant code entities. In the ideal case, a development tool would correctly suggest all the entities that should be changed without the developer resorting to asking the Guru for advice. The worst case occurs when the Guru is consulted to determine each entity that should be changed. Referring back to the change propagation model shown in Figure 6.1, we would like to minimize the number of times the Guru suggests an entity to change.

The metrics discussed in this section will focus on the accuracy of the tool instead of the time required to perform the change itself. The time required to perform a change is likely to be highly dependent on the developer performing the change and the type of tool support (*e.g.* code editor) provided to the developer. Moreover, the time required may be difficult

to track since most practitioners rarely record the time spent on each change.

6.3.1 A Simple Example

Consider the following example, Dave is asked to introduce a new feature into a large legacy system. He starts off by changing initial entity A. After entity A is changed, a tool suggests that entities B and X should change as well as. Dave changes B, but then examines X and realizes that it does not need to be changed. So Dave does not need to perform any change propagation for X. He then asks the tool to suggest another entity that should change if B were changed. The tool suggests Y and W, neither of which need to change – therefore Dave will not perform any change propagation for Y or W. Dave now consults Jenny, the head architect of the project (the Guru). Jenny suggests that Dave should change C as well. Dave changes C and asks the tool for a suggestion for an entity to change given that C was changed. The tool proposes D. Dave changes D and asks the tool for new suggestions. The tool does not return any entities. Dave asks Jenny who suggests no entities as well. Dave is done propagating the change throughout the software system.

6.3.2 Performance Measures for a Single Change Set

There exists a variety of metrics that could be used to measure the performance of a tool in assisting developers perform changes. As highlighted earlier, we will focus on the ability of the tool to locate the relevant entities that should be changed instead of focusing on the time required to perform the changes themselves.

6.3.2.1 Defining Recall and Precision

To measure the performance of a tool in propagating changes, we use traditional information retrieval concepts: recall and precision. For our sim-

ple example, Figure 6.2 shows the entities and their interrelationships. Edges are drawn from A to B and from A to X because the tool suggested that, given that the change set contains A, it should contain B and X as well. For similar reasons, edges are drawn from B to Y and W, and from C to D. We will make the simplifying assumption that a tool provides *symmetric suggestions*, meaning that if it suggests entity F when given entity E, it will suggest entity E when given entity F. We have illustrated this symmetry in Figure 6.2 by drawing the edges as undirected edges.

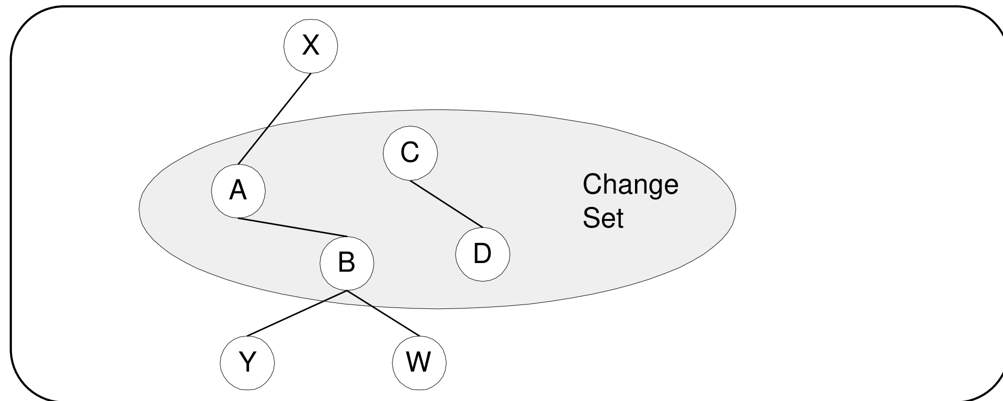

Figure 6.2: Change Propagation Graph for the Simple Example - An edge between two entities indicates that a tool suggested one when informed about changes to the other one.

The total set of suggested entities will be called the *Predicted* set; *Predicted* = {B, X, Y, W, D}. The set of entities that needed to be predicted will be called the *Occurred* set; *Occurred* = {B, C, D}. Note that this does not include the initially selected entity (A), which was selected by the developer (Dave) and thus does not need to be predicted. In other words, *Occurred* = *ChangeSet* - {*InitialEntity*}.

We define the number of elements in *Predicted* as P (P = 5), and the number of elements in *Occurred* as O (O = 3). We define the number of elements in the intersection of *Predicted* and *Occurred* (this intersection

is $\{$B, D$\}$) as PO (PO = 2). Based on these definitions, we define:

$$Recall = \frac{PO}{O}$$
$$Precision = \frac{PO}{P}$$

In our example, $Recall = \frac{2}{3} = 66\%$ and $Precision = \frac{2}{5} = 40\%$. The rest of this chapter will use these definitions of *Recall* and *Precision*.

We will make another simplifying assumption, which is that each prediction by a tool is based on a single entity known to be in the change set. For example, a heuristic may base a prediction on a single element C known to be in the change set, and not on a set of entities such as $\{$A, C$\}$ known to be in the change set. A further assumption is that the developer (Dave) will query the tool for suggestions based on every so far suggested entity (which is determined to be in the change set) before querying the Guru (Jenny). An implication of our simplifying assumptions is that the tool may not do as well in making predictions as they would without these assumptions. Nevertheless, this limitation is not a concern as we are more interested in comparing the relative difference between several development tools using the same precision and recall model.

Our simplifying assumptions imply that *the ordering of selections and queries to a heuristic are immaterial*. For example, Dave might initially select entity B or C or D instead of A. Further, if Dave had a choice of queries to the tool (say, to get suggestions based on either entity M or N), either query could be given first. Regardless of the selections and ordering, the values determined for *Precision* and *Recall* would not be affected. The *Development Replay* approach depends on change sets that are recovered from the source control system to measure the performance of a tool. These recovered change sets do not record the ordering of selections. So, not only do our assumptions simplify our analysis, they avoid the need for information that is not available in source control systems.

There is an interesting implication of our assumptions, as we will now explain. In Figure 6.2, within the change set, there are two connected

components, namely {A, B} and {C, D}. With an *ideal* tool, which could be used to predict all entities in a change set without recourse to a Guru, there would necessarily be exactly one connected component. If there is more than one connected component, each of these, beyond the initial one, implies a query to a Guru. In other words, if CC is the number of connected components and G is the number of queries to the Guru, then G = CC - 1. With an ideal tool, CC = 1 and G = 0, while with the worst tool, CC = N and G = N - 1, where N is the number of entities in the change set. Based on our previous definition of $Recall$, it can be proven that

$$Recall = 1 - \frac{(CC - 1)}{(N - 1)}$$

This is the $Recall$ formula actually used in our analysis.

The F-measure which is a weighted harmonic mean of recall and precision metrics could be used [vR79]:

$$F_\beta = \frac{(\beta^2 + 1) * Precision * Recall}{\beta^2 * Precision + Recall}$$

Here β ranges between 0 and infinity. β values give varying weights to recall and precision. For example to indicate that recall is half as important as precision, β would have a value of 0.5. A value of 1.0 indicates equal weight for recall and precision. A value of 2.0 indicates that recall is twice as important as precision. F_0 is the same as precision, F_∞ is recall. Higher values for F correspond to higher effectiveness. Due to the dangers of missing to propagate a change to a source code entity, it may be desirable to assign β a value of 2.0 to indicate the importance of recall. Alternatively a senior developer may prefer not to waste a lot of time investigating irrelevant entities, therefore she/he would prefer a tool with high precision. She/he would use a β value of 0.5. For the results presented in this chapter, we show the precision and recall values as they are more intuitive to reason about their meaning. We then use the

F-measure (F_1) to combine the precision and recall values into a single value to subjectively compare the performance of development tools. The maximum possible value for the F-measure is 1.0 when both recall and precision are 1.0. The lowest possible value for F-measure is 0 when both recall and precision are zero. An information retrieval practical range for the F-measure is between 0.44-0.48 where precision usually lies in the 35%-40% range and recall is around 60% [Bel77].

6.3.2.2 Other Performance Metrics

Another possible performance metric is a utility function which assigns a value or cost to each suggested entity. Such a measure is commonly used in the TREC filtering task which focuses on sorting through large volumes of dynamically generated information and presenting to the users the information details which are likely to satisfy their current information requirements [Hul98]. The larger the utility score, the better the filtering or, in our case, the more effective the development tool is in assisting developers perform the change. For example,

$$U = 6 * PO - 5 * (P - PO)$$

The utility of the tool for each change set is added up and a final total utility is used to measure the performance of the tool for all the change sets. To prevent a large change set from negatively affecting the overall outcome, a minimum utility is defined (U_{min}) that would be used if the utility for a particular change set is less than U_{min}. The results presented in this chapter do not use the utility function technique to study a tool's performance.

6.3.3 Performance Measures for Multiple Change Sets Over Time

In the previous subsection, we presented metrics to measure the effectiveness of a tool in assisting a developer performing a single change (*single*

change performance metrics). We are more interested in measuring the performance of a tool when used to perform a large number of changes over an extended period of time. In this section we present metrics to assess the long term effectiveness of a tool.

6.3.3.1 Average Performance of a Tool

To measure the performance over time for multiple change sets, the simplest measure is the average performance of a single change performance metric such as the average of the recall or precision for all change sets. To calculate the average we sum up each change set and divide by the number of studied change sets (M):

$$
\begin{aligned}
Average\ Recall &= \frac{1}{M} * \sum_{i=1}^{M} \Big(Recall_i \Big) \\
Average\ Precision &= \frac{1}{M} * \sum_{i=1}^{M} \Big(Precision_i \Big)
\end{aligned}
$$

6.3.3.2 Stability/Volatility of the Performance of a Tool

The use of an average to measure the performance has its limitations in particular, it does not take into account the fact that the performance of a tool may vary widely from one usage to the next. For example, a tool may perform remarkably well for a change set but its performance may be very disappointing for the following change set.

Developers are interested in not only the average performance of a tool but in the stability of its performance as well. Developers would like tools that consistently deliver reasonable performance instead of tools whose performance varies considerably. This is particularly a concern when the performance of a tool is studied over a long period of time. For example, a tool may be beneficial at the beginning of a project but as the project

ages and its design decays the tool may not be as helpful in propagating changes.

We use the standard deviation (σ) of the tool's performance over a large number of change sets to assess the stability of the tool's performance.

6.3.4 Relative Performance of a Tool Over Time

When studying the performance of a tool, practitioners are more interested in the relative performance of the tool. In other words, they investigate whether they should adopt tool A, or tool B, or whether they should stick with their current tools. In the previous subsection, we proposed the use of the average and the standard deviation to measure the performance of a single tool. We now focus on metrics to subjectively compare the performance of multiple tools over time.

Simply comparing the average tool performance for two tools is not sufficient. Instead we should ensure that the difference in average is statistically significant. We therefore need to perform a statistical test of significance. We use a statistical paired T-test and formulate the following test hypotheses:

$$H_0 : \mu(Perf_A - Perf_B) = 0$$

$$H_A : \mu(Perf_A - Perf_B) \neq 0,$$

where $\mu(Perf_A - Perf_B)$ is the population mean of the difference between the performance of each tool for the same change sets. When the tools have been used to assist in propagating changes for a large number of change sets, we can use a T-test. Alternatively, a non-parameterized test such as a U-test can be used when we have a smaller number of change sets [MW47].

If the null hypothesis H_0 holds then the difference in average is not significant. If H_0 is rejected with a high probability then we can be con-

fident about the performance improvements a tool is likely to offer developers performing changes to the source code and we could recommend the adoption of the tool by developers.

6.3.5 Relative Stability/Volatility of the Performance of a Tool

Whereas standard deviation is used to measure the stability of the performance of a tool, it has its limitations. For example if the development process itself is not stable with the team varying their focus or if the quality of the design of the software system is varying over time, then the standard deviation measure is likely to show high volatility. This volatility may be attributed to the development process itself instead of being solely due to the tool's performance. Therefore, we must compare the stability of a tool against other tools as well. A tool with high performance may be too volatile and may hinder its adoption by developers who seek a tool which they can depend on and trust for its consistency.

In this section, we discussed a number of performance metrics. We first focused on metrics that measure the performance of a tool in assisting a developer propagating a single change set throughout a software system. We then discussed the issues surrounding applying such performance metrics to measure the performance of a tool for a large number of changes over time. We highlighted the need to study the performance of an adopted tool and the stability of its performance as well. We also discussed the risks associated with the volatile performance of tools and asserted the need for tools with stable performance so developers would trust their suggestions.

Traditionally, researchers would need to conduct long term studies to perform such performance analysis for each studied tool. Using our definitions of recall and precision, they could measure the performance of tools by monitoring the change process and making developers use the tool to suggest entities to change. This is a time consuming process and

would require developers to adopt the tools in their development process. Also it would prevent the researchers from experimenting with several tools as they could only test one tool at a time. To overcome these limitations, we use the Development Replay (DR) approach to measure the effectiveness of several tools by replaying back the development history of a software system. The following section details the DR approach.

6.4 The Development Replay (DR) Approach

In this section, we present a software infrastructure and framework which permits researchers to write a description of a not-yet-existing or not-yet-adopted software development tool that would assist developers in propagating changes. The description is written in a programming language (Perl) and has access to a number of information sources that have traditionally been used to build tools that assist in change propagation such as the sources discussed in section 6.2.1. Using the tool's description, the framework uses *actual* change sets that have previously occurred during the development of a software system to measure the effectiveness of a tool in assisting developers performing the change propagations. The effectiveness is measured using the precision and recall metrics defined in the previous section. We end this section with a critical analysis of the limitations and applicability of the results derived using the DR approach.

The usage of the studied development tool is simulated to assist a developer perform a change set. The performance of the tool is measured for each change set in the history of a project and the overall effectiveness of the tool is reported. The reported results can be used to subjectively compare several software development tools.

To ensure the accuracy of the replay process, the state of the software project is tracked throughout the lifetime of a project. Hence when a particular change occurs to the software system, we can determine the state of the project at that exact moment in its history.

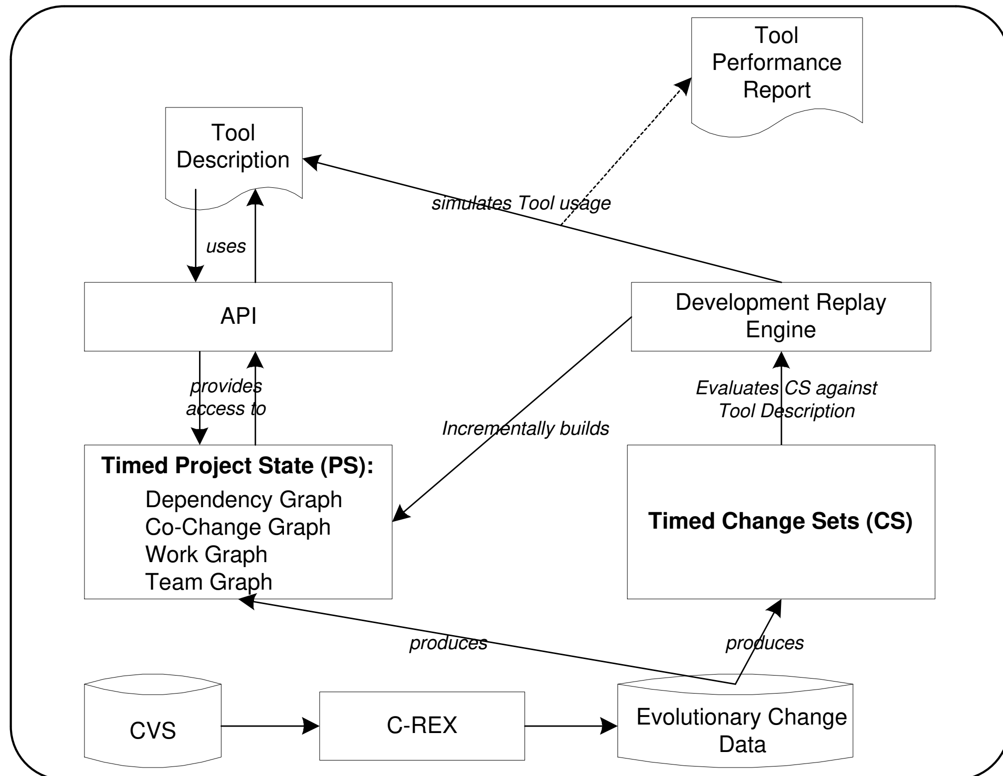

Figure 6.3: The Development Replay Infrastructure

The DR framework provides an API to access timed information concerning the entities, developers, change history, and naming information of the project. The information is timed in the sense that the API is given the exact time of the change that is being investigated and is able to report details about the project at that exact moment in time instead of reporting details for example about the most recent dependency structure of the software system. The simulated tool description can use the DR API to determine the project state information at any moment in time such as:

1. The dependencies among the source code entities in the software system such as function call, data usage, and type definition.
2. The co-change history between source code entities such as the fact that a change to function *writeToFile* is always associated with a

change to function *readFromFile*.

3. The name of the developers who modified each code entities.

4. The developers who worked on the same entities as a particular developer.

This information contains details up to the current replayed change. An overview of the DR infrastructure is shown in Figure 6.4. The DR permits tool adopters to encode the results produced by a not-yet-existing or a studied tool when told that a specific entity has changed. The tool description can use a rich API to gain access to a variety of information about the current state of the software project during the replay process. The timed project state and the timed change sets data are derived from source control systems using a specialized evolutionary extractor, called C-REX [HH04b] (*see* Chapter 3).

For each file in the software system, the source control system tracks its creation, and its initial content. In addition, it maintains a record of each change done to a file. For each change, a *modification record* stores the date of the change, the name of the developer who performed it, the specific lines that were changed (added or deleted), a detailed explanation message entered by the developer giving the reason for the change, and other files that were changed with it. To build a project state, the level at which the modification record stores change information (at the file level) is too high. C-REX preprocess and transforms the content of the source control system into an optimized and more appropriate representation. Instead of changes being recorded at the file level we record them at the source code entity level (function, variable, or data type definition). Then we can track details such as:

- Addition, removal, or modification of a source code entity. For example, adding or removing a function.

- Changes to dependencies between the modified entities and other source code entities. For example, we can determine that a function

no longer uses a specific variable or that a function now calls another function.

The C-REX extractor attaches additional information to each change set about the developer performing the change and the purpose of the change as well.

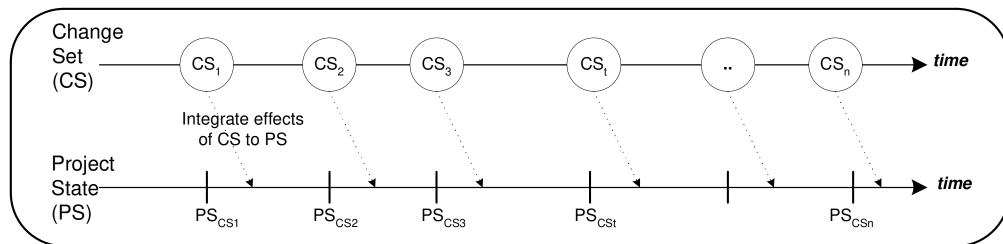

Figure 6.4: Maintaining the Project State Incrementally

Using the information produced by C-REX, the Development Replay engine (shown in Figure 6.3) can incrementally build a timed project state. Figure 6.4 gives an overview of how the DR engine reads the time encoded change sets, simulates the usage of a software development tool to perform the associated changes, then integrates the results back into project state. The project state contains all relevant information up to but not included the currently examined change set.

To measure the performance of a particular tool we simulate its usage to perform changes that are represented in the timed change sets recovered from the source control system. We examine sequentially through time all modification records that are not General Maintenance (GM) record and where no entities were added. For each modification record, the tool had to predict the change propagation process outlined in section 6.3. Then the performance of the tool is measured.

We note that we did not study all the change sets accessible through the DR infrastructure. Instead using a lexical technique, similar to [MV00], the DR infrastructure examines the content of the detailed message attached to each modification record and marks all General Maintenance

(GM) modifications which are mainly bookkeeping changes. These modifications do not reflect the implementation of a particular feature. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well. We do not consider these GM modifications as they are rather general and we do not expect any tool to predict the propagation of changes in these modifications.

We classified the remaining modification records into two types:

- Records where entities are added, such as the addition of a function, and
- Records where no new entities are added.

We chose to study the change propagation process using only modification records where no entities were added. This choice enables us to compare different change propagation tools fairly, as it is not feasible for any tool to predict propagation to or from newly created entities. We note that we still use the records where entities are added to build the historical co-change information but we do not measure the performance of any tool using these records. Furthermore, to avoid penalizing tools which make use of historical information as they work on building a historical record of changes to give useful suggestions, we do not measure the performance of the tool for the first 250 modification records for a software system.

Table 6.1 gives a breakdown of the different types of modification records in the software systems we studied. The studied modification records represent on average 60% of all the available records in the history of a project, after removing GM modifications and modifications where entities are added. We believe that the studied modification records are a representative sample of changes done to large software projects throughout their lifetime.

Application Name	All Records	GM Records	New Entities Records	Studied Records
NetBSD	25,839	6,204	4,086	15,567
	(100%)	(24%)	(16%)	(60%)
FreeBSD	36,635	7,703	8070	20,862
	(100%)	(21%)	(22%)	(57%)
OpenBSD	13,653	2,741	2,743	8,169
	(100%)	(20%)	(20%)	(60%)
Postgres	6,199	1,461	1,514	3,224
	(100%)	(23%)	(24%)	(52%)
GCC	7,697	901	1114	5682
	(100%)	(12%)	(14%)	(74%)

Table 6.1: Classification of the Modification Records for the Studied Systems

6.4.1 Threats to Validity and Limitations of Results Derived Through the DR Approach

The DR approach has a number of limitations. We now discuss these limitations. We also discuss the purpose of the DR approach and focus on the applicability of the results derived through it. The DR approach permits the empirical evaluation of some of the benefits of adopting particular software maintenance tools or strategies. All empirical research studies should be evaluated to determine whether they were able to measure what they were designed to assess. In particular, we need to determine if our findings that a particular tool is more effective than another tool are valid and applicable or are they due to flaws in our experimental design. Four types of tests are used [Yin94]: construct validity, internal validity, external validity, and reliability.

6.4.1.1 Construct Validity

Construct validity is concerned to the meaningfulness of the measurements – Do the measurements quantify what we want them to? The main

conjecture of our work is that developers desire the tool that permits them to perform changes quickly and accurately. We focus on the accuracy of the tool instead of the time required to perform the change itself. The time required to perform a change is likely to be highly dependent on the developer performing the change. Moreover, the time needed for each change is difficult to track since most practitioners rarely keep detailed records of their time.

The DR approach assumes that the accuracy of a tool in propagating changes is a sufficient reason to encourage the adoption of a particular tool. The approach does not tackle the issues of the tool's user interface and the developers interaction with the tool as well. For example a tool with a complex user interface may be abandoned by its users. Also some developers may be more proficient users of the tool than others. The approach also assumes that the training costs for adopting a tool is negligible and should not be a major concern. That last point may not be a major concern given we are interested in the long term benefits of adopting a particular tool. Nevertheless, it is a limitation of the approach.

6.4.1.2 Internal Validity

Internal validity deals with the concern that there may be other plausible rival hypotheses to explain our findings – Can we show that there is a cause and effect relation between the usage of a specific tool and the developers ability to propagate changes accurately throughout a software system, or are there other possible explanations?. The DR approach measures and compares the performance of software development tools by simulating their usage using *actual* change sets which are recovered from source control repositories. By keeping all other project variables constant and simulating the usage of a tool through play back we have a clear cause and effect relation between the simulated tool usage and the effectiveness of the tool. That said the introduction of a tool as part of the development process may affect the type of changes that developers are likely to perform. For example, a tool that reduces the time needed

to perform rather complex changes may encourage developers to perform more complex changes instead of opting for workarounds for these complex changes [BMSK02]. Moreover the performance of the tool may be dependant on other factors that are not modeled by the DR approach. Software development is a complex process with a number of factors and facets interacting together and affecting its outcome. The DR approach uses source control systems to recover and recreate the historical state of a software project, unfortunately a large number of issues and factors surrounding a software system are not present in the source control data (such as personal communications and knowledge that resides in the heads of the software developers). The DR approach can integrate additional knowledge present in other software development repositories such as mailing list repositories to assist in improving the accuracy of the development replay process. Unfortunately, automated integration of mailing list information is not an easy task as the data is not as structured as source code and change data which are stored in source control systems.

In our case study (presented later in this chapter), we make use of change sets recovered from source control systems of large open source projects to measure the performance of specific tools. We make the assumption that each change set contains only change that are related, *i.e.*, that involve a change propagation. In principle, it is possible that a developer may check in several unrelated entities as part of the same modification record. For our purposes, we assume that this occurs rarely. We believe that this is a reasonable assumption based on the development process employed by the studied open source projects and discussions with open source developers [BP03, Mit00, Wei03]. In most open source projects, access to the source code repository is limited. Only a few selected developers have permission to submit code changes to the repository. Changes are analyzed and discussed over newsgroups, email, and mail lists before they are submitted [CM03, MFH00, YK03]. We believe that this review process reduces the possibility of unrelated changes being submitted together in a modification record. Moreover, the review

process helps ensure that changes have been propagated accurately in most cases. Thus most change sets would contain a complete propagation of the changes to all appropriate entities in the software system. Recent work by [CSY+04] cautioned of the reliability of open source change logs. Change logs are summarizations of the purpose of changes that occurred to a software system as it evolves and are usually stored in a single file called the `ChangeLog` file. They are used to provide a high level overview of changes. The quality of these change logs is not a concern for the DR approach. The DR approach builds the change sets using the source control database instead of relying on change logs which omit a large number of details about the project's evolution.

6.4.1.3 External Validity

External validity tackles the issue of the generalization of the results of our study – Can we generalize our results to other software systems and projects? We believe that the external validity of our results is reasonably high.

The DR approach uses detailed historical records stored in source control systems. This ensures that the studied code development process is a realistic process which involves experienced developers working on large software systems over long periods of time. Alternatively, we could have performed controlled experiments which would run for limited time. We would not be able to confidently simulate realistic change patterns. In that case we would not have individuals with such experience and knowledge performing simulated modifications to the source code.

Concerning the results in our case study, we used several types of software systems. Nevertheless, they are all open source infrastructure software systems with no graphical user interface. Other systems such as those with graphical interface and which may implement business logic such as banking and online purchasing systems may produce different results. Also commercial software systems may exhibit different characteristics than open source systems. Fortunately, the DR approach permits us

to easily assess the benefits of adopting a tool to specific project. Instead of depending on prior results, the DR approach can be used to quickly measure the effectiveness of a variety of tools to the particular project at hand as long as the project uses a source control system to store changes to its code as it evolves.

6.4.1.4 Reliability

Reliability refers to the degree to which someone analyzing the data would reach the same conclusions/results. We believe that the reliability of our study is very high. The data used in our study is derived from source control systems. Such systems are used by most large software systems which makes it possible for others to easily run the same experiments on other data sets to reproduce our findings and tailor them to their specific project as well.

6.4.1.5 Summary of Limitations

Although the DR approach has its limitations, we believe it can greatly assist in assessing the effectiveness of software development tools. The DR approach gives us "*back-of-the-envelope*" calculations. We do not advocate fully depending on the results reported by the DR approach, instead the results should be used to evaluate the worthiness of performing more costly analysis such as pilot studies over extended periods of time to gain more concrete and validated results.

The DR approach gives us a *gut feeling* about the possible benefits and shortcoming of tools. The approach also gives us the flexibility of experimenting with a large number of alternative tool designs and ideas with no associated costs or risks as we are using historical data that has already been collected for other purposes. Moreover, the DR approach derives results that are project specific as it performs its specialized analysis using data from the project for which the tool adoption is being examined instead of relying on results reported for other software projects.

6.5 Case Study

In this section, we show how the DR approach can be used to determine the benefits of adopting different change propagation tools.

In [HH04c], we presented a study of the change propagation process in large software systems. The study focused on studying which of the information sources (presented in Section 6.2.1) are good indicators of change propagation. The study used five open software systems. The studied systems have been developed for the last 10 years and in total have over 40 years of historical modification records stored in their source control system. Table 8.1 lists the type of the software system, the date of initial modification processed in the source control data, and the programming language used. We chose to study systems with a variety of development processes, features, project goals, personnel, and domain of the studied software systems to help ensure the generality of our results and their applicability to different software systems.

Application Name	Application Type	Start Date	Files	Prog. Lang.
NetBSD	OS	March 1993	15,986	C
FreeBSD	OS	June 1993	5,914	C
OpenBSD	OS	Oct 1995	7,509	C
Postgres	DBMS	July 1996	1,657	C
GCC	C/C++ Compiler	May 1991	1,550	C

Table 6.2: Characteristics of the Studied Systems

The main results of the study are:

- Developer information in the studied software systems is not a good indicator of change propagation. The concept of code ownership is not strictly adhered to in these systems [BH99].
- Code structure dependency relations, such as Call, Use, and Define (CUD), are not a good indicators of change propagation in compar-

ison to historical co-change or code layout (same file) information. On average only 42% of entities to which a change should propagate are due to CUD relations.

- Code layout (same file) information is a better indicator of change propagation in comparison to the historical co-change information. Unfortunately, building a tool which uses only the code layout information is not sufficient as it will only guide developers to examine entities in the current file. Thus entities in other files to which changes have to be propagated will never be suggested using such a tool.

- Historical co-change information is the best practical indicator of change propagation. Although its performance is not as good as the code layout information, it is still capable of assisting developers in propagating changes across layout boundaries (*i.e.* changes to entities that are not in the same file).

These results have led us to investigate the benefits of adopting a tool which uses historical co-change information to assist developers performing changes to software systems.

At first, we sought to improve the precision results of historical co-change information. We investigated several techniques to reduce the set of suggested entities to ensure that only the relevant entities to a change are suggested. Some of the possible pruning techniques are:

- *Frequency* techniques return the most frequently related entities up to some threshold. For example, the distribution of change frequency seems to follow a $zipf$ distribution which indicates that a limited number of entities tend to change frequently and a large number of entities change very infrequently [Zip49].

- *Recency* techniques return entities that were related in the recent past. These techniques support the intuition that development tends

to focus on related functionality during particular time periods (*process information sources*).

- *Hybrid* techniques combine $Frequency$ and $Recency$ techniques using counting or some type of exponential decay function as done by [GKMS00, HH] to predict faults in software systems and assist managers in allocating testing resources.
- *Random* techniques randomly pick a set of entities to return up to some threshold such as a count. This technique might be used when there is no frequency or recency data to prune the results.

We investigated a family of hypothetical tools – FREQ(A) and RECN(M). Both tool families prune entities from the historical co-change information using recency and frequency techniques:

- Given a changed entity E, the FREQ(A) tools would suggest all entities that have changed with E in the past at least twice together and more that A% of the time.
- Given a changed entity E, the RECN(M) tools would suggest all entities that have changed with E in the past M months.

We used the DR approach to measure the performance of such tools. We experimented with values $A = \{40, 60, 80\}$ for the FREQ tools and $M = \{2, 4, 6\}$ for the RECN tools. The performance results for the five studied systems are summarized in Table 6.3 for FREQ tools and Table 6.4 for RECN tools. The F-measures shown in Table 6.3 and Table 6.4 show that FREQ(A) tools perform better than the RECN(M) tools.

6.5.1 Enhancing the Performance of FREQ(A) tools

We sought to improve the performance of the FREQ(A) tools by increasing the number of suggested entities that are relevant to the change while maintaining a high precision. We adopted an approach which *relax*ed the

Application	FREQ(60)		FREQ(70)		FREQ(80)	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	0.43	0.49	0.35	0.60	0.30	0.66
FreeBSD	0.40	0.49	0.32	0.60	0.27	0.66
OpenBSD	0.38	0.54	0.31	0.63	0.27	0.68
Postgres	0.43	0.45	0.34	0.56	0.29	0.63
GCC	0.41	0.51	0.32	0.61	0.27	0.68
Average	0.41	0.50	0.33	0.60	0.28	0.66
F-measure	.45		.43		.39	

Table 6.3: Performance of FREQ(A) tools for the Five Studied Software Systems

Application	RECN(2)		RECN(4)		RECN(6)	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	0.42	0.41	0.52	0.30	0.58	0.24
FreeBSD	0.43	0.39	0.53	0.27	0.61	0.21
OpenBSD	0.39	0.45	0.48	0.35	0.55	0.28
Postgres	0.42	0.33	0.54	0.22	0.62	0.17
GCC	0.31	0.49	0.42	0.37	0.49	0.30
Average	0.39	0.41	0.50	0.30	0.57	0.24
F-measure	.40		.37		.34	

Table 6.4: Performance of RECN(M) tools for the Five Studied Software Systems

FREQ(A) rule by incorporating other information source to suggest additional entities. In particular we defined two extensions to the FREQ(A) tools:

- **FREQFIL(A, B)**: For a changed entity E, FREQFIL(A, B) returns the same entities as FREQ(A). In addition, it returns all entities defined in the same file as E that have changed with E in the past at least twice together and more that (A-B)% of the time.

- **FREQCUD(A, B)**: For a changed entity E, FREQCUD(A, B) returns the same entities as FREQ(A). In addition, it returns all entities

related to E through a CUD relation that have changed with E in the past at least twice together and more that (A-B)% of the time.

We re-ran our experiments using the DR infrastructure on the five studied systems using values for $A = \{60, 70, 80\}$, and $B = \{10, 20, 30\}$. We produced nine results for each extension – eighteen results in total. The best performing (highest F-measure) FREQFIL extension was for A = 80 and B = 10, with precision = $.49$, recall = $.51$ and F-measure = $.50$. The results indicate that the FREQFIL(80,10) tool can on average suggest to a developer half of all entities to which a change must be propagated and that half of its suggestions are correct. Whereas the best performing FREQCUD extension was for A = 70 and B = 10, with precision = $.42$, recall = $.46$, and F-measure = $.44$. In addition, we find that a T-test on paired F-measure was significant at less than $2.2e^{-16}$ for all systems. Therefore, we are over 99% confident that the improvement in performance of FREQFIL over FREQCUD tool is statistically significant for all the studied systems[1].

We then analyzed the standard deviation of the F-measure to ensure that the performance results are stable over a long period of time. We found that for the F-measure variance for FREQFIL is 0.171, and for the FREQCUD it is 0.169. Therefore the performance of the FREQCUD tool may be more stable nevertheless the difference in standard deviation is not substantial. The good performance of the FREQFIL tool encourage us to adopt such a tool.

In summary, the DR approach permitted us to investigate the effectiveness of a large number (over 20) of not yet developed software tools with no cost associated with conducting long term case studies or even building such tools. The results show that a tool that uses historical co-change information combined with code layout (same file) information is

[1]The T-test is performed on the square root of the F-measure for each change set to ensure that the data has a normal distribution, a requirement for the T-test. Due to the large number of change sets used in our analysis, the normality of the data is not a major concern as the T-test is a robust test. Nevertheless we ensure the normality to guarantee the validity of our results.

likely to outperform tools that are based solely on either historical co-change information, code layout, or code structure information. Using these findings, we are more confident in building such a tool and conducting more costly long term case studies to validate the effectiveness of such a tool.

In our current analysis, we associated an equal importance to precision and recall. Examining Table 6.3 for FREQ tools and Table 6.4 for RECN tools, we note that FREQ tools always have better precision than RECN tools and that RECN tools usually have better recall than FREQ tools. If we were to consider recall twice as important as precision ($\beta = 2.0$), then the F-measures for FREQ tools would be (0.43, 0.36, and 0.32), whereas the F-measures for RECN tools would be (0.39, 0.44, and 0.45). In this case, the performance of the RECN tools is better. We would then explore improving the RECN tools instead of the FREQ tools.

6.6 Related Work

The work presented in this chapter focuses on two main research themes: change propagation and the use of historical data to assist in software development tasks. In this section, we discuss our work in light of prior results in both areas of research.

6.6.1 Change Propagation

We have presented a technique that measures the effectiveness of software development tools in assisting developers in propagating changes to other relevant entities in the source code.

Arnold and Bohner give an overview of several formal models of change propagation [AB93, BA96]. The models propose several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure to assist in code propagation. Rajlich proposes

to model the change process as a sequence of snapshots, where each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent [Raj97]. The model uses graph rewriting techniques to express its formalism. Our change propagation model (presented in Section 6.2) builds on top of the intuition and ideas proposed in these models. It simplifies the change propagation process and uses it as a benchmark to measure the effectiveness of software development tools.

Assessing the effectiveness of an approach or a tool in assisting a developer propagate changes can be done through a number of metrics. In this chapter we used precision and recall at the change set level. We also introduced the idea of using a utility function which has been traditionally used in the information retrieval community to measure the effectiveness of search techniques. At first glance our definition of precision and recall may seem the most intuitive one, nevertheless there are a number of other several possible definitions of precision and recall used by others. In particular, given a software system with N change sets and each change set having N_j changed entities, precision and recall could be defined relative to the changes performed to the software system or for each entity in the software system. For example, a technique may be able to suggest 50% of all entities that should change given any changed entity, and 50% of its suggestions are correct. Or a technique performance may be measured for each entity, for example if entity A is changed, then the technique can predict correctly 50% of all entities that should change with A and 50% of its suggestions may be correct. But for another entity B it can only predict 10% of the entities that should change and 100% of its suggestions are correct. Both metrics measure the performance of a technique/tool, the first metric gives a measure that is based on the change (*change based metric*), whereas the second metric gives a metric that is based on each entity (class/function) in a software system (*entity based metric*). We chose to use change based metrics that focus on the overall performance for all change sets. For example, a tool may perform well for a particular change to a specific class but it may perform badly

in assisting in propagation changes for other classes which occur in the same change set. Therefore, if a change set contains one changed entity for which a tool has good performance and many entities for a which a tool has bad performance then the overall performance of the tool for that change would be average. An entity based metric may report excellent results for a few entities. Such measurements may give an incorrect impression of the quality of a tool given that these few entities with good performance may rarely change instead most of the changes could be done to entities with bad performance.

For change based metrics, we can define precision and recall at different level of details:

- At the query level: For each changed entity (e_{ji}) in a change set (O_j), the developer queries the tool for suggestions (P_{ji}), the ($Precision_{ji}$) and ($Recall_{ji}$) for the suggestions for each entity (e_{ji}) is measured against the changed entities in the change set as follows:

$$\begin{aligned} Recall_{ji} &= \frac{P_{ji} \cap O_j}{O_j} \\ Precision_{ji} &= \frac{P_{ji} \cap O_j}{P_{ji}} \end{aligned}$$

  To measure the precision and recall for a number of change sets, the average of the precision and recall is taken for all the queries, *i.e.*:

$$\begin{aligned} Average\ Recall &= \frac{1}{M} * \sum_{j=1,i=1}^{N,N_j} \Big(Recall_{ji}\Big) \\ Average\ Precision &= \frac{1}{M} * \sum_{j=1,i=1}^{N,N_j} \Big(Precision_{ji}\Big) \end{aligned}$$

  One of the limitation of this metric is that for many changed entities a tool may not be able to give a suggestion ($P_{ji} = \{\}$) implying a precision of 1 and recall of 0, therefore we may inflate the precision of a tool.

- At the change set level: This is the metric adopted by us and it focuses on measuring precision and recall at the change set level. We choose this metric as it gives us a good measure of the perceived benefit of using a tool by a developer to implement a new feature, enhance a specific feature, or fix a particular bug. In addition, we can use the results of such a metric to monitor the evolution of a software system, for example we can detect the decay of the design of a software system when the recall of a tool that is based on the CUD relations drops. This is an indication that changes to a software system are being scattered throughout the code.

- For a period: This metric combines all the query result sets and all the changed sets for a period of time and measures their overall precision and recall as follows:

$$\begin{aligned} Overall\ Recall &= \frac{\sum_{j=1}^{N}(P_{ji} \cap O_j)}{\sum_{j=1}^{N} O_j} \\ Overall\ Precision &= \frac{\sum_{j=1}^{N}(P_{ji} \cap O_j)}{\sum_{j=1}^{N} P_{ji}} \end{aligned}$$

  This metric does not have to worry about queries where a tool returns no suggestions ($i.e.\ P_{ji} = \{\}$), as it combines the results for all the queries over a period of time. This metric is similar to the query level metric but it aggregates the results using a different technique instead of using an averaging technique.

For the DR approach any of the aforementioned metrics could be used since the main purpose of the DR approach is to compare a number of tools using a common effectiveness metric.

The work in the area of change propagation closest to our work is by Briand $et\ al.$ [Lio99]. Briand $et\ al.$ study the likelihood of two classes being part of the same change due to an aggregate value that is derived from object oriented coupling measures. The results of the analysis are done at

the class level whereas we perform our analysis at the entity level (such as function), in other words the suggestions by their approach are too coarse. Second, they use an entity based metric for each class in the software system. Briand *et al.* performed their study on a single academic object oriented small software system with at most 3 developers working on the system at any time and with 6 developers working on it throughout its lifetime. Our analysis was done on a number of large open source software system with hundreds of developers working on them. Furthermore, the dependency structure of the software system studied by Briand remained constant. This fact has permitted them to calculate their cohesion metrics at the beginning of the study period and use this data in their analysis instead of re-calculating the data throughout the lifetime of a project. The DR approach permits the analysis to be performed on an up-to-date and accurate view of the software system instead of depending on stale data and assuming it does not vary considerably. Such an assumption does not hold for most large projects with an active developer and user base.

6.6.2 The use of historical data

Several researchers have proposed the use of historical data related to a software system to assist developers gain a better understanding of their software system and its evolution. Cubranic and Murphy presented a tool that uses bug reports, news articles, and mailing list postings to suggest pertinent software development artifacts [CM03]. These information sources (*i.e.* bug reports and mailing list postings) could be used as information sources by developers and tools to assist in propagating changes. Once these sources of information are integrated into the DR framework, tool adopters can empirically study the effectiveness of this data in assisting developers. Other possible sources of information are design rationale graphs such as presented in [BMS03, RM02]. Yet these later approaches require a substantial amount of human intervention to build the data required to assist developers in conducting changes.

Chen *et al.* have shown that comments associated with source code modifications provide rich indexing for source code when developers need to locate source code lines associated with a particular feature [CCW+01]. We extend their approach and map changes at the source line level to changes in the source code entities, such as functions and data structures. Furthermore, we map changes to the dependencies between the source code entities. We then provide time tagged access to this data through the DR framework. This timed data can be used by others to study the benefits of building tools that make use of such data.

Eick *et al.* presented visualization techniques to explore change data [ESEES92, SGEMGK02]. Gall *et al.* propose the use of visualization techniques to show the historical logical coupling between entities in the source code [GHJ98].

Three works are most similar to our motivation to evaluate the effectiveness of software development tools. These are work by Zimmermann *et al.* [ZWDZ04], work by Shirabad [Shi03] and work by Ying [Yin03]. These works along with our work focus on studying the effectiveness of tools and heuristics in supporting developers during the change propagation process. Work by Zimmermann and Ying uses prior co-changes to suggest other items to change; the suggestions are evaluated against actual co-changes. They do not study the relative performance against other tools (heuristics). In particular, they do not examine other sources of information such as process or entity information. Their results show that historical information is valuable in assisting developer propagate changes throughout the source code for many large open source software systems. Work by Shirabad uses code structure and layout information as well as textual tokens derived from problem reports to suggest other items to change; the suggestions are evaluated against actual co-changes. Shirabad's results show that textual tokens derived from problem reports are more effective in suggesting entities to propagate changes to, than simply using code structure and layout information. Shirabad's results agree with our findings that show that historical co-change infor-

mation are more effective than simple static dependencies in propagating changes.

Works by Shirabad, Zimmermann and Ying perform their analysis using a batch process where the historical information stored in the source control system is divided into a training and testing periods. The training data set is used to train the system, then the effectiveness of the tool is measured using the testing data. Whereas work by us uses an adaptive approach where each suggestion done by a tool uses information from all prior changes up to the current change. We believe a batch process has its limitations as the tool's performance will not react in a timely fashion to changes and shifts in interest during the development of a software system. Zimmermann *et al.* recent work uses an adaptive process and is similar to our approach in that context.

Work by Zimmermann uses a change based period metric, whereas work by Ying uses a change based query metric. Work by Shirabad defines a relevance metric which considers a suggestion to be relevant if both entities changed together in any change set in the testing period. Furthermore, work by Zimmermann proposes a metric similar to the likelihood metric proposed by Briand. Work by Ying defines an interestingness metric that measures the value of a tool's suggestion against the current dependency structure of a software system but the suggested evaluation approach is rather manual. For example a change is not considered interesting if a tool proposes that if a *.h* file changes then its corresponding *.c* file should change as well. The DR approach permits the automation of such analysis and is able to statistically show the benefit of a tool against other types of tools.

Ying and Shirabad focus their prediction at the file level whereas Zimmermann *et al.* and our work focus our prediction at the entity level (and can easily lift such predictions to the file level).

All three approaches use a variety of data mining techniques to perform their analysis such as association rule mining for Zimmermann *et al.*, machine learning algorithms for Shirabad, and market basket analysis

for Ying. All data needed for the three approaches are easily available through the DR software infrastructure. Table 6.5 summarizes these three approaches along with our work and Briand's work.

6.7 Conclusion

Practitioners are always in search of silver bullets – software development tools or strategies – that could assist them in maintaining large software systems. The effectiveness of such silver bullets are unfortunately rarely assessed in realistic settings. Ideally, the effectiveness of these tools should be based on *(1)* using the tools by practitioners *(2)* over an extended period of time *(3)* to perform real changes *(4)* on large software systems. This is a costly approach and is usually not possible due to many process and cost reasons. We presented an approach to *automatically* measure some of the claims of software maintenance tools and strategies.

The *Development Replay (DR)* approach analyzes source control repositories to derive a detailed project state and change sets. The project state tracks the evolution of a variety of project entities such as the interactions between source code entities (*e.g.* functions and variables), or the interaction between developers and source code entities (*e.g.* File Y is always changed by developer A). The change sets represent changes to the source code to implement or enhance features, or to fix bugs.

Using this data, researchers can confidently gauge some of their claims about proposed maintenance strategies and tools as early as possible. The approach as well permits researchers to experiment with a variety of ideas and rerun their experiments. The DR approach offers researchers this flexibility while ensuring that the effectiveness of their alternate tools are measured using actual changes performed by practitioners over a long period of time working on large software systems.

We presented an example of using the DR approach to investigate tools assisting developers to propagate changes in the source code. Change

propagation is a central aspect of software development. As developers modify software entities such as functions or variables to introduce new features or fix bugs, they must ensure that other entities in the software system are updated to be consistent with these new changes. The DR approach permitted us to easily investigate the performance of a large number of tools to assist developers without performing costly long term studies.

Nevertheless, the approach has its limitations. We acknowledge that the approach is not sufficient to measure the full effectiveness of development tools. Traditional case studies are still needed. The main value of the approach is that it permits researchers to experiment ahead of time with a variety of techniques to determine the most promising tool to build and empirically study its effectiveness. This analysis is done ahead of time instead of recognizing limitations or possible improvements of a studied tool in the middle of a study.

Unless researchers gain a better understanding of the specific factors which cause tools and methods to be more or less cost-effective, the creation of new tools or technology will essentially be a random act. The Development Replay approach along with empirical studies support researchers in arriving to well-founded decisions which are more likely to be adopted by practitioners in the field.

We believe that the approach and results presented herein should encourage researchers and tool developers to search for different and more sophisticated tools with better performance. These new heuristic and ideas can be validated easily using the DR approach and data derived from the development history of large software projects.

Work by	Level of Analysis	Information Sources	Performance Metrics	Metric Type	Analysis Technique	System Types
Briand *et al.*	Class	Entity (CUD)	Likelihood	Entity	Batch	C++ (Academic)
Shirabad	File	Entity (Layout, Naming)	Relevance	Entity	Batch	Pascal-like (Commercial)
Ying	File	Entity (Co-Change)	Precision, Recall, Interestingness	Change based Query	Batch	C++, Java (Open Source)
Zimmermann *et al.*	Entity, File	Entity (Co-Change)	Precision, Recall, Likelihood	Change based Period	Batch, Adaptive	C, C++, Java Python, Latex (Open Source)
Hassan and Holt	Entity, File	Entity (all sources) Developer, Process	Precision, Recall, Std. Deviation, F-measure	Change based Change Set	Adaptive	C (Open Source)

Table 6.5: Summary of Work by Briand, Shirabad, Ying, and Zimmermann *et al.* in relation to our work.

Part III

Using Software Repositories to Support Managers

Managers of large projects need to prevent the introduction of faults, assure their quick discovery, and their immediate repair while ensuring that the software can evolve gracefully to handle new requirements by customers. Moreover, they endeavor with varying degrees of success to wisely allocate their limited testing and development resources to the most appropriate parts of the code. Unfortunately, in many cases such attempts are based on ad-hoc techniques and rough approximations. Their success depends on their intuition, experience and chance. Bug prediction and resource allocation issues become non-trivial challenges which managers must face and resolve successfully.

This part deals with both of these issues by presenting two pieces of research work:

- **The Top Ten List**: We propose (*The Top Ten List*) which highlights to managers the ten most susceptible subsystems to have a fault. The list is updated dynamically as the development of a software system progresses. Managers can use the Top Ten list to optimize the usage of their limited resources to deliver quality products on time and within budget by focusing testing resources to the subsystems suggested by the list.

- **The Development Process Chaos**: We define the modifications done to the source code as the *Development Process Chaos*. We propose a complexity metric that is based on the development process followed by practitioners to produce the code instead of on the code or the requirements. Through a case study using six large open source software systems, we show that the number of prior faults is a better predictor of future faults than the number of prior modifications. Also the case study indicates that fault predictors based on our development process chaos models are better predictors of faults in large software systems when compared against predictors based on prior modifications or prior faults.

This part is likely to be of interest to managers of large software systems. This part shows that historical changes stored in source control repositories could be used to predict bugs and assist managers in allocating resources. The historical records for several open source projects are used to verify the benefits of our proposed research ideas.

CHAPTER 7

The Top Ten List: Dynamic Fault Prediction

To remain competitive in the fast paced world of software development, managers must optimize the usage of their limited resources to deliver quality products on time and within budget. We present an approach (The Top Ten List) which highlights to managers the ten most susceptible subsystems to have a fault. The list is updated dynamically as the development of the system progresses. Manager can focus testing resources to the subsystems suggested by the list.

We present heuristics to create the Top Ten List and develop techniques to measure the performance of these heuristics. To validate our work, we apply our presented approach to six large open source projects (three operating systems: NetBSD, FreeBSD, OpenBSD; a window manager: KDE; an office productivity suite: KOffice; and a database management system: Postgres). Furthermore, we examine the benefits of increasing the size of the Top Ten list and study its performance.

7.1 Introduction

MANAGERS of large projects need to ensure that the project is delivered within budget with minimal schedule slippage. They have to prevent the introduction of faults, ensure their quick discovery and immediate repair, and make sure the software can evolve gracefully to handle new customer demands. Unfortunately, all of these demands need to be done with restricted personnel resources within limited time. Resource allocation becomes a non-trivial challenge which managers must face and resolve successfully.

Managers would like to optimize their resources usage. They would like to allocate resources to areas that are in need of these resources and reassign them as soon as interest and focus shifts. In this chapter, we focus on the challenges surrounding fault detection and repair in large software systems. We would like to give managers a *Top Ten List* of subsystems that are most susceptible to faults. We need the list to be updated dynamically to reflect future risks based on the current status of the project. By limiting the number of files in the list, we hope to give managers an easy and clear way to allocate their limited resources. By updating the list as the software system evolves and the risks associated with components change, we hope to give mangers a dynamic tool which is always able to give informed and up-to-date warnings. Finally, we would like to build a tool that is not intrusive and requires as little details and setup as possible to permit managers to get a high return on their investment.

Previous research in software faults has focused on two areas:

1. **Count** based techniques which focus on predicting the number of faults in subsystems of a software system. Managers can use these predictions to determine if the software is ready for release or it has many lurking bugs. They can use the predictions to guide their resource allocations as they wind up the project towards release.

These models are validated by dividing the data into equal size periods and predicting the faults in one period using data from all the previous period. For example,the fault data from one release can be used to predict faults in following releases. Examples of such work are [GKMS00, OA96, Sch99].

2. **Classification** based techniques which focus on predicting which subsystems in a software system are fault-prone. Fault-prone is defined by the manager, for example a fault prone subsystem may be any subsystem with more than two faults in a release. These predictions can be used to assist managers in focusing their resources allocation in a release, by allocating more testing resources and attention to fault-prone subsystems. Again these models are validated by testing if the data from one release can be used to predict if a subsystem is fault prone in following releases. An example of such work is [MK92].

These approach focus on long term planning. They are designed for long term prediction and are validated by using data from one software release to predict values in following releases by building some types of statistical models. In this chapter we focus on short term dynamic prediction. We present an approach to validate short term predictions and we show an analysis of this framework using several heuristics for fault predictions. This focus on short term planning would permit managers to monitor more closely the development and testing processes instead of only depending on long term planning which tends to be harder to react to varying competitive market pressures over the lifetime of a software system.

We focus on predicting the subsystem that are most likely to have a fault in them in the near future, in contrast to count based techniques which focus on predicting an absolute count of faults in a system over time, or classification based techniques which focus on predicting if a subsystem is fault prone or not. For example, even though a subsystem may not be fault prone and may only have a few number of predicted faults,

it may be the case that a fault will be discovered in the next few days or weeks. Or in another case, even though a fault counting based technique may predict that a subsystem has a large number of faults, they may be dormant faults that are not likely to cause concerns in the near future. If we were to draw an analogy to our work and rain prediction, our prediction model focuses on predicting the areas that are most likely to rain in the next few days. The predicted rain areas may be areas that are known to be dry areas ($i.e.$ not fault prone) and may be areas which aren't known to have large precipitation values ($i.e$ low predicted faults).

The prediction are presented to managers as a list of the Top Ten most likely subsystems to have faults. That list is modified over time as new files are modified or as new faults are discovered and fixed. To validate the quality of our predictions, we borrow concepts from the vast literature of caching – file system and web proxy caching. In particular, we use the idea of *Hit Rate* which is traditionally used to determine the performance of caching systems. A high Hit Rate indicates that the Top Ten list is performing well and fault that were discovered recently had been already present in the list. Moreover, we present a new metrics – *Average Prediction Age* – to measure the practical benefits of predictions in the Top Ten list. A prediction that warns of a fault occurring within a couple of hours is not as valuable as a prediction that warns of a fault a couple of weeks before its occurrence.

7.1.1 Organization of Chapter

This chapter is organized as follows. Section 7.2 introduces the motivation behind our work and explains the concepts of *Hit Rate* and *Average Prediction Age*. We use both concepts to evaluate and compare different fault prediction heuristics presented in this chapter. In Section 7.3, we present several heuristics to build the Top Ten List based on various characteristics. Then in Section 7.4, we present the six open source systems used in our case study and the data used in our analysis. In Section 7.5, we measure the performance of the proposed heuristics by ana-

lyzing the development history of the studied software systems using the *Hit Rate* and *Average Prediction Age* concepts introduced earlier. Later in Section 7.6 we analyze the performance benefits of increasing the size of the proposed Top Ten list. In Section 7.7, we discuss our results and address shortcomings and challenges we uncovered in our approach. Section 7.8 showcases related work in the fault prediction literature. Finally, section 7.9 summarizes our results.

7.2 Motivation

To cope with a large number of tasks at hand, managers are always in search of a silver bullet that would give them a list of issues to focus their limited resources on. Hence, the idea of the Top Ten list. The Top Ten list is a list of the top ten subsystems which are most susceptible to have a fault appear in them in the near future. Managers can use this list to focus their limited resources and maximize their resource usage.

The inspiration of the idea of Top Ten list comes from the idea of a resource cache. Previously, caching has been proposed to solve many problems associated with limited resources and latency associated with acquiring them. In the file system domain, caching is used to store previously used files in memory so future requests to these files would be fulfilled from memory instead of accessing the hard drive which is much slower than memory. The same ideas and concepts have been applied to database and web systems.

Conceptually, a cache is used to store a limited number of resources for cheap access. Heuristics employed by the cache system determine which resources to store, usually based on the probability that the resource will be accessed in the near future. For example, in a file system cache it is expected that a file that was accessed recently will be accessed again within the next few minutes. By storing this file in the cache, consecutive accesses will be much faster as they won't require slow disk access. Unfortunately, a cache is usually a limited resource. For example memory

is much smaller than hard disk, or a web proxy server is much smaller than the whole Internet. Thus cache replacement heuristics are used to decide which resources should stay in the cache and which ones need to be evicted to store new cacheable resources.

We believe the same idea can be adopted for deciding which subsystems are most susceptible to having a fault in the near future. A manager of a project can only focus on a limited number of resources. These limited resources can be thought off as the cache system size. Cache heuristics can be developed to determine which subsystems are no longer susceptible to a fault and which are still susceptible to a fault. For example, research has shown that previous faults in a subsystem are good indicator of future faults [FN99]. One heuristic would build the Top Ten list based on the number of previously discovered faults in a subsystem. Thus the Top Ten list would contain the ten most faulty subsystems. Other heuristics based on the number of developers that worked on the subsystem, the recency of the latest fault or modification, the size of the subsystem, the number of modifications, or a metric that is based on fusion of a subset of these ideas are a few of the possible heuristics.

The huge literature in fault analysis and prediction can be used to develop such heuristics and many of previous fault prediction findings can be validated using our presented approach. Fenton and Neil organize defect prediction models based on the source of the data used for the prediction into three main areas [FN99]:

1. Models based on size and complexity metrics
2. Models based on testing metrics
3. Models based on process quality metrics.

In addition, work by Graves *et al.* [GKMS00] and Khoshgoftaar *et al.* [KA98] suggest using code change metrics such as code churn [EGK+01] to build quality prediction models. Any of these aforementioned model data can be used to build the Top Ten list. In particular, models based on

size and complexity and models based on the code change process are the most promising ones to build the Top Ten list, as the value of their metrics tends to change over the short term as source code is modified to enhance the software system. In contrast models based on process quality metrics such as CMM ratings tend to be more stable and would be more useful for long term predictions.

By basing the idea of Top Ten list on caching systems, we can borrow many of the well developed concepts used to study the performance of caching systems in our analysis. In particular, the concept of *Hit Rate (HR)*. Hit Rate is the most popular measure of the performance of a caching system. It is the number of times a referenced resource is in the cache. For example a Hit Rate of 60% indicates that six out of every ten requests to the cache found the resource being requested in the cache. For the analysis of the Top Ten list this would mean that six out of the ten subsystems that were in the Top Ten list had faults in them as predicted by the heuristic used to build the list. Thus, the higher the Hit Rate the better the prediction power of the heuristic and the usefulness of the Top Ten list, as managers aren't wasting resources on subsystems that are not susceptible to faults while missing other subsystems that are susceptible.

Unfortunately, using Hit Rate is not sufficient to measure the practical efficiency of the Top Ten list algorithms. Hit Rate only tells us if a subsystem that had a fault was in the Top Ten list or not. We hope to give managers enough advance warning time to react to the fault prediction. For example, if we have a 90% Hit Rate yet the subsystems that have faults are put in the Top Ten list just seconds or minutes before the fault is discovered in them then such predictions although from a theoretical stand point are valid they are not practically useful. We would like to have a measure that is more practical, as managers require enough time to react to the proposed predictions. Hence, the time of adding a subsystem to the Top Ten list is important to obtain a more accurate measure of the performance of the Top Ten list. In contrast for web or file systems, the time of entry of a resource in the cache does not matter as long as

the resource was found in the cache when requested. To overcome this limitation of the Hit Rate, we adopted two new metrics:

1. *Adjusted Hit Rate (AHR)*: The adjusted Hit Rate is a modified Hit Rate calculation which counts a hit only if the subsystem had been in the cache/Top Ten list for over 24 hours (other time limits are possible). For example we do not count a hit if the subsystem has been in the Top Ten list for just a couple of minutes. This will prevent us from over inflating the performance of the heuristics used to build the list. In the rest of this chapter we use the term Hit Rate to refer to AHR, unless otherwise noted.

2. *Average Prediction Age (APA)*: The Average Prediction Age calculates on average for each hit how long a subsystem has been in the cache/Top Ten list. Although HR has been adjusted to account for prediction with a very short warning, we measure the APA to get a better idea of the age of the prediction. For example, two heuristic may have similar HR but one heuristic predicts on average faults a week a head of time whereas the other predicts them a full month a head of time. A longer APA indicates a better performing heuristic for building the Top Ten list.

Using the HR and APA metrics, we proceed to evaluate various heuristics proposed in the following section.

7.3 Heuristics For The Top Ten List

As noted earlier previous findings and observations from published literature in fault prediction can be used as heuristics to build the Top Ten list. For the purpose of this chapter, we chose to use the following heuristics for their simplicity and intuitiveness. They are by no mean a full listing of all possible heuristics instead they are some examples to validate our proposed Top Ten list approach:

7.3.1 Most Frequently Modified (MFM)

The Top Ten list contains the subsystems that were modified the most since the start of the project. The intuition behind this heuristic is that subsystems that are modified frequently tend over time to become disorganized. Also, many of the assumptions that were valid at one time have the tendency to no longer be valid as more features and modifications are performed on these subsystems. Eick *et al.* studied the concept of code decay and used the modification history to predict the incidence of faults [EGK+01, ELL+92]. Graves *et al.* showed that the number of modifications to a file is a good predictor of the fault potential of the file [GKMS00]. In other words, the more a subsystem is changed the higher the probability that it will contain faults.

This heuristic will tend to have a high APA as frequently modified subsystems will remain in the Toplist for a long time. This may degrade the HR of this heuristic as it won't adapt to changes in the modification of files. For example, if in one release of an operating system all the work has concentrated on improving the memory manager and in the following release all the work focused on improving the file system, then the MFM heuristic will still be affected by the modification counts of the previous release and will give out bad predictions. This limitation is a concern for any frequency based approach and is commonly refereed to in the literature of caching as the *cache pollution problem* [CMCmWH91]. To overcome this problem, heuristics that update the list based on a combination of the frequency and recency of a modification could be used.

7.3.2 Most Recently Modified (MRM)

The Top Ten list contains the subsystems that were recently modified. In contrast to the Top Ten list built using the MFM heuristic, the MRM Top Ten list is changing at a much higher rate as new files are modified continuously and are inserted in the Top Ten list. The intuition behind this heuristic is that subsystems that are modified recently are the ones

most likely to have a fault in them. Finding faults in subsystems that were not modified for a long time is highly unlikely. In [GKMS00], Graves *et al.* showed that more recent changes contribute more to fault potential than older changes over time.

7.3.3 Most Frequently Fixed (MFF)

The Top Ten list contains the subsystems that have had the most faults in them since the beginning of the project. The intuition behind this heuristic is that subsystems that have had faults in them in the past will always tend to have faults in them in the future. Again this heuristic, like MFM suffers from the *cache pollution problem*.

7.3.4 Most Recently Fixed (MRF)

The Top Ten list contains the subsystems that had faults in them recently. The intuition behind this heuristic is that subsystems that had faults in them recently will tend to have more faults showing up in the future till most of the faults are found and fixed. In contrast, a Top Ten list built using the MFF will be a lot more stable than a list built using the MRF, as the subsystems in the list won't be changed as often.

The aforementioned heuristics represent a small sample of a huge variety of heuristics that can be used to build a Top Ten list. Conceptually, each heuristic can depend on one or a combination of the following characteristics of a software system.

1. *Recency*: The recency of modifications or fault fixes applied to the source code, such as MRM and MRF.
2. *Frequency*: The frequency of modifications or fault fixes applied to the source code, such as MFM and MFF.
3. *Size*: The size of subsystems, the size of modifications.

4. *Code Metrics*: The fault density, the cyclomatic complexity [McC76], or simply the LOC.

5. *Co-Modification*: Subsystems modified together will tend to have faults during similar times, for example.

We note that the problem of fault prediction has some characteristics that are different from classical caching literature, in particular:

- Whereas for file and web systems the number of possible resources to be cached is rather large, the number of subsystems that are analyzed for inclusion in the Top Ten list is limited, as managers have a limited number of resources to allocate to investigate the suggestions of the Top Ten list.

- Furthermore, CPU usage, algorithm complexity, and responsiveness of the caching heuristics are not a major issue due to the small number of subsystems that need to be analyzed. Also we expect the Top Ten list to be generated daily thus more complex and elaborate algorithms could be used to build the list overnight, if needed. This is not possible in web and file system caching where the user expects an immediate and quick response.

- Finally, as pointed out earlier, a simple HR metric is not sufficient to measure the practical benefits of a heuristic, as managers require enough advance warning time to react to suggestions.

7.4 Studied Systems

To study the benefits of using the Top Ten list in the development of large software systems, we evaluated our proposed approach using six large open source software systems. Table 8.1 summarizes the details for these software systems. The oldest system is over ten years old and the youngest system is five years old. For each system, we list the number

of subsystems it has and the number of faults that were discovered in it according to our fault discovery process described below. For example, the Postgres database systems contains 104 subsystems and over its lifetime has had 1401 faults. Furthermore, it is written in C.

To measure the performance of the Top Ten list, we used the development history of these six software systems. The development history is stored in a source control system, such as CVS [CVSa, Fog99] or Perforce [Per]. The source control system stores all modifications that occur to each subsystem in the software system as it evolves. Each modification records the changed lines in the subsystem, the reason for the change, and the exact date of the change. Using a lexical technique, similar to [MV00], we automatically classify modifications into three types based on the content of the detailed message attached to a modification:

Fault Repairing modifications (FR): These are all modifications which contain terms such as *bug*, *fix*, or *repair* in the detailed message attached to the modification. The Top list attempts to predict ahead of time which subsystems are most susceptible to have such a modification applied to them in the near future.

General Maintenance modifications (GM): These are modifications that are mainly bookkeeping ones and do not reflect the implementation of a particular feature. These modifications are removed from our analysis and are never considered. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well.

Feature Introduction modifications (FI): These are modifications that are not FR or GM modifications.

The detailed description of the history of code development provides a rich opportunity to replay the history of the development of a software

system and measure the benefits that the developers would have got if ideas such as the Top Ten list were accessible to them.

Application Name	Application Type	Start Date	Subsys. Count	Faults	Prog. Lang.
NetBSD	OS	21 March 1993	393	2451	C
FreeBSD	OS	12 June 1993	182	3264	C
OpenBSD	OS	18 Oct 1995	401	1015	C
Postgres	DBMS	9 July 1996	104	1401	C
KDE	Windowing System	13 April 1997	167	6665	C++
Koffice	Productivity Suite	18 April 1998	259	5223	C++

Table 7.1: Summary of the Studied Systems

We believe that the variety of development processes used, implementation programming languages, features, domain of the studied software systems ensures the generality of our results and their applicability to different software systems. In the following section, we present the performance of the heuristics presented in Section 7.3 against each of the studied software systems.

7.5 Measuring The Performance Of The Top Ten List

In this section, we measure the performance of the heuristics proposed in Section 7.3, to build the Top Ten list. For each of the software systems we analyzed the source control repository automatically without any user intervention. We chose to ignore the first year in the source control repository, due to the special startup nature of code development during that year as each project initializes its development process and the corresponding effect on its source code repository. We then used the following three years to measure the performance. For each heuristic, we plot the

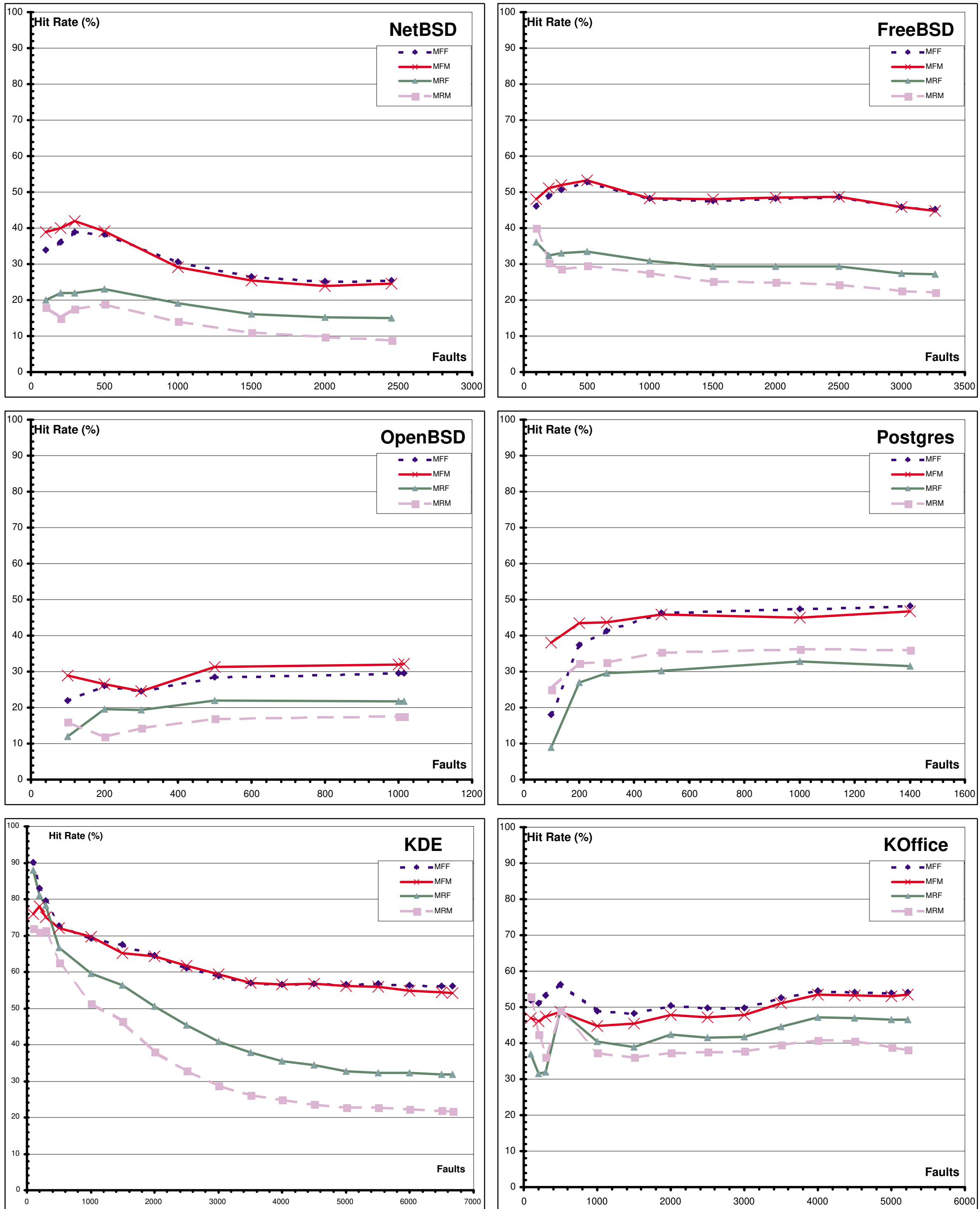

Figure 7.1: Hit Rate For The 4 Proposed Heuristics

Hit Rate (HR) versus the fixed faults over the three year period. Furthermore, we calculate the total Hit Rate and the Average Prediction Age (APA) over the studied three years for each of the six open source systems we studied.

Figure 7.1 shows the performance for the four proposed heuristics. In the figure we show the *Hit Rate* of the Top Ten list using each heuristic for each fault that occurs. For example for *NetBSD* once there are 1000 faults, the Hit Rate for the heuristics are as follow: MFF (29%) MFM (30%) MRF (20%) and MRM (15%). We note that we do not show the Hit Rate for the first 100 faults, as we choose to use the first 100 faults to calibrate our Top Ten list with some historical data to gain a more realistic and fair comparison of the different heuristics as the Top Ten list fills up slowly over time.

Examining the figure, we note that the two heuristics (MFM and MFF) that are based on a count of modifications or faults have the best performance. In contrast, the other two heuristics (MRM and MRF) which are are based on the recency of modifications and detection of faults in a subsystem have a much worse performance.

Furthermore, the performance of MFF at the beginning is always worse than the performance of MFM, this is due to the fact that at the beginning there are not as many faults thus the MFF heuristic performance is negatively affected. The need of MFF for a large number faults to calibrate itself suggests the need for a heuristic based on the modifications count at the beginning of the development of the project. Later on we may switch to a heuristic that is based on the fault counts if it is performing better. In our analysis, we see that around 400 - 500 faults, the MFF has enough faults to calibrate well.

Over time, the performance of the proposed heuristics either decline or stay constant except for the *Koffice* system where it improves. The decline in the prediction quality may suggest that the Top Ten list has been polluted by subsystems that were very highly modified/fixed in the past but are no longer being modified in the later years. An enhanced

heuristic that overcomes this problem may be very beneficial in improving the performance of the list.

Table 7.2 summarizes the performance metrics over the three years of data used in the study. In particular, we notice that the unadjusted Hit Rate for the recency based heuristics such as MRM and MRF drops significantly once the Adjusted Hit Rate is calculated. By examining the Average Prediction Age we see that it is less than a day in many of the cases where the recency based heuristic is used.

Application	Heuristic	HR (%)	AHR (%)	APA (in days)
NetBSD	MRM	22.4	9	0.3
	MRF	20.6	15	0.8
	MFM	24.4	24.4	133.8
	MFF	25.3	25.3	138.7
FreeBSD	MRM	32.6	22.2	0.98
	MRF	32.6	27.2	1.7
	MFM	44.9	44.9	252.7
	MFF	45.1	45.1	245.1
OpenBSD	MRM	28.5	17.6	0.71
	MRF	24.5	21.8	3.11
	MFM	32.1	32.1	182.22
	MFF	28.8	28.8	168.5
Postgres	MRM	42.1	36.2	3.3
	MRF	35.4	31.4	4.4
	MFM	48.4	48.4	287.8
	MFF	46.6	46.6	288.6
KDE	MRM	46.6	21.7	1.4
	MRF	49.3	31.7	3.9
	MFM	54.3	54.3	375.4
	MFF	56.1	56.1	394.1
Koffice	MRM	53.6	38.3	2.4
	MRF	56	46.6	4.6
	MFM	53.4	53.4	133.8
	MFF	54.1	54.1	341.3

Table 7.2: HR, AHR, and APA for the Studied Systems During the 3 Years

7.6 The Effects Of a Larger List

In the previous section, we presented the performance of the Top Ten list approach using various heuristics. In this section, we examine if increasing the size of the list would improve the performance of the heuristics. We focus on only two of the four proposed heuristics, we chose MFM to represent the frequency based heuristics as its performance is very similar to MFF and we chose MRM to represent the recency based heuristics as its performance is similar to MRF.

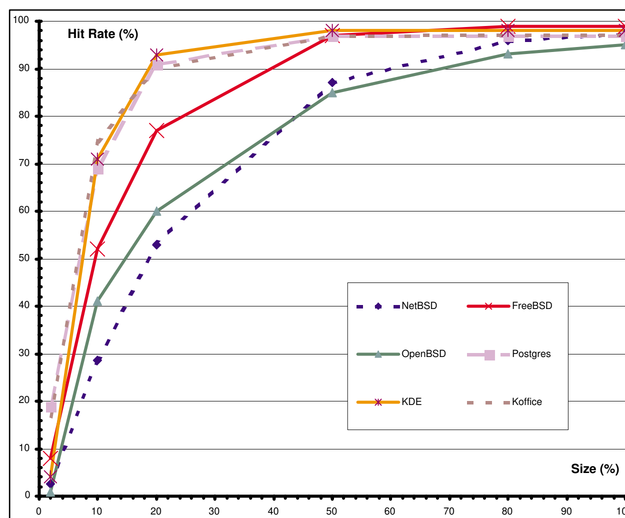

Figure 7.2: Hit Rate Growth As a Function of The Top List Size Using MRM Heuristic

For both MFM and MFF, we re-ran the same experiments done in the previous section while varying the size of the Top Ten list. We chose to make the size of the list a function of the number of subsystems in the software system. Thus we chose to have the size of the list vary between 2%, 10%, 20%, 50%, 80%, and 100% of the number of subsystems. In the case of 100%, we are able to see the best possible HR but unfortunately this is not practical as managers would have to focus their attention to all the subsystems in the software system which defeats the purpose of a

Top list.

Figures 7.2 and 7.3 show the growth of the Hit Rate as we vary the size of the Top list. We notice that when the Top list size is under 50% of subsystems in the software system then MFM (frequency based heuristic) outperforms the MRM (recency based heuristic). Once we are above 50% both types of heuristics have the same performance. Also we can never reach a Hit Rate of 100% as we always have misses in our predictions as we populate the list initially. For example, for the MFF heuristic a subsystem would have to have at least one fault that was not predicted at the beginning to be considered for inclusion in the predicted Top list.

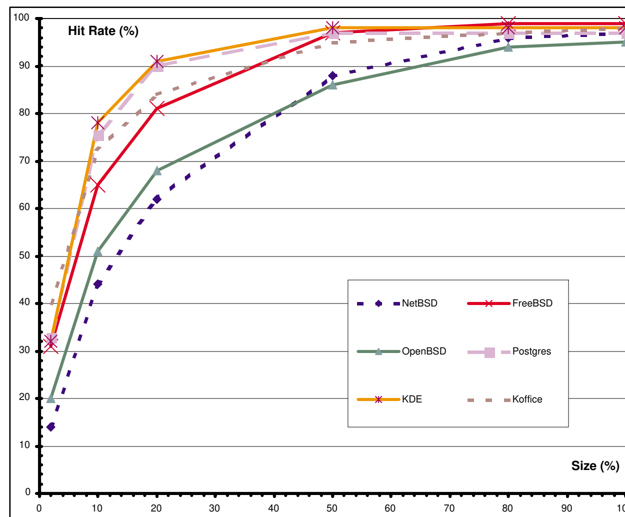

Figure 7.3: Hit Rate Growth As a Function of The Top List Size Using MFM Heuristic

Examining the growth of the Hit Rate in Figures 7.2 and 7.3, we notice that the Hit Rate exhibits a logarithm growth as we increase the size of the Top list. This indicates that the benefit of increasing the size of the Top list diminishes exponentially. From both figures, we see that a Top list which is around 20% the number of subsystems in the software achieves the best return on investment for managers. The 20% value supports previous findings by Munson and Khoshgoftaar [MK92] and by Adams

[Ada84] which showed that faults tend to occur in a subset of the subsystems of large software systems. It also showcases the value of using the Top Ten list as managers will expend less effort on fault free subsystems and can allocate more resources to troublesome ones.

7.7 Discussion

In this section, we elaborate on issues relating to the performance of the heuristics used to build the Top list.

7.7.1 An Accurate Measure of the Performance of a Heuristic

The Top Ten list assists managers in allocating testing resources by focusing on the subsystems that are likely to have a fault appear in them in the near future. In our analysis, we used the change history to measure the performance of heuristics. In particular, we used the fact that a fix was applied to a subsystem as an indication that a fault was detected in that subsystem. A record of all reported bugs throughout the lifetime of a software system could have been used to evaluate the performance of the different heuristics. Unfortunately, for most open source systems detailed bug tracking systems do not exist. Furthermore, reported bugs are not an indication of the occurrence of faults or their severity, since the reported bugs may be due to misunderstandings of the functionality of the software system by the reporter of the bug. We believe that the use of fixes instead of bug reports is a reasonable measure since bug reports do not exist for the systems we studied, and managers tend to focus their resources on the most critical faults.

7.7.2 Performance of Fault Based Heuristics

In our analysis we used two heuristics (MFF and MRF) that are based on fault counts. Unfortunately even though these two heuristics have good

performance as presented in the previous section, it may be challenging to measure their performance if a Top list did actually exist for the development team. The Top list biases the effort and work performed by a development team. There is a high tendency for developers to focus their testing resources to subsystems that are in the Top list. Thus over time, the fault discovery may be influenced by the Top list and using the fault counts becomes an inaccurate measure. Instead using heuristics based on the modification counts are likely to be more stable and un-affected by the Top list suggestions. This poses an interesting challenge for software engineering research where introducing new techniques to a process may invalidate the validation of benefits of the new techniques. Thus, even though historical data show the benefits of a research idea, validating the idea in a practical setting may reveal interesting challenges and issues.

7.7.3 Determining a Practical Average Prediction Age

Throughout the chapter we emphasized the need for heuristics that are able to provide high HR. To ensure that our results are useful and practical we measured the Prediction Age (PA) for each hit and chose not to count hits with low PA. As a manager is not given enough warning to react when the PA is low. We then chose to measure the APA which is the sum of the PA's for all the Hits divided by the number of hits. Looking at Table 7.2, we list the APA for all heuristics for each of the studied software systems. As pointed out earlier, recency based heuristics have a rather low APA. Unfortunately, frequency based heuristics have a high APA. This is mainly due to the *cache pollution problem*. The need for a heuristic that can combine a low APA with a high HR is justified. It would be very useful and practical for managers to get advance warnings that are not too early and are not too late. We now briefly discuss and present some measurements for such a heuristic.

Based on the results shown in Table 7.2, we would like a heuristic which keeps track of the recency and measures the frequency of events as well. We propose the use of an exponential decay function to build our

heuristic. The decay function would reduce exponentially the effect of a modification or a fault on the probability that a fault will be discovered, based on how long ago a fault/modification to the subsystem has occurred. Then to measure the frequency, instead of adding up the number of times a modification/fault occurred, we add up the exponentially decayed values. Consequently, given two subsystems who both have had 3 modifications to them, the subsystem with the 3 more recent modifications will have a higher heuristic value and would be considered more likely to have a fault discovered in the near future. More formally, we define a heuristic function (HF) and the Top list is created by choosing subsystems with the highest HF value. The HF for a modification based heuristic is defined as:

$$HF(S) = \sum_{m \in M(S)} e^{T_m - Current\ Time}$$

where $M(S)$ is the set of modifications to a subsystem S and T_m is the time of modification m.

Application	AHR (%)	APA (in days)
NetBSD	25.3	26.1
FreeBSD	42	129
OpenBSD	33.1	38.6
Postgres	49	33.8

Table 7.3: AHR and APA for the Exponential Decay Heuristic

We reran our results on four of the software systems in our system. Table 7.3 shows the performance results for using an exponential decay heuristic. We note that the APA values are much more moderate compared to the corresponding values shown in Table 7.2. The APA suggests that the new heuristic provides enough early warning and is still capable of dynamically updating as the development in the project changes over time.

7.8 Related Work

The work most closely related to the work presented in this chapter is done by Khoshgoftaar *et al*. In [KA98], they present a technique to predict the order of the subsystems that are most likely to have a large number of faults. The main similarity between our work is the recognition that managers have a limited number of resources and need to focus their resources on a selected few subsystems in a large software project. Whereas, Khoshgoftaar orders subsystems based on their degree of fault proneness, we order subsystems based on their likelihood of containing a fault in the near future. Thus, our technique may choose to rank highly subsystems that may not be considered fault prone, yet they may have just a few faults appearing very soon in them.

7.9 Conclusion

We presented a new approach to assist managers in determining which subsystems to focus their limited resources on. By using this approach managers should be able to allocate testing resources wisely, locate faults in a timely manner and fix them as soon as possible. The approach uses ideas that have been extensively researched in the literature of web and file systems. The idea of caching as a limited resource is extended to the idea of limited testing resources. We show that the problem of determining which entities to cache is similar to the problem of determining which subsystems to focus testing resources on. We present the concept of Hit Rate which is widely used to measure the performance of various caching heuristics. Then we extend it to measure the performance of our heuristics that are used to build the Top Ten list.

We studied our proposed approach and heuristics using the development history of six large open source project. We saw that we can achieve a Hit Rate that is higher than 60% for some of the systems. We then examined the possibility of increasing the size of the Top Ten list and

noticed that a list that contains 20% to 30% of the subsystems in a software system provides very good results even when using rather simple heuristics. We then presented a more elaborate heuristic based on an exponential decay function. We showed that the results using the new heuristics combine the benefits of early warnings for faults and the ability to dynamically adjust as new development data is available.

We believe that the Top list approach holds a lot of promise and value for software practitioners, it provides a simple and accurate technique to assist them in allocating resources as they maintain large evolving software systems.

CHAPTER 8

Code Development Chaos: a New Perspective on Software Complexity

We offer a novel view on the problem of complexity in software. We propose a complexity metric that is based on the process followed by software developers to produce the code instead of on the code or the requirements.

We conjecture that a chaotic or complex development process negatively affects its outcome, the software system. We validate our hypothesis empirically through a case study using data derived from the development process history of six large open source projects (three operating systems: NetBSD, FreeBSD, OpenBSD; a window manager: KDE; an office productivity suite: KOffice; and a database management system: Postgres).

8.1 Introduction

COMPLEXITY must be monitored and controlled at all times to ensure the successful evolution of a software system. The natural conviction is that unnecessary or excessive complexity has many negative effects on a project. Yet, complexity is needed to introduce new features and satisfy more demanding customers otherwise the software system will be abandoned [LRW+97]. Managing the complexity of the software system becomes a paramount goal while striving to meet users' needs. Brooks' words, describing software development, mirrors this sentiment well:

> *"Complexity is the business we are in and complexity is what limits us."* Fred Brooks, The Mythical Man-Month [Bro74].

The literature on software metrics contains a wealth of studies that measure the complexity of source code. These measures are correlated to faults or associated with difficulties in understanding and maintaining software systems. For example, Halsted metrics measure the lexical and textual complexity of the source code [Hal77]. McCabe metrics focus on measuring the logic flow and structural complexity of the source code. A source code with complex logic is expected to be difficult to understand and maintain. This complexity may eventually lead to the introduction of bugs in the software system and is likely to cause the dissatisfaction of its users.

Complexity lurks throughout a software project and does not solely reside in the source code. Other facets of a software project such as its design, the customer's requirements, the team structure and size, the development process, market pressure, and problem domain are all susceptible to complexity as well. Moreover, all these facets interact in a feedback loop – *an increase in complexity in one facet is likely to affect other facets*. Figure 8.1 shows a simplistic view of complexity flowing from one facet of a project to the next. For example a complex set of requirements,

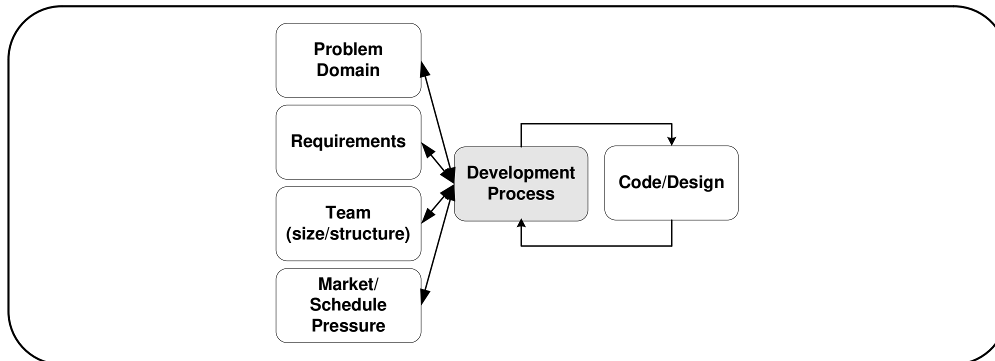

Figure 8.1: Flow Of Complexity Between the Facets of a Software Project

or a large software development team will increase the complexity of the development process. This will eventually have ill-effects on the source code of the project. Also the code and the design of the project can as well affect the development process, for example a complex design or spaghetti code will complicate the code development process.

Somewhat distant from the source code, researchers have demonstrated techniques and approaches to measure and control the complexity of the design and architecture [KBWA94, Par72, VN96, WB99]. Other researchers developed of approaches to deal with the complexity of requirements [BG02]. Furthermore, researchers have shown the effect of the team's structure and the interaction between its members on the structure of the software system and its complexity [BHB99b, HG99].

Though many of the previous approaches to study complexity have provided promising results, we believe they have certain limitations. For example, the requirements for a projects may not be well documented and may be changing throughout the lifetime of the project. In many cases, these measurements may not reflect the actual complexity that the project and software developers will have to deal with when implementing these requirements [Nor02]. For example, even though the requirements might be simple, the source code may be overly complicated and implementing these requirements may not be an easy task. Complexity metrics based on the source code have their limitations as well, since they

do not consider the development process and requirements at hand. For example, a complex piece of code that has not changed since the start of a project is not likely to have faults appear in it. Furthermore, studies have shown that most complexity metrics are correlated to simple measures such as the LOC of the application [GKMS00, OA96, LPS02].

A promising approach is to consider historical information about the software system. Results by Yu *et al.* [YSD98] indicate that prior faults are good predictors of future faults in a software system. Results by Graves *et al.* [GKMS00], Khoshgoftaar *et al.* [KAJH99], and Leszak *et al.* [LPS02] indicate that the number of prior modifications to a file is a good predictor of its fault potential. In other words, the more a file is changed the higher the likelihood it will contain faults.

In this chapter, we expand on such approaches, which make use of historical information about a software system to gauge its reliability. We examine the complexity of the modifications done to the source code. We refer to this complexity as the **complexity** or **chaos of the code development process**. The code development process plays a central role in a software project. The process is responsible for producing the code needed to satisfy the requirements of the customers, while dealing with the complexity and challenges associated with the current code and the other facets of the project. A software system with a chaotic code development process is undesirable. It will likely produce a system with has many faults and the project is likely to face delays. We conjecture that:

> *A chaotic code development process negatively affects its outcome, the software system, such as the occurrence of faults.*

Using concepts from information theory, we defined models which capture our intuition about the complexity of modifications and the chaos of the code development process. We found that events such as large refactorings or delays in releases were accompanied with increases in our proposed model measurements [HH03c]. Furthermore, we demonstrated using mathematical regression analysis that our proposed model measurements correlate with faults [HH03b]. In this chapter, we are interested

in the ability of our proposed model measurements in predicting the fault potential of a software system. In particular, we compare the performance of predictors based on our complexity models with the performance of predictors based on the number of prior modifications and prior faults. Based on a case study using six large open source projects, our results indicate that our development process chaos models are good predictors of fault potential when compared with other historical approaches (such as prior modifications and prior faults) to predict faults.

8.1.1 Overview Of Chapter

This chapter is organized as follows. Section 8.2 gives our view of the code development process. Instead of monitoring code changes as they are performed by developers, we turn our attention to source control systems used by software projects. Source control systems provide a convenient repository of data to study the code development process. Section 8.3 presents the mathematical concepts needed to measure the chaotic nature of the code development process. In particular, we introduce information theory and Shannon's entropy.

Section 8.4, 8.5, and 8.6 present the complexity models we use in our work. Section 8.4 introduces our first and simplest model for code development process complexity – **The Basic Code Development (BCD) Model**. We then proceed to give a more elaborate and complete model in Section 8.5 – **The Extended Code Development (ECD) Model**. Both these models calculate a single value that measures the overall chaotic nature of a project within some time period. Then in Section 8.6, we reformulate the ECD model to introduce a finer grained model – **The File Code Development (FCD) Model**. The FCD Model measures the effects of the chaotic nature of development process on individual source code files or subsystems.

In Section 8.7, we empirically compare the performance of predictors based on the FCD model with the performance of predictors based on the

number of prior modifications and prior faults using data from six large open source projects. We end Section 8.7 with a critical review of our findings and their applicability to other software systems.

Section 8.8 presents related work in the field of software evolution, entropy, and open source systems. Section 8.9 summarizes our findings.

8.2 The Code Development Process

We use the term **code development process** to mean the pattern of modifications to the source code of a project. The modifications are done by developers to implement new features and repair faults. By studying these patterns of modifications and quantifying their degree of complexity over time (using defined models), we hope to achieve a better understanding of the evolution of complexity facing developers working on a project.

Source control systems are used extensively by large software projects to control and manage their source code [Roc75, Tic85]. Data stored in these repository presents a great opportunity to study the code development process and validate our ideas. The data collection costs are minimal since it is collected automatically as modifications are done to the source code.

The repository of a source control system contains various details about the development history of every file in a project. It contains the creation date of a file, its initial content and a record of every modification done to the file. A **modification record** stores the date of the modification, the name of the developer who performed the changes, the number of lines that were changed, the actual lines of code that were added or removed, and a detailed message entered by the developer explaining the reasons for the change.

Using a lexical technique, similar to [MV00], we automatically divide modifications into three types based on the content of the detailed message attached to a modification:

Fault Repairing modifications (FR): These are all the modifications which are done to fix a bug. In our analysis, we labeled all modifications which contain terms such as *bug*, *fix*, or *repair* in the detailed message as FR modifications. Fault repairing modifications represent the fault repair process which likely differs from the code development process. We expect repairs to be spread out through the source code. These modifications are not used in the calculation of the development process complexity. But they are used in the validation process in our case study presented in Section 8.7.

General Maintenance modifications (GM): These are modifications that are mainly bookkeeping modifications and do not reflect the implementation of a particular feature. These modifications are removed from our analysis and are never considered. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well.

Feature Introduction modifications (FI): These are the modifications that are done to add or enhance features. Using our lexical analysis, we labeled all modifications that are not FR or GM modifications as FI modifications. These modifications are used in the calculation of the development process complexity.

A software system which has to endure highly scattered modifications to its code base as it implements the requirements of its customers, will have a high tendency of becoming a complex project. In contrast, a project where modifications are limited to specific spots in the code will have less complexity associated with it. A complex code base, the addition of a large number of features within a short period of time, or a large number of developers simultaneously changing the source code of a project are some of the many reasons that could cause the code modifications to be highly scattered. This scatter of modifications throughout the source

code, within a short time, makes it difficult for developers working on the project to track of its progress and the changes. For instance in [LPR98], Lehman *et al.* noted that the portion of a software system changed during a release tends to remain constant in relation to the rest of the software system over time, and that a sudden increase in the scatter of the changes during a release is likely to have adverse affect on the software system as noted in their OS/360 case study.

Various observations by Brooks support our intuition and our model [Bro74]. In particular, Brooks warned of the decay of grasp of what is going in a complex system. A complex modification pattern will cause delays in releases, high bug rates, stress and anxiety to all the personnel involved in a project. As the ability of team members to understand the changes to the system deteriorates so does their knowledge of the system. New development performed by them will be negatively affected. Similarly, Parnas warned of the ill-effects of *Ignorant Surgery*, modifications done by developers who are not sufficiently knowledgeable of the code [Par94]. Such ignorance may be due to the developers being junior developers or it may be due to the fast past of development which prevents developers from keeping track of other changes. For instance, in a study of the root cause of faults in a large telephony system it was determined that over 35% of faults where due to change coordination, missing awareness, communication, or lack of system knowledge problems [LPS02]. Information hiding [Par72] and good designs attempt to reduce the need to track other changes, but as the scatter of changes increases so does the likelihood that developers will miss tracking changes that are relevant to their work and managers will have a harder time allocating testing resources or tracking the progress of the project. In short, a chaotic code development process is a good indicator of many project problems.

over 35

In this section, we explained the need to measure the complexity of the development process and the amount of predictability and chaos that is associated with code modifications as a system evolves. We have yet

to explain how we can measure such complexity. In the following section we introduce concepts from information theory that will form the mathematical basis of our models to measure the complexity of the development process.

8.3 Information Theory

In 1948, Shannon laid down the basis of *Information Theory* in his seminal paper - *A mathematical theory of communication* [Sha48]. Information theory deals with assessing and defining the amount of information in a message. The theory focuses on measuring uncertainty which is related to information. For example, suppose we monitored the output of a device which emitted 4 symbols, A, B, C, or D. As we wait for the next symbol, we are uncertain as to which symbol it will produce ($i.e.$ we are uncertain about the distribution of the output). Once we see a symbol outputted, our uncertainty decreases. We now have a better idea about the distribution of the output; this reduction of uncertainty has given us information.

Shannon proposed to measure the amount of uncertainty/entropy in a distribution. The ***Shannon Entropy***, H_n is defined as:

$$H_n(P) = -\sum_{k=1}^{n}\Big(p_k * \log_2 p_k\Big),$$

where $p_k \geq 0, \forall k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$.

For a distribution P where all elements have the same probability of occurrence ($p_k = \frac{1}{n}, \forall k \in 1, 2, ..., n$), we achieve *maximum entropy*. On the other hand for a distribution P where one of the elements i has probability of occurrence $p_i = 1$ and all other elements have probability of occurrence equal to zero ($i.e.$ $p_k = 0, \forall k \neq i$), we achieve *minimal entropy*.

By defining the amount of uncertainty in a distribution, H_n describes the minimum number of bits required to uniquely distinguish the distri-

bution. In other words, it defines the best possible compression for the distribution (*i.e.* the output of the system). This fact has been used to measure the quality of compression techniques against the theoretically possible minimum compressed size.

8.4 The Basic Code Development Model

If we view the code development process of a software system as a system which emits data, and we define the data as the FI modifications to the source files, we can apply the ideas of information theory and entropy to measure the amount of *uncertainty* / *chaos* / *randomness* in the development process.

In the following three sections, we present three models which capture the entropy of development at different levels of detail. In this section, we present the Basic Code Development (BCD) Model for the entropy of software development and its evolution. The BCD model measures the overall development complexity for a software project. The following section extends the model to be more complete. Then the third section expands the model to deal with the effect of complexity on the files or subsystems in a software system instead of simply quantifying the overall code development complexity for a software project.

8.4.1 Basic Model

Suppose we have a software system which consists of four files. If we were to examine the development history of this system which is stored in a source repository, we will find for each file the dates for each modification to the file and the reason for modifying the file. We only concentrate on FI modifications.

Once the FI modifications are extracted, we can plot for each file the moments in time that specific file was modified. As can be seen in Figure 8.2, we put stars to indicate that for a specific file when it was modi-

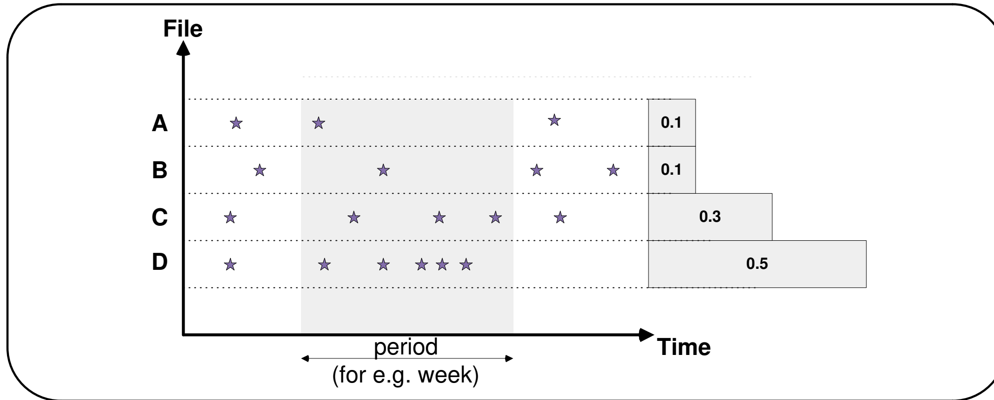

Figure 8.2: The Entropy of a Period of Development

fied. We now define a period of time, for example a week, or a month. For that period of time, we can define a file modification probability distribution P. P gives the probability that $file_i$ is modified in a period. For each file in the system, we count how many times it was modified during a period and divide by the total number of modifications in that period for all files. For example, in Figure 8.2, in the highlighted grey period we have 10 modifications for all the files in the system. $file_A$ was modified once so we have a $p(file_A) = \frac{1}{10} = 0.1$. For $file_B$ we get $p(file_B) = \frac{1}{10} = 0.1$, for $file_C$ we get $p(file_C) = \frac{3}{10} = 0.3$, and so on. On the right side of Figure 8.2, we can see a graph of the file modification probability distribution P for the shaded period.

If we monitor the modifications to the files of a software system and find that the probability of modifying $file_A$ is 1 and all other files is zero, then we have minimal entropy. On the other hand, if the probability of modifying each file is equal ($i.e.\ file_k = \frac{1}{n}$) then the amount of entropy/chaos in the system is at its maximum.

Instead of simply using the number of modifications to the file, we use the number of lines modified over the period to build the file modification probability. The lines changed in a modification is the sum of the lines added and deleted as described in the modification record.

8.4.2 Intuition

Consider these two changes. In the first change, the developer had to modify over a dozen files to add a feature. When asked about the steps required to add the feature, she/he may not recall half of them. Whereas another change to add a different feature required the modification of a single file. Recalling the modifications required for the latter feature is much easier. Intuitively, if we have a software system that is being modified across all or most of its files, the developers and the managers will have a hard time keeping track of all these modifications. The concern about the complexity of keeping track of scattered changes have been expressed by many developers working on large software systems, such as telephony systems [SGM+98].

If we were to imagine the brains of developers as a storage system, then the lower the change entropy the easier it is for developers and for managers to recall and track what has changed. The number of bits needed to remember all these changes is proportional to the number of files that have been modified. Miller has shown that human short-term memory is limited, therefore information overload and losing grasp of the current structure of a software system and the latest modifications to it is quite possible [Mil56].

The BCD model focuses on quantifying the patterns of changes instead of measuring the number of changes or measuring the effects of changes to the structure of the source code. A developer may not be aware of all the essential information when making a change because the amount of information (recent changes) is too large and unpredictable to keep track within her/his short-term memory. Thus our measurement of the pattern of change activity is a reasonable indicator of overload.

Faults are introduced due to misunderstandings about the structure of the system and its current state. Entropy gives us a way to measure redundancy and patterns. Change patterns with low information content as defined by entropy are easier to track and remember by developers

and others working on the project. By being aware of the current state of the software system, developers are less likely to introduce bugs in it and managers are likely to have an easier time monitoring the project.

The BCD model, along with the next two models, do not incorporate Fault Repairing (FR) modifications in the entropy calculation, instead we only use the FI modifications. FR modifications are not used since they represent bug fixes which are likely to be more scattered and to touch areas that are not being developed during the current period. This property of bugs fixes is likely to inflate the entropy measurement for a period. Furthermore, bug fixes are not likely to introduce new features or functionality, instead they are simply revisiting old changes which developers are already aware of and are less likely to need recalling them. Nevertheless, the models could be redefined to include FI modifications if need be.

The models focus on quantifying entropy for several modifications within a period not just for a specific modification. This choice of grouping several modifications is likely to inflate the entropy measurements, but we are more concerned with variations across periods instead of the absolute entropy values. In addition, by grouping modifications we can gauge the challenges that managers and developers may have to deal with to cope with wide spread changes due to several modifications. Nevertheless, the models could be adjusted to quantify entropy for every modification.

8.4.3 Files As a Unit of Measurement

In the BCD model we use the file as our unit of code to build the modification probability distribution P for each period. Other units of code can be used, such as functions or code chunks that are determined by a person with good knowledge of the system. Our choice of files is based on the belief that a file is a conceptual unit of development where developers tend to group related entities such as functions, data types, *etc.*

Based on our experience in studying large software system we found this to be the norm with some notable exceptions. For example in the VIM text editor [VIM], we found two files $misc1.c$ and $misc2.c$ which comprise a substantial amount of the source code and which contain code that is not related.

Furthermore, in recent work [HH04c] we were able to empirically support this belief. We showed that the probability of two source code entities changing together is high, if both entities are contained in the same file, at least for large open source software systems written in the C programming language.

8.4.4 Evolution of Entropy

We can view the file modification probability distribution P_j for a period j, as a vector which characterizes the system and uniquely identifies its state. We can divide the lifetime of a software system into successive periods in time, and view the evolution of a software system as the repeated transformation of the development process from one state to the next. Looking at Figure 8.3, we can see the P_j's calculated for 4 consecutive periods with their respective entropy. This allows us to monitor the evolution of chaos/entropy in the development process. If the project and the development process are not under control nor managed well, then state of the system will head towards maximum entropy/chaos.

The manager of a large software project should aim to control and manage the entropy. Monitoring for unexpected spikes in entropy and investigating the reasons behind them would let managers plan ahead and be ready for future problems. For example, a spike in entropy may be due to an influx of developers working on too many aspects of the system concurrently, or to the complexity of the source code or to a refactoring or redesign of many parts of the system. In the refactoring case, the manager would expect the entropy to remain high for a limited time then to drop as the refactoring eases future modifications to the source code. On the other

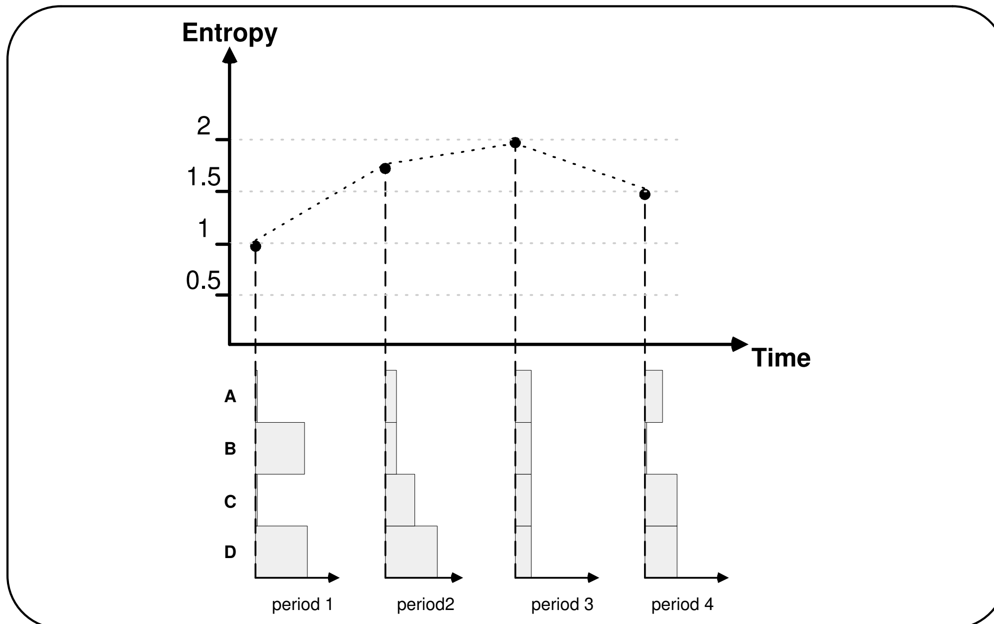

Figure 8.3: The Evolution of the Entropy of Development

hand, complex source code may cause a consistent rise in entropy over an extended period of time, till issues causing the rise in entropy/complexity are addressed and resolved [HH03c].

8.5 Extended Code Development Model

In this section, we extend our BCD model to address some of the characteristics and challenges associated with the evolution of large software systems. In the BCD model we used a fixed period size to measure the evolution of entropy. Also we assumed that the number of files in a software system remains fixed over time. The Extended Code Development (ECD) model presented in this section deals with these limitations.

8.5.1 Evolution Periods

In our BCD model, we presented the idea of using the file modification probability distribution as a vector to characterize each period in our study of the evolution of a software system. We used fixed length periods such as a month, or a year. We now present more sophisticated methods for breaking up the evolution of a software projects into periods:

Time based periods: This is the simplest technique and it is the one presented in the BCD model in Section 8.4. The history of development is broken into equal length periods based on calendar time from the start of the project. For example, we break it on a monthly or bi-monthly basis. A project which has been around for one year, would have 12 or 6 periods respectively. We chose a 3 month period in our experiments. The choice of three months was used mainly due to the fact that it represent a quarter and we believe that a quarter is a good amount of time to implement a reasonable amount of enhancements to a software system. This period creation method was used by us in [HH03c].

Modification limit based periods: The history of development is broken into periods based on the number of modifications to files as recorded in the source control repository. For example, we can use a modification limit of 500 or 1000 modifications. A project which has 4000 modifications would have 8 or 4 periods respectively. To avoid the case of breaking an active development week into two different periods, we attach all modifications that occurred a week after the end of a previous period to that period. To prevent a period from spanning a long time when little development may have occurred, we impose a limit of 3 months on a period even if the modification limit was not reached. We chose in our experiments a limit of 600 modifications. This period creation method was used by us in [HH03c]. The entropy values depend on the limits chosen to define the modification or time based periods. This dependency on the

period size is not a concern since for our purposes we are interested in the variations in the entropy values over time for similar sized periods rather than the absolute values.

Burst based periods: Based on studying the development history for several large software systems, we observed that the modification process is done in a bursty pattern. Over time, we see periods with many code modifications then they are followed by short periods of no or little code modifications. We chose to use that observation to automatically break up the development history into periods. If we find a period of a couple of hours where no code modifications have occurred, we consider all the previous code modifications to be part of the previous period and we start a new period. This period creation method is used in our case study presented in Section 8.7 and in [HH03b]. The Burst based period creation method is the most general method, as we do not need to specify modification counts or time limits which may differ between projects or over time.

8.5.2 Adaptive System Sizing

As a software system evolves, the number of files in it changes; increasing as new files are added, or split, and decreasing as files are removed, or merged. We need to adjust our entropy calculations, presented in Section 8.4, to deal with the varying number of files in a software system.

To compare the entropy when there is a varying number of files in the software system, we define H, which we will call ***Standardized Static Entropy*** as:

$$\begin{aligned} H(P) &= \frac{1}{Max\ Entropy\ for\ Distribution} * H_n(P) \\ &= \frac{1}{\log_2 n} * H_n(P) \\ &= -\frac{1}{\log_2 n} * \sum_{k=1}^{n} \Big(p_k * \log_2 p_k\Big), \end{aligned}$$

$$= \quad -\sum_{k=1}^{n}\Big(p_k * \log_n p_k\Big),$$

where $p_k \geq 0, \forall k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$. The standardized static entropy H normalizes Shannon's entropy H_n, so that $0 \leq H \leq 1$. We can now compare the entropy of distributions of different size, such is the case when we examine the various periods of a software system as new files are added or removed. It is interesting to note that using standardized static entropy H, we could compare the entropy between different software projects. For example, we could compare the evolution of two operating systems side by side or even an operating system and a window manager.

The Standardized Static Entropy, H, depends on the number of files in a software system, as it depends on n. For many software system there exist files that are rarely modified, for example, platform and utility files [LPR98]. Developers working on the software systems do not need to worry about tracking changes to these files, as the probability of them changing is very low. To prevent these files from reducing the standardized entropy measure, we defined a working set standardized entropy H' – **Adaptive Sizing Entropy**. In H' instead of dividing by the actual current number of files in the software system, we divide by the number of *recently* modified files. We define the set of recently modified files using two different criteria:

Using Time: The set of recently modified files is all files modified in the preceding x months, including the current month. In our experiments we used 6 months. Our choice of six months as a window originates from our belief and our experience developing large software systems. We found that usually what is hot (relevant) at the beginning of the year and is the focus of the development tends not to be a concern towards the end of the year. This is mainly due to the fact that throughout the earlier part of the year most of the problems and features related to these files are addressed.

Using Previous Periods: The set of recently modified files is all files modified in the preceding x periods, including the current period. We don't show results from using this model in this chapter but in our experiments we used 6 periods in the past to build the working set of files.

As we have two different criteria to create a period based on size, then we have two different results based on the use of a time based or a modification limit period creation models.

An adaptive sizing entropy H' usually produces a higher entropy than a traditional standardized entropy H, as for most software systems there exists a large number of files that are rarely modified and would not exist in the recently modified set. Thus the entropy would be divided by a smaller number. In some rare cases, the software system may have undergone a lot of changes/refactorings and it may happen that the size of the working set is larger than the actual number of the files that currently exist in the software system, as many files may have been removed recently as part of a cleanup [HH03c]. In that rare case, an adaptive sizing entropy H' will be larger than a traditional standardized entropy H.

8.6 The File Code Development Model (FCD)

The two previously presented models in Sections 8.4 and 8.5 produce a value which quantifies the entropy for each period in the development lifetime of a software system. We have used the ECD model to monitor the evolution of entropy for open source projects in [HH03c] and to correlate events in the project's history to spikes or drops in the entropy measurements.

In this section, we extend the ECD model to deal with assigning a complexity value to a file. By assigning a complexity value to a file we can later (*see* Section 8.7) measure the ability of our entropy models to predict faults in specific files or subsystems.

We believe that files that are modified during high complexity/chaotic development periods, as determined by our ECD Model, will have a higher tendency to contain faults as the developers performing the changes won't have a good grasp of the latest changes to the source code. We define the **History Complexity Metric (HCM)** for each file in a software system. The HCM assigns to a file the effect of the complexity of a period, as calculated by our ECD model. A file that has been modified during periods of high complexity/entropy will have a high HCM value to indicate that the file will tend to be more prone to faults.

Given a period i, with entropy H_i where a set of files, F_i are modified with a probability p_j for each file $j \in F_i$, we define **History Complexity Period Factor (**$HCPF_i$**)** for a file j during period i as:

$$HCPF_i(j) = \begin{cases} c_{ij} * H_i, & j \in F_i \\ 0, & otherwise \end{cases}$$

c_{ij} is the contribution of entropy for period i (H_i) assigned to file j. We define three $HCPF$ by varying the definition of c_{ij}:

- $HCPF^1$ with $c_{ij} = 1$: This factor assigns the full complexity value (H_i) to every modified file in a period ($j \in F_i$). This is the simplest model.
- $HCPF^2$ with $c_{ij} = p_j$: This factor assigns a percentage of the complexity associated to a period (H_i). The percentage is the probability of file j being modified during period i.
- $HCPF^3$ with $c_{ij} = \frac{1}{|F_i|}$: This factor distributes evenly the complexity associated to a period (H_i) between all modified files in that period.

More elaborate definitions of $HCPF$ are possible but this is beyond the scope of this chapter and our validation process.

Now we define the **History Complexity Metric (**HCM**)** for a file j over a set of evolution periods $\{a, .., b\}$ as:

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} HCPF_i(j)$$

We use this simple HCM definition to indicate that complexity associated to a file keeps on increasing over time, as a file is modified. Using this simple HCM and our three $HCPF$ definitions, we have three HCM metrics namely: HCM^{1s}, HCM^{2s}, and HCM^{3s}, where the s superscript indicates the use of the simple HCM formula. In addition, we define a more elaborate HCM^{1d}, which employs a decay model using the simplest $HCPF$ ($HCPF^1$). In HCM^{1d}, earlier modifications would have their contribution to the complexity of the file reduced in an exponential fashion over time. Similar decay approaches have been used by us in [HH] (*see* Chapter 7) and others [GKMS00]:

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} e^{\phi * (T_i - Current\ Time)} HCPF_i^1(j),$$

where T_i is the end time of period i and ϕ is the decay factor.

We define the HCM for a subsystem S over a set of evolution periods $\{a, .., b\}$ as the sum of the HCMs of all the files that are part of that subsystem:

$$HCM_{\{a,..,b\}}(S) = \sum_{j \in S} HCM_{\{a,..,b\}}(j)$$

If a file were to move from one subsystem to another during a studied evolution period, the moved file would contribute to the HCM of its old subsystem till the time it was moved. Then it would contribute to its new subsystem afterwards.

Using the 4 defined HCMs at the subsystem level (HCM^{1s}, HCM^{2s}, HCM^{3s}, and HCM^{1d}), we proceed to validate that the HCM metric is a better predictor of faults in a software system compared to using the number of prior modifications or prior faults. This validation provides a concrete substantiation of our code development complexity model.

8.7 Case Study

In this section we present a quantitative case study which reexamines prior research results, which make use of historical information to predict faults, and compares fault predictors based on our entropy models to these results. We focused on studying three aspects of fault predictors:

- **Modifications vs. Faults:** This study compares the performance of *prior modifications* with the performance of *prior faults* in predicting future faults in a software system.
- **Modifications vs. Entropy:** This study compares the performance of *prior modifications* with the performance of our HCM *entropy models* in predicting future faults in a software system.
- **Faults vs. Entropy:** This study compares the performance of *prior faults* with the performance of our HCM *entropy models* in predicting future faults in a software system.

Application Name	Application Type	Start Date	Subsystem Count (*high level*)	Subsystem Count (*low level*)	Prog. Lang.
NetBSD	OS	March 1993	25	235	C
FreeBSD	OS	June 1993	33	152	C
OpenBSD	OS	Oct 1995	28	265	C
Postgres	DBMS	July 1996	16	280	C
KDE	Windowing System	April 1997	32	108	C++
Koffice	Productivity Suite	April 1998	85	158	C++

Table 8.1: Summary of the Studied Systems

To perform our study we used several open source software systems. Table 8.1 summarizes the details of the software systems we studied. The oldest system is over ten years old and the youngest system is five years

old. We based our analysis on the first five years in the life of each studied open source project. We chose to ignore the first year in the source control repository, due to the special startup nature of code development during that year as each project initializes its repository. Our case study employed an approach similar to [GKMS00], in particular:

- We built Statistical Linear Regression ($SLR\ Model$) models for every software system in Table 8.1. These $SLR\ Model$s used data from the second and third years from the source control repository to predict faults in the fourth and fifth years of the software project. In total, we build six SLR models: 4 models for the HCM entropy metrics, one for prior faults, and one for prior modifications. All the built SLR models predicted faults in the fourth and fifth years.
- We then measured the amount of error in each model and compared it to the other models. In particular, we compared
  - The performance of modifications and fault models.
  - The performance of modifications and entropy models.
  - The performance of faults and entropy models.
- We performed statistical tests to determine that the difference in error is statistically significant and not due to the natural variability of the studied data.

In the following subsections, we elaborate on these steps.

8.7.1 Building the Statistical Linear Regression Models

To perform our studies, we built six Statistical Linear Regression ($SLR\ Model$) models for each software system in Table 8.1, namely:

$SLR\ Model_m$: uses the number of modifications to predict faults.

$SLR\ Model_f$: uses the number of faults to predict faults.

$SLR\ Model_{HCM1s}$**:** uses the HCM_{1s} values to predict faults.

$SLR\ Model_{HCM2s}$**:** uses the HCM_{2s} values to predict faults.

$SLR\ Model_{HCM3s}$**:** uses the HCM_{3s} values to predict faults.

$SLR\ Model_{HCM1d}$**:** uses the HCM_{1d} values to predict faults.

The built SLR models have the following form, where y is the dependant variable and x is the predictor/independent variable:

$$y = \beta_0 + \beta_1 x$$

For each model, y represents the number of faults in a subsystem. This is determined based on the number of Fault Repairing (FR) modifications in the fourth and fifth years in the source control repository data. As for x, it represents the value for the following variables from the data in the second and third year of the source control repositories:

$SLR\ Model_m$**:** x represents the number of modifications.

$SLR\ Model_f$**:** x represents the number of faults.

$SLR\ Model_{HCM1s}$**:** x represents the HCM_{1s} for each subsystem.

$SLR\ Model_{HCM2s}$**:** x represents the HCM_{2s} for each subsystem.

$SLR\ Model_{HCM3s}$**:** x represents the $HCM1_{3s}$ for each subsystem.

$SLR\ Model_{HCM1d}$**:** x represents the HCM_{1d} for each subsystem.

The HCM models are based on the ECD bursty model that has a one hour quiet time between bursts. The HCM_{1d} uses a decay factor (ϕ) of 10, which minimizes the error for the $SLR\ Model_{HCM1d}$. To ensure the mathematical validity of our SLR models, we actually use the mathematical log of the x values, instead of x. The use of a log transformation ($e.g.$ $log(number\ of\ modifications)$) stabilizes the variance in the error for each

data point in the SLR model, a requirement for linear regression models which assume that the error variance is always constant [Wei80]. The SLR model parameters (β_0 and β_1) are estimated using the fault data from the fourth and fifth years. Table 8.2 shows the R^2 statistic to indicate the quality of the fit. The better the fit, the higher the R^2 value. A zero R^2 indicates that there exits no relationship between the dependant y and independent variable x. We notice that the C systems have a better fit in comparison to the C++ systems (KDE and Koffice) for all the SLR models. The $SLR\ Model_{HCM1d}$ has the best fit of all the SLR models for all the studied systems.

Application	R_f^2	R_m^2	R_{1s}^2	R_{2s}^2	R_{3s}^2	R_{1d}^2
NetBSD	0.57	0.55	0.54	0.53	0.61	0.71
FreeBSD	0.65	0.48	0.57	0.58	0.59	0.65
OpenBSD	0.45	0.44	0.54	0.55	0.54	0.57
Postgres	0.57	0.36	0.49	0.51	0.60	0.61
KDE	0.31	0.26	0.28	0.29	0.36	0.57
Koffice	0.30	0.27	0.33	0.33	0.27	0.41

Table 8.2: The R^2 statistic for all the SLR Models for the Studied Systems

8.7.2 Measuring and Comparing the Prediction Error for the SLR Models

Once we estimated β_0 and β_1 for the SLR Models for every system, we measured the amount of prediction error. Mathematically for every model with β_0 and β_1 as parameters, we get a $\widehat{y_i}$ for every x_i, where $\widehat{y_i}$ is the number of expected faults in the subsystem in the fourth and fifth years:

$$\widehat{y_i} = \beta_0 + \beta_1 x_i$$

We define the absolute prediction error as

$$e_i = \mid \widehat{y_i} - y_i \mid$$

where y_i is the actual number of faults that occurred in subsystem i during the fourth and fifth years.

Thus the total prediction error of an SLR model is:

$$E = \sum_{i=1}^{n} e_i^2,$$

for all n subsystems in the software system under study. To achieve the goals of our study, we need to compare the prediction errors for the SLR models. For example, to determine if prior modifications are better than prior faults in predicting faults, we need to compare E_m with E_f, where E_m and E_f are the total prediction error for the $SLR\ Model_m$ and $SLR\ Model_f$ respectively.

8.7.3 Determining the Statistical Significance for The Difference in Prediction Error between Models

Unfortunately, simply comparing the total prediction errors ($e.g.\ E_m$ and E_f) is not sufficient. Instead we need to ensure that the difference in the prediction error is statistically significant and not due to the natural variance of the data.

To perform the statistical test of significance between two SLR Models ($SLR\ Model_A$ and $SLR\ Model_B$), we use a statistical paired T-test and formulate the following test hypotheses:

$$H_0 : \mu(e_{A,i} - e_{B,i}) = 0$$

$$H_A : \mu(e_{A,i} - e_{B,i}) \neq 0,$$

where $\mu(e_{A,i} - e_{B,i})$ is the population mean of the difference between the absolute error of each observation pair. As the data size is large enough

(the smallest software system has over 80 subsystems) and the T-test is robust for non-normally distributed data, we can safely use a T-test. Alternatively, a non-parameterized test such as a U-test can be used for smaller software systems [MW47].

If the null hypothesis H_0 holds then the difference in prediction is not significant. Thus we need H_0 to be rejected, with a high probability.

8.7.4 Comparing Models

In this subsection, we conduct our study to determine the quality of the prediction of the built models. We compare the prediction error between several of the SLR Models while ensuring that the difference in prediction error is statistically significant using the definitions in the earlier subsections.

8.7.4.1 Modifications vs. Faults

Application	$E_m - E_f$ (%)	$P(H_0 \ holds)$
NetBSD	+11.7 (+04%)	0.67
FreeBSD	+71.2 (+48%)	0.00
OpenBSD	+03.7 (+02%)	0.84
Postgres	+47.2 (+49%)	0.02
KDE	+26.3 (+07%)	0.32
Koffice	+26.3 (+04%)	0.51

Table 8.3: The Difference of Error Prediction and T-Test Results for the $SLR\ Model_m$ and $SLR\ Model_f$ for the Studied Systems

We are interested in determining if prior modifications are better than prior faults in predicting faults; therefore, we compare the total prediction error for both the $SLR\ Model_m$ and $SLR\ Model_f$. The second column in Table 8.3 shows the percentage of difference in prediction error when $SLR\ Model_m$ is used instead of $SLR\ Model_f$. The results indicate that

the number of prior faults is a better predictor of faults than the number of prior modifications. The third column shows the results for the T-test which determines if the difference is statistically significant or if it is due to the natural variability of the data. The T-test on paired observations of absolute error was significant at better than 0.02 for the FreeBSD and Postgres systems (marked in grey in Table 8.3). For these two systems, we are over 98% confident that the increase in prediction error between $SLR\ Model_f$ and $SLR\ Model_m$ is statistically significant. Whereas for the other systems, the increase is not statistically significant indicating the performance of both models (prior faults or prior modifications) is statistically similar.

> These results indicate that prior faults should be used to predict faults instead of using prior modifications. Using a predictor based on prior modifications may cause a 48.5% rise in prediction error when compared to a prior faults predictor.

8.7.4.2 Modifications vs. Entropy

Application	$E_{HCM3s} - E_m$ (%)	$P(H_0\ holds)$	$E_{HCM1d} - E_m$ (%)	$P(H_0\ holds)$
NetBSD	-39.8 (-14%)	0.03	-106.5 (-36%)	0.00
FreeBSD	-47.4 (-22%)	0.02	-72.0 (-33%)	0.00
OpenBSD	-40.4 (-18%)	0.01	-53.8 (-23%)	0.00
Postgres	-52.7 (-37%)	0.04	-56.9 (-40%)	0.03
KDE	-52.1 (-13%)	0.01	-165.2 (-42%)	0.00
Koffice	+03.3 (+01%)	0.83	-69.9 (-18%)	0.01

Table 8.4: The Difference of Error Prediction and T-Test Results for the $SLR\ Model_m$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$ for the Studied Systems

Given that our entropy models are derived from the number of modifications to the source code, we are interested in comparing the performance of a predictor based on prior modifications with predictors based on our HCM entropy models. We chose the simple $SLR\ Model_{HCM3s}$ and the decay $SLR\ Model_{HCM1d}$ to compare with the $SLR\ Model_m$. Both HCM models were the best two performing HCM models based on the R^2 statistic in Table 8.2. The second and fourth columns in Table 8.4 shows the percentage of difference in prediction error when the $SLR\ Model_{HCM3s}$, or the $SLR\ Model_{HCM1d}$ are used instead of $SLR\ Model_m$ respectively. The third and fifth columns in Table 8.4 show the results for the T-test which determines if the difference in prediction error is statistically significant or if it is due to the natural variability of the data. Greyed cells in Table 8.4 indicate that we are 95% confident that the decrease in prediction error for $SLR\ Model_{HCM3s}$, or that the $SLR\ Model_{HCM1d}$ is statistically significant except for the Koffice system. For the Koffice system, the difference in prediction error is not statistically significant and is likely due to the normal variability of the data.

> These results indicate that both HCM (simple and decay) based model are statistically likely to always outperform prior modifications in predicting future faults. The decrease in prediction error using an HCM model ranges between 13% to 40% when compared to the prediction error of a model based on prior modifications.

8.7.4.3 Faults vs. Entropy

Section 8.7.4.1 showed that the number of prior faults is a better predictor of future faults than the number of modifications. Section 8.7.4.2 showed that models based on our entropy metrics are better predictors of faults than the number of modifications. We would like to compare the performance of predictors based on our entropy metric models (HCM models) with a predictor based on the number of prior faults. We chose the simple $SLR\ Model_{HCM3s}$ and the decay $SLR\ Model_{HCM1d}$ to compare with

Application	$E_{HCM3s} - E_f$ (%)	$P(H_0\ holds)$	$E_{HCM1d} - E_f$ (%)	$P(H_0\ holds)$
NetBSD	-28.14 (-10%)	0.26	-94.84 (-34%)	0.00
FreeBSD	+23.81 (+16%)	0.30	-00.79 (-01%)	0.97
OpenBSD	-36.59 (-16%)	0.02	-50.05 (-22%)	0.01
Postgres	-05.53 (-06%)	0.71	-09.71 (-10%)	0.55
KDE	-25.72 (-07%)	0.32	-138.87 (-38%)	0.01
Koffice	+19.20 (+05%)	0.34	-54.07 (-15%)	0.04

Table 8.5: The Difference of Error Prediction and T-Test Results for the $SLR\ Model_f$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$ for the Studied Systems

the $SLR\ Model_f$ model. Both HCM models were the best two performing HCM models based on the R^2 statistic in Table 8.2. The second and fourth columns in Table 8.5 shows the percentage of difference in prediction error when the $SLR\ Model_{HCM3s}$ or the $SLR\ Model_{HCM1d}$ are used instead of $SLR\ Model_f$ respectively. The third and fifth columns in Table 8.5 show the results for the T-test which determines if the difference in prediction error is statistically significant or if it is due to the natural variability of the data. Greyed cells in Table 8.5 indicate that the difference between prediction errors is statistically significant. For the $SLR\ Model_{HCM3s}$ model, only the cell for the OpenBSD system is grey indicating that the improvement in prediction error for this system is statistically significant. These results indicate that the $SLR\ Model_{HCM3s}$ performs as good as the number of prior faults for all studied systems except for the OpenBSD where it outperforms the prior faults predictor by 16%. For the $SLR\ Model_{HCM1d}$, all cells except the ones corresponding to FreeBSD and Postgres are grey. These results indicate that $SLR\ Model_{HCM1d}$ outperforms the number of prior faults in predicting future faults except for the FreeBSD and Postgres systems where it performs as good as the prior faults.

These results indicate that models based on our entropy metrics are as good as (or even better) predictors of faults than prior faults for most studied software systems. The decrease in prediction error using an HCM model ranges between 15% to 38% when compared to the prediction error of a model based on prior faults.

In this subsection, we have shown that:

- The number of prior faults is a better predictor of future faults than the number of prior modifications.
- The HCM based predictors are better predictors of future faults in large software systems when compared with predictors based on prior modifications or prior faults.

8.7.5 Threats to Validity

Our case study has produced statistically significant results showing that a complex code development process negatively affects the software system by causing the appearance of faults. We used the FCD model to quantify complexity in the code development process. We used the count of Fault Repairing (FR) modifications as an indicator of the quality of the software system. However, we must carefully consider our results before applying our findings elsewhere. In this subsection, we present a critical analysis of our findings.

Empirical research studies should be evaluated to determine whether they were able to measure what they were designed to assess. Therefore we need to determine if our findings are sufficient to support our conjecture about the code development process and its negative effect on the software system. Four types of tests are used [Yin94]: construct validity, internal validity, external validity, and reliability.

8.7.5.1 Construct Validity

Construct validity is concerned to the meaningfulness of the measurements – Do the measurements quantify what we want them to? The main conjecture of our work is that a complex code development process negatively affects the software system by causing faults to appear in it. We used as a dependant variable, the number of Fault Repairing (FR) modifications as recorded by the source control system. However, we do not consider bugs that may have been found but never fixed, as we used the bug fixes as recorded by the source control system instead of using the reported bug counts stored in a defect management system. There may exist subsystems in which a large number of bugs have been discovered yet they were never fixed during our period of analysis. We believe the chance of this occurring is low nevertheless it is a possibility. Furthermore, the number of fixed bugs are likely to be correlated to the number of discovered bugs. Alternatively, we could have used data from defect management systems. Unfortunately, such defect tracking systems do not exist for most of the studied software systems.

8.7.5.2 Internal Validity

Internal validity deals with the concern that there may be other plausible rival hypotheses to explain our findings – Can we show that there is a cause and effect relation between the complexity of the code development process and the occurrence of faults, or are there other possible explanations? Demonstrating causality requires more than simply showing statistically significance relations, we need to show temporal precedence as well. We need to show that the complex code development process caused the appearance of faults in the software system. Unfortunately, this is a rather hard task and may be difficult to demonstrate, as we believe the complexity in the code development process interact with all the other project facets in a feedback loop as shown in Figure 8.1. A complex code base requires complex development process to maintain it and a complex

development process produces a complex code base. Furthermore, a complex set of requirements may cause the development process to become process which in turn may cause the appearance of faults in the software system. Therefore to show true casuality we would need to build a richer and detailed theory which can measure the effect of the feedback loop on the interacting facets in a software project. We believe this would be a very challenging task and may require us to perform a controlled experiment with subjects but then the results of such experiment would have a much weaker external validity (*i.e.* would be hard to generalize). Our results do not show a casuality relation but intuitively we believe that a complex code development process negatively affects the software system.

8.7.5.3 External Validity

External validity tackles the issue of the generalization of the results of our study – Can we generalize our results to other software systems and projects? We believe that the external validity of our results is reasonably high.

The use of the detailed historical records stored in source control systems ensures that the studied code development process is a realistic process which involves experienced developers working on large software systems over long periods of time. Alternatively, we could have performed controlled experiments which would run for limited time. We would not be able to confidently simulate realistic change patterns. In that case we would not be able to have individuals with such experience and knowledge performing simulated modifications to the source code.

Furthermore, we examined a large number of software systems. Each of these software systems is developed by a large number of developers, over several years, using a variety of modern programming languages (C and C++).

Although we examined a large number of software systems, the systems used in our study are all open source systems which have several

interesting characteristics that may not hold for other commercial systems. Some of the characteristics are: the large size of the projects' code base, the large size of the development team, the distribution of the development team around the world (distributed development – with team member's rarely meeting in person and relying on electronic communications such as emails and newsgroups instead of in-person meetings such as water cooler and lunch time conversations), and the self selective nature of the team (*i.e.* developers volunteer to work on the project and have full freedom to choose which areas to contribute to). All these characteristics contribute to limiting the generalization of our results. We believe that our results are generalizable to large open source systems with an extended network of developers spread out throughout the world. Our results are likely to generalize as well to commercial software systems which are developed by teams distributed around the world, and probably even to commercial software systems developed in a single location. We need to study a few commercial systems, before we can confidently generalize our results.

8.7.5.4 Reliability

Reliability refers to the degree to which someone analyzing the data would reach the same conclusions/results. We believe that the reliability of our study is high. The data used in our study is derived from source control systems. Such systems are used by most large software systems which makes it possible for others to easily run the same experiments on other data sets to reproduce our findings.

The gathering of the data and its classification has been carefully documented by us and explained. In particular, we used an automatic lexical based technique, as described in Section 8.2, to classify modifications to the source code as fault repairs, feature enhancements, or general modifications. One threat to validity is the reliability of such an automated classification and the probability of others performing similar classifications. Alternatively, a manual classification of all modifications to the source

code may increase the reliability of our study when conducted by others. Unfortunately, such manual classification is neither possible nor feasible for large long lived software systems. Furthermore, we would have to deal with the consistency of classifications done by different individuals.

Results by Mockus and Votta indicate that automated classifications of changes using a lexical approach show *moderate* agreement with manual classifications done by the developers who performed these changes for large commercial telephony systems [MV00]. In addition, a study by us shows that our automated classifications agree over 70% of the time with classifications done manually by professional software developers [HH04d] (*see* Chapter 4).

8.8 Related Work

Barry *et al.* use a volatility ranking system and a time series analysis to identify evolution patterns in a retail software system based on the source modification records [BKS03]. For example, Eick *et al.* studied the concept of code decay and used the modification history to create visualization of the change history of a project [EGK+01, ELL+92]. Graves *et al.* showed that the number of modifications to a file is a good predictor to the fault potential of the file [GKMS00]. Leszak *et al.* showed that that there is a significant correlation between the percentage of change in reused code and the number of defects found in those changed components [LPS02].

Mockus *et al.* uses source modification records to assist in predicting the development efforts in large software systems for AT&T [MWZ03]. Previous research has focused primarily on studying the source code repositories of commercial software systems for predicting faults or required effort. We believe that:

1. This focus on commercial source systems may limit the applicability of the results. The results may be dependent on the studied system or organization as only a single system is used in the validation.

2. By focusing on open source systems we are able to study a much larger set of systems to validate our findings and are more confident about our results.

We hope in future work to validate our findings against large commercial software systems to determine if our approach holds for such systems as well.

Whereas our model quantifies the complexity of the development process as calculated from the source code modification statistics, previous studies [AEH01, BCLV01, Cha95, Cha02, Har92, Wey88] quantify the complexity of the source code. For example, in previous models the distribution of special tokens in the source code or the control flow structure of the source are used to calculate the entropy. Our work aims to compute a measure of chaos in the development process instead of just focusing on computing the complexity of the source code. We conjecture that detecting chaos in the code development process will serve as an early warning measure to help prevent the occurrence of faults in the software system.

Work by Lehman *et al.* [Leh80, LB85, LRW+97] and Godfrey *et al.* [Mic00] focus on studying the evolution of size (number of modules) and LOC between releases of software systems. Instead we focus on measuring the entropy/chaos in the development process. In our presented case studies, we quantified the idea of complexity/entropy in the development process using our models and have shown that chaos is a good indicator of faults due to increase in complexity. Whereas our presented models study the evolution of complexity over time, Lehman advocates studying the evolution of software over releases/versions. Our models could be extended to use releases as a unit of observation instead of time. This would permit us to compare our findings to Lehman's previous findings and laws of evolution.

Outside of the software engineering domain, the measure of entropy has been used to improve the performance of Just In Time compilers and profilers [SY99]. It has been used for edge detection and image searching

in large image database [DV00]. Also, it has been used for text classification and several text based indexing techniques [DMK].

8.9 Conclusion

In this chapter, we presented a new perspective on the complexity of software. We examined the complexity and chaos associated with the development process. We view the development process as a system with an unknown output, in other words we are uncertain about the files that will be modified by the process over time. Using the ideas of uncertainty and entropy from information theory, we measure how much information exists in the development process. We hypothesize that too much information will require more effort for project members to keep track of the development process over time. As the entropy of the development process increases, developers are likely to lose their grasp of the state of the software system and their shared image of the software system is likely to deteriorate.

In particular, we conjecture that: *A chaotic / complex code development process negatively affects its outcome, the software system, such as the occurrence of faults*. We presented models to quantify the complexity in a software system over time based on the source code modification history as stored in the source control repository. Furthermore, we performed studies to gain a better understanding of the proposed models and to validate mathematically our conjecture. We used data derived from six large open source projects in our studies. Our case study has showed that the number of prior faults is a better predictor of future faults than the number of prior modifications. Also we showed that predictors based on our chaos models are better predictors of future faults in large software systems when compared with predictors based on prior modifications or prior faults.

The studies presented in this chapter may have a number of potential limitations as outlined in Section 8.7. Nevertheless, this chapter makes

a number of important contributions. It showcases the benefits of using source control systems to monitor the evolution of software system and plan for the success of the project. It is also the first work to define the idea of code development chaos using a sound mathematical concept (Shannon's Entropy) and to show the effects of change patterns on the quality of large software systems.

Brooks warned of the effect of the program maintenance process on the evolution of a software project:

> *"Program maintenance is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence."* Fred Brooks, The Mythical Man-Month [Bro74].

We believe that such obsolescence can be avoided and that software systems can be maintained and evolved for many years as long as anti-regressive activities, such as refactorings, are performed as suggested by Lehman [Leh80]. A good and up to date knowledge of the state of complexity in the software project is needed to perform such activities. The ideas and models for code development process entropy presented herein can offer such assistance. Using our entropy measurements, managers can monitor with great detail the evolution of complexity in their software project and thereby control it. Managers can also use our entropy measures as early warnings of potential faults occurring throughout their source code.

Part IV

Conclusion

CHAPTER 9

Contributions and Future Work

Data in software repositories, such as source control repositories, represents a valuable resource that is used by practitioners to maintain and manage software projects. We have presented techniques and approaches to transform software repositories from static record keeping repositories to active repositories used by software practitioners to predict and plan various aspects of their project. We overview our findings and discuss opportunities for extending our work.

A FEW years ago access to the source code of large applications was usually limited. In many cases, companies were reluctant to provide software engineering researchers access to their source code. As the open source movement gained popularity, researchers were finally able to acquire the source code for several open source projects. Researchers could finally apply their research ideas and verify their findings using large non-trivial software systems (*e.g.* [HGH01]).

As these open source systems evolved, they left behind them a huge trail of historical information which is recorded in a variety of repositories

such as source control systems, bug tracking systems, and mailing lists. Whereas access to such historical information for commercial systems is usually very limited, access to such information for open source systems is freely available and open. This historical information offers us the chance to understand better the evolution of software systems and to study the benefits of integration such historical information in traditional software engineering research and industrial practices.

Unfortunately, the process of acquiring such information in a convenient format is challenging, since such repositories are mainly designed as record keeping repositories. Also the large amount of data stored in these repositories complicates the data recovery process. The complexity of recovering this historical information has hindered other researchers from experimenting with it. The engineering contributions of this thesis to the area of software engineering is the proposal and development of evolutionary extractors that could recover such historical information and represent it in an easy to use format. Easy access to such rich and detailed data will encourage interested researchers and practitioners to explore the potential of historical project information and will assist researchers in gaining a better understanding of software development practises and evolutionary patterns [Per02].

The conceptual contribution of our work is the development of techniques and approaches that make use of this recovered historical information to augment traditional software engineering methods. For example, we demonstrated that attaching historical information to the dependency graph (*Source Sticky Notes*), could assist in architecture understanding and investigation techniques along with well established techniques such as the reflexion framework. We hope that our methods and techniques will encourage other researchers to experiment with enriching their current research methods and techniques with historical information. Moreover, we foresee that our results will not only incite practitioners to consider using such information in their work, but will encourage practitioners to offer researchers access to historical repositories for industrial projects.

In the following section, we summarize our contributions and findings.

9.1 Thesis Contributions and Findings

1. **Evolutionary Software Extractors:** We introduced the idea of an evolutionary extractor and advocated the need for such extractors to study and mine the evolutionary history of software projects from historical repositories such as source control repositories. We presented the implementation challenges and techniques for an evolutionary code extractor for the C programming language (C-REX).

2. **Source Sticky Notes:** We proposed the benefits of attaching historical information to each dependency in a software system. We showed that these notes are useful in speeding up and automating the software architecture understanding process.

3. **Development Replay Approach:** We argued for using historical information to assess the expected and claimed benefits of adopting new software maintenance tools and strategies. The Development Replay (DR) approach permitted us to investigate the effectiveness of a large number (over 20) of not-yet-developed software tools with no cost associated with conducting long term case studies or even building such tools. Our results show that a simple tool that uses historical co-change information combined with code layout (same file) information is likely to outperform tools that are based solely on either historical co-change information, code layout, or code structure information.

4. **Top Ten List:** We introduced the notion that not all bugs are created equal, instead managers are more concerned about bugs that are likely to occur in the near future versus faulty code that is not likely to have fault appear in it for some time. We proposed metrics and models to measure traditional bug prediction techniques using such notion and ideas. If we drew an analogy to bug prediction and

rain prediction, our prediction models focus on predicting the areas that are most likely to rain in the next few days. The predicted rain areas may be areas that are known to be dry areas (*i.e.* not fault prone) or may be areas which aren't known to have large precipitation values (*i.e* low predicted faults). We believe that the Top list approach holds a lot of promise and value for software practitioners, it provides a simple and accurate technique to assist them in allocating resources as they maintain large evolving software systems.

5. **Software Development Chaos:** We conjectured that a chaotic or complex development process negatively affects its outcome, the software system. We proposed a complexity metric that is based on the process followed by software developers to produce the code instead of on the code or the requirements. We showed through a case study that the number of prior faults is a better predictor of future faults than the number of prior modifications. Also we showed that predictors based on our development chaos models are better at predicting future faults in large open source software systems than predictors based on prior modifications or prior faults.

All of our contributions were validated through several case studies using a large number of open source software systems and through a survey of professional software developers.

The following section lists suggestions for possible extensions of the research work presented in this thesis.

9.2 Suggestions for Extending this Research

9.2.1 Evolutionary Extractors for C++ or Java

C-REX is an evolutionary extractor for the C programming language. We chose to develop an extractor for the C programming language, since we had access to a large number of repositories for systems written in C. It

would be interesting to implement evolutionary extractors for other programming languages, for example J-REX for the Java language or Cpp-REX for the C++ language. Such extractors could recover the evolutionary history of their software systems. We can then study and compare the evolution history of systems written in different programming languages.

9.2.2 Integrating Source Sticky Notes into Graphical Browsers

Software Exploration tools such as Rigi [Hau88], PBS [FHK+97], and Shrimp [WS00] assist software developers in understanding the structure of their software system. The implementation of Source Sticky Notes presented in this thesis is text based. We believe that integrating these notes with a graphical interface in a software exploration tool would be beneficial. This integration would permit developers to simply right click on an unexpected dependency and a number of relevant Source Sticky Notes would pop up in a floating window.

9.2.3 Better Change Propagation Techniques and More Realistic Evaluations

To demonstrate the feasibility and usefulness of the Development Replay (DR) approach, we presented an example of using it to compare several change propagation tools. We believe that the performance of the presented tools could be improved. Luckily, such improvements could be easily validated using the DR approach.

In our work, we used the concepts of precision and recall to compare the performance of several tools. A high recall would prevent the occurrence of bugs due to missed propagations. A high precision would save the developer's time since she/he will not need to examine incorrect suggestions. It is not clear what is the most appropriate balance that would encourage developers to adopt such tools. Would developers want a tool

that is likely to give a large number of incorrect suggestions (*low precision*) but that is not likely to miss any of the entities that should change (*high recall*)? Or would developers prefer a more conservative tool that would suggest few correct suggestions (*high precision*) but miss other relevant entities (*low recall*)? The answers to such questions are likely to be dependant on the experience of a developer and their knowledge of the software system being changed. Nevertheless, such answers should be derived through case studies and interviews of software developers of varying experience.

9.2.4 Commercial Software Systems

Applying many of the techniques and ideas presented in this thesis on commercial software systems would let us determine if the presented findings and results hold for such systems or if they are specific to open source systems.

The following section provides some insight into new research opportunities that arise from our work.

9.3 Opportunities for Future Research

9.3.1 Grokking Through Time

`Grok` is a relational calculator which has been used for software architecture recovery [BHB99a, HH02]. `Grok` takes as input facts (entities, relations and attributes) about the software system. Using the `Grok` language one can write scripts to manipulate these facts and re-emit them. The `Grok` language is based on Tarski's binary relational algebra. It would be interesting to extend `Grok` to perform analysis across versions by borrowing concepts from temporal logic.

The work presented in this thesis is done using a variety of scripts and programs written in the Perl programming language. Using a Time

extended `Grok` (`TGrok`) language would permit researchers and users of the mined data to express their queries using high level mathematical constructs. More elaborate formal analysis of the mined historical data would be possible.

9.3.2 Visualizing the Recovered Data from Software Repositories

Visualization approaches may reveal interesting patterns about how software systems evolve or change. It would be interesting to develop visualization techniques that could cope with the large amount of historical data. *Evolution Spectrographs* is an example of a visualization technique that has been used to study and explore data recovered using C-REX and other evolutionary extractors [WSHH04, WHH04].

9.3.3 Recovery Of Aspects and Validation of Recovered Aspects

Aspect-oriented techniques aim to improve the handling of crosscutting concerns, within large software systems. The concerns are explicitly captured in well-modularized entities, called aspects. The hope is that such modularization will ease the maintainability and understandability of a software system.

The co-change historical information could be used to to locate parts of the source code that are likely to change together and which represent similar concerns or aspects. This information could be used to propose a restructuring of the source code. The expected improvement due to restructuring could be measured by using the DR approach and the change history of the project.

9.3.4 Change Distance and Design Quality

A good design is likely to have localized changes. We would like to measure the locality of changes. For example, the most local change would require the change of a single code entity (such as a function or a class) to implement a specific feature. If two entities are changed, the closer they are in the software dependency graph, the better the design is likely to be. In short, we would like to determine the minimum number of entities (nodes) and dependencies (edges) needed to connect all the entities changed to implement a specific feature. This aforementioned problem is a well known graph theory problem called the Steiner tree problem — the minimum interconnection problem [HRW92]. The solution to the Steiner problem is unfortunately NP complete but new heuristic algorithm are able to give approximate solutions.

It would be interesting to measure the change distance by approximating the Steiner Tree for the changed entities. We could as well monitor variations to the change distance throughout the lifetime of several software projects using the DR approach.

9.3.5 Discovery of Short Term and Long Term Evolution Patterns

Work by Lehman *et al.* [LRW+97] has led to the development of the laws of software evolution which are long term observations about the evolution patterns followed by software systems. The evolution process is studied using a limited number of data points which represent the releases of a software system. Such aggregation causes the appearance of jumps in the values of the data due to the discrete nature of the data. For example, huge jumps in the LOCs of a system show up from one release to the next, as releases may be a couple of months apart. Furthermore, the time aggregation prevents the discovery of short term (release level) patterns.

We would like to use the detailed information produced by C-REX to discover short term patterns and explain long term patterns. Further-

more, we are interested in discovering evolutionary patterns along a number of characteristics instead of simply focusing on simple characteristics such as LOCs. For example, we could study the performance of change propagation tools over time and between releases. A change propagation tool which is based on historical co-change information is likely to perform well during periods of maintenance but its performance may suffer when new features are being developed.

9.3.6 Evolution of Clones

Using the change data produced by C-REX, we could examine the evolution of clones. In particular, we could study how clones appear and if they ever disappear. We could also examine the history of co-change for clones: Are clones likely to be changed together or in close proximity of each other?

9.3.7 Standardization of Output

We hope that the results shown in this thesis will encourage other researchers to consider adopting historical information to enhance their current techniques and approaches. To permit such adoption, evolutionary extractors need to be available for others to use. Furthermore, tools that make use of information generated by such extractor along with the extractors themselves should use standard formats to ease the exchange of data among tools and researchers.

9.3.8 Development Decision Support (DDS) Appliances

The work presented in this thesis has shown the value of software repositories in assisting practitioners in their activities. Unfortunately, companies are always reluctant to implement procedures and ideas that may assist their development teams in the future. They are more concerned

with the short term impact on their development cycle. Such mentality explains the quick adoption of RAD (Rapid Application Development) tools and hinders the adoption of many research results. In [Has01], we commented that until companies begin planning beyond the next release and adopt more mature development cycles which need better planning and forecasting tools, the adoption of research tools in a commercial setting is likely to be minimal.

Nevertheless, we believe that as the complexity of software systems increases, the need for research tools will become eminent. Moreover, we believe that one way to ease the adoption of results based on mining software repositories is the creation of Development Decision Support (DDS) Appliances. Such appliances are dedicated machines (similar to intranet search appliances) which do not require the intervention of the practitioners to maintain and which are very easy to setup. These appliances are configured to continuously mine all available repositories in a software development organization, and to provide different results and charts that could support practitioners in their activities. Practitioners would consult such appliances using a web browser, therefore requiring no client side software installation.

Based on our survey of practitioners, it seems that many of them already use, in an ad-hoc fashion, information stored in software repositories such as source control and bug repositories. We believe that DDS appliances are likely to be adopted by practitioners over time if they implement the appropriate tools and methods that would assist practitioners in their activities.

9.3.9 Mining Other Repositories and Creating New Repositories

The work presented in this thesis highlighted some of the many benefits of software repositories. We focused on source control repositories as an example of a software repository. Nevertheless, other software repositories such as mailing lists could be explored and different algorithm could

be developed to showcase their benefits as well. Moreover, some of our findings could be used to develop new repositories or enriching current repositories. For example, it would be useful to ask developers to indicate the type of a change (feature addition, bug fix, *etc.*) instead of having to use a lexical approach based on heuristics.

9.3.10 Migrating Source Control Repositories

The historical information about a software system is critical for its future. We have shown that such information could assist developers in understanding its structure and in predicting faults. Unfortunately, most software projects use several source control systems throughout their lifetime. Projects may start with simple source control systems such as CVS, progress to other systems such as Perforce, then eventually adopt very elaborate enterprise level source control systems such as ClearCase. Most projects tend to abandon their old history when moving to a new source control system. We believe that such historical information must be migrated into the new source control system to avoid its loss. Investigating tools and techniques to automate this migration would be valuable and beneficial to practitioners.

9.4 Closing Remarks

The field of software repositories mining is maturing thanks to the rich, extensive, and easily accessible software repositories available from open source projects. We believe the field is likely to take a central role in supporting software development practices and software engineering research.

Our work contributes to the field of software engineering by helping to show that software repositories contain a wealth of useful information that could be easily mined and integrated with several software development practices to assist developers and managers. We hope this work

will encourage academic researchers to explore integrating historical information in their analysis, and will entice practitioners to consider the potential of their repositories which are currently mainly used as static record keeping repositories.

Bibliography

[AA55] Stuart AA. A test for Homogeneity of the Marginal Distributions in a Two-way Classification. *Biometrika*, 42:412–416, 1955. 91

[AB93] R.S. Arnold and S.A. Bohner. Impact analysis - toward a framework for comparison. In *Proceedings of the 13th International Conference on Software Maintenance*, pages 292–301, Montral, Quebec, Canada, 1993. 166

[ABGM99] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the effectiveness of software tools. In *Proceedings of the 21st International Conference on Software Engineering*, pages 324–333, Los Angeles, CA, May 1999. 40, 137

[Ada84] E. Adams. Optimizing preventive service of software products. *IBM Journal for Research and Development*, 28(1):3–14, 1984. 199

[AE70] Maxwell AE. Comparing the Classification of Subjects by Two Independent Judges. *British Journal of Psychiatry*, 116:651–655, 1970. 91

[AEH01] S.K. Abd-El-Hafiz. Entropies as measures of software information. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 110–117, Florence, Italy, 2001. 240

[AL98] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering*, pages 84–93, Kyoto, Japan, Apr 1998. 141

[AP01] Annie I. Anton and Colin Potts. Functional paleontology: System evolution as the user sees it. In *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, May 2001. 22

[AT98] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of the 5th Working Conference on Reverse Engineering*, pages 30–39, Honolulu, HI, October 1998. 35

[BA96] S.A. Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Soc. Press, 1996. 166

[BCLV01] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *Proceedings of the 7th International Software Metrics Symposium*, pages 210–219, 2001. 240

[BD00] B. Bruegge and A. Dutoit. *Object-Oriented Software Engineering*. Prentice Hall, 2000. 129

[Bel77] N. J. Belkin. The problem of matching in information retrieval. In *Theory and Application of Information Research, The Second International Research Forum in Information Science*, pages 187–197, Copenhagen, Netherlands, 1977. 147

[BG02] Robert O. Briggs and Paul Gruenbacher. EasyWinWin: Managing Complexity in Requirements Negotiation with GSS. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, Hawaii, USA, 2002. 207

[BH99] Ivan T. Bowman and Richard C. Holt. Reconstructing Ownership Architectures To Help Understand Software Systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, Pennsylvania, USA, May 1999. 141, 161

[BHB99a] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, USA, May 1999. xx, 107, 108, 112, 250

[BHB99b] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Reconstructing Ownership Architectures To Help Understand Software Systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, USA, May 1999. 207

[BJR00] Lars Bratthall, Enrico Johansson, and Bjorn Regnell. Is a design rationale vital when predicting change impact? a controlled experiment on software architecture evolution. In *Proceedings of the International Conference on Product Focused Software Process Improvement*, Oulu, Finland, 2000. 129

[BKS03] Evelyn J. Barry, Chris F. Kemere, and Sandra A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering*, pages 106–113, Portland, Oregon, May 2003. 239

[BKT03] David Budgen, Barbara Kitchenham, and Scott Tilley. Workshop on "Where's the evidence? The role of empirical practices in software engineering", 2003. Available online at http://www.swen.uwaterloo.ca/~kostas/STEP2003/Workshops/evidence-workshop.htm. 2

[BMS03] Elisa L.A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003. 129, 170

[BMSK02] Elisa L.A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Proceedings of the 1st IEEE International Conference on Aspect-oriented software development*, pages 120–126, Enschede, The Netherlands, April 2002. 158

[BP84] Victor R. Basili and Barry Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42 – 52, 1984. 3, 19

[BP03] Andreas Bauer and Markus Pizka. The contribution of free software to software evolution. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003. 158

[Bro74] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley Professional, 1974. 137, 206, 212, 242

[BS77] Everitt BS. *The Analysis of Contingency Tables*. Chapman and Hall, London, 1977. 91

[CCW+01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 364–374, Florence, Italy, 2001. 3, 19, 67, 72, 74, 127, 129, 171

[Cha95] Ned Chapin. An entropy metric for software maintainability. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 522–523, January 1995. 240

[Cha02] Ned Chapin. Entropy-metric for systems with COTS software. In *Proceedings of the 8th International Software Metrics Symposium*, pages 173–181, 2002. 240

[Cle] Rational ClearCase - Product Overview. Available online at http://www-306.ibm.com/software/awdtools/clearcase/. 72

[CM03] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–419, Portland, Oregon, May 2003. 40, 130, 158, 170

[CMCmWH91] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *International Symposium on Microarchitecture*, pages 69–73, 1991. 189

[CNR90] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990. 42

[Coh60] J. Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurements*, pages 37–46, December 1960. 88

[CPP] CPPX: Open Source C++ Fact Extractor. Available online at http://swag.uwaterloo.ca/~cppx. 42

[CSY+04] Kai Chen, Stephen R. Schach, Liguo Yu, Jeff Offutt, and Gillian Z. Heller. Open-Source Change Logs. *Empirical Software Engineering*, 9(197):210, 2004. 159

[CTA] Exuberant Ctags. Available online at http://ctags.sourceforge.net. 49

[CVSa] CVS - Concurrent Versions System. Available online at http://www.cvshome.org. 72, 192

[CVSb] CVSup Home Page. Available online at http://www.cvsup.org/. 66

[DMK] Inderjit Dhillon, S. Manella, and R. Kumar. Information theoretic feature clustering for text classification. 241

[DP03] Dirk Draheim and Lukasz Pekacki. Process-Centric Analytical Processing of Version Control Data. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003. 37

[DV00] M. Do and M. Vetterli. Texture similarity measurement using kullback-leibler distance on wavelet subbands. In *Proceedings of the 2000 International Conference on Image Processing*, Vancouver, Canada, September 2000. 241

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audris Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001. 3, 186, 189, 239

[ELL+92] Stephen G. Eick, Clive R. Loader, M. David Long, Scott A. Vander Wiel, and Lawrence G. Votta Jr. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*,

pages 59–65, Melbourne, Australia, May 1992. 3, 19, 189, 239

[Ema99] Khaled El Emam. Benchmarking Kappa: Interrater Agreement in Software Process Assessments. *Empirical Software Engineering*, 4(2):113–133, December 1999. 89

[ESEES92] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft–A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992. 3, 171

[FHK+97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997. Available online at `http://www.almaden.ibm.com/journal/sj/364/finnigan.html`. 249

[FN99] N. E. Fention and M. Neill. A Critique Of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. 186

[Fog99] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999. 192

[FPG94] Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 11(4):86–95, 1994. 134

[Ger04a] Daniel M. German. An empirical study of fine-grained software modifications. In *Proceedings of the 20th International Conference on Software Maintenance*, Chicago, USA, September 2004. 37

[Ger04b] Daniel M. German. An empirical study of fine-grained software modifications. In *Proceedings of the 20th Inter-*

national Conference on Software Maintenance, Chicago, USA, September 2004. 64, 67

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the 14th International Conference on Software Maintenance*, Bethesda, Washington D.C., November 1998. 3, 19, 37, 67, 171

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003. 3, 37, 67

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000. 3, 19, 67, 83, 163, 183, 186, 189, 190, 208, 225, 227, 239

[Gla92] R. L. Glass. We have lost our way. *The Journal of Systems and Software*, 18(3):111–112, March 1992. 104

[Gla03a] Robert L. Glass. Questioning the Software Engineering Unquestionables. *IEEE Software*, 20(3):119–120, 2003. 134

[Gla03b] Robert L. Glass. The State of the Practice of Software Engineering. *IEEE Software*, 20(6):20–21, 2003. 2

[GM03] Daniel German and Audris Mockus. Automating the measurement of open source projects. In *Workshop on Open Source Software Engineering (in ICSE03)*, Portland, Oregon, May 2003. 50, 51

[HAK+96] John P. Hudepohl, Stephen J. Aud, Taghi M. Khoshgoftaar, Edward B. Allen, and Jean Mayrand. Emerald: Software Metrics and Models on the Desktop. *Computer*, 13(5), 1996. 3, 19

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier, Amsterdam, Netherlands, 1977. 206

[Har92] W. Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, November 1992. 240

[Has01] Ahmed E. Hassan. Architecture Recovery of Web Applications. Master's thesis, University of Waterloo, 2001. 254

[Hau88] Hausi A. Müller and K. Klashinsky. Rigi – A System for Programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, April 1988. 42, 249

[HG99] James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, pages 85–95, Los Angeles, USA, May 1999. 207

[HGH01] Ahmed E. Hassan, Michael W. Godfrey, and Richard C. Holt. Software Engineering Research in the Bazaar. In *Proceedings of ICSE Workshop on Open Source Software Engineering*, Toronto, Canada, May 2001. 245

[HH] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. Submitted for Publication. 22, 91, 163, 225

[HH00] Ahmed E. Hassan and Richard C. Holt. A Reference Architecture for Web Servers. In *Proceedings of the 7th*

Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, November 2000. 107

[HH02] Ahmed E. Hassan and Richard C. Holt. Architecture Recovery of Web Applications. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002. 33, 250

[HH03a] Ahmed E. Hassan and Richard C. Holt. Adg: Annotated dependency graphs for software understanding. In *Proceedings of VISSOFT 2003: Annual DESIGNFEST On Visualizing Software For Understanding And Analysis*, Amsterdam, Netherlands, September 2003. 51

[HH03b] Ahmed E. Hassan and Richard C. Holt. Studying the chaos of code development. In *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, British Columbia, Canada, November 2003. 51, 73, 74, 75, 91, 208, 221

[HH03c] Ahmed E. Hassan and Richard C. Holt. The Chaos of Software Development. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003. 208, 219, 220, 223

[HH04a] Ahmed E. Hassan and Richard C. Holt. C-REX: An Evolutionary Code Extractor for C. Submitted for Publication, 2004. 121

[HH04b] Ahmed E. Hassan and Richard C. Holt. C-REX: An Evolutionary Code Extractor for C. May 2004. Submitted for Publication. 153

[HH04c] Ahmed E. Hassan and Richard C. Holt. Predicting Change Propagation in Software Systems. In *Proceedings*

of the 20th International Conference on Software Maintenance, Chicago, USA, September 2004. 40, 66, 161, 218

[HH04d] Ahmed E. Hassan and Richard C. Holt. Source Control Change Messages: How are they used? What do they mean? 2004. Draft Available Online. 65, 239

[HH04e] Ahmed E. Hassan and Richard C. Holt. Using Development History Sticky Notes to Understand Software Architecture. In *Proceedings of the 12th International Workshop on Program Comprehension*, Bari, Italy, June 2004. 40, 51, 66, 73, 74

[HRW92] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*. North-Holland (Annals of Discrete Mathematics, Vol 53), 1992. 252

[Hul98] David A. Hull. The TREC-7 filtering track: description and analysis. In Ellen M. Voorhees and Donna K. Harman, editors, *Proceedings of TREC-7, 7th Text Retrieval Conference*, pages 33–56, Gaithersburg, US, 1998. National Institute of Standards and Technology, Gaithersburg, US. 147

[KA98] Taghi M. Khoshgoftaar and Edward B. Allen. Predicting the Order of Fault Prone Modules in Legacy Software. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, November 1998. 186, 202

[KAH+98] Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, Gary P. Trio, and Ronald M. Flass. Using Process History to Predict Software Quality. *Computer*, 31(4), 1998. 3

[KAJH99] Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Data Mining for Predictors

of Software Quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5), 1999. 208

[KBWA94] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, 1994. 207

[KPJ+02] Barbara A. Kitchenham, Shari Lawrence Pfleeger Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002. 2, 134

[KSRP99] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 226–235, Los Angeles, USA, May 1999. 129

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution - Process of Sofware Change*. Academic Press, London, 1985. 240

[Leh80] M. M. Lehman. Programs, life cycles and laws of software evolution. *IEEE Transactions on Software Engineering*, 68:1060–1076, 1980. 240, 242

[Lio99] Lionel C. Briand and Jürgen Wüst and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 475–482, Oxford, England, UK, August 1999. 169

[LPR98] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of Evolution Metrics on Software Maintenance. In *Proceedings of the 14th International Conference on Software Maintenance*, Washington, DC, USA, 1998. 212, 222

[LPS02] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *The Journal of Systems and Software*, 61(3):173–187, 2002. 3, 73, 208, 212, 239

[LRS01] M. M. Lehman, J. F. Ramil, and U. Sandler. An Approach to Modelling Long-Term Growth Trends in Software Systems. In *Proceedings of the 17th International Conference on Software Maintenance*, Florence, Italy, 2001. 69

[LRW+97] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution the nineties view. In *Proceedings of the 4th International Software Metrics Symposium*, Albuquerque, NM, 1997. 36, 37, 68, 206, 240, 252

[LS81] B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981. 104

[LS03] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 102–112, Victoria, British Columbia, Canada, November 2003. 67

[MC04] Jonathan I. Maletic and Michael L. Collard. Supporting Source Code Difference Analysis. In *Proceedings of the 20th International Conference on Software Maintenance*, Chicago, USA, Sept 2004. 68

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(6):308–320, 1976. 191

[MFH00] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 2000. ACM Press. 158

[MH] Tests of Marginal Homogeneity. Available online at `http://ourworld.compuserve.com/homepages/jsuebersax/margin.htm`. 91

[Mic00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 131–142, San Jose, California, October 2000. 36, 37, 68, 240

[Mil56] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 3(81):81–97, 1956. 216

[Mit00] Mark Mitchell. GCC 3.0 State of the Source. In *4th Annual Linux Showcase and Conference*, Atlanta, Georgia, October 2000. 158

[MK92] John Munson and Taghi Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992. 183, 198

[MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998. 35

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, October 1995. ACM. 11, 101, 109, 122

[MV00] Audris Mockus and Lawrence G. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130, San Jose, California, October 2000. 51, 73, 74, 75, 84, 154, 192, 210, 239

[MW47] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, pages 52–54, December 1947. 149, 231

[MWZ03] Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *Proceedings of the 25th International Conference on Software Engineering*, pages 274–284, Portland, Oregon, May 2003. 2, 3, 19, 239

[Nor02] Norman F. Schneidewind. Report on Results of Discriminant Analysis Experiment. In *Proceedings. of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 9–16, December 2002. 207

[OA96] Niclas Ohlsson and Hans Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, dec 1996. 183, 208

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, 1972. 207, 212

[Par94] D.L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279 – 287, Sorrento, Italy, May 1994. 137, 212

[PE85] Dewayne E. Perry and W. Michael Evangelist. An Empirical Study of Software Interface Errors. In *Proceedings of the International Symposium on New Directions in Computing*, pages 32–38, Trondheim, Norway, August 1985. 3, 19

[PE87] Dewayne E. Perry and W. Michael Evangelist. An Empirical Study of Software Interface Faults — An Update. In *Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences*, pages 113–136, Hawaii, USA, January 1987. 3

[Per] Perforce - The Fastest Software Configuration Management System. Available online at http://www.perforce.com. 50, 72, 192

[Per02] Dewayne E. Perry. Laws and principles of evolution. In *Proceedings of the 18th International Conference on Software Maintenance*, page 70, Montreal, Canada, October 2002. 246

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical Studies of Software Engineering: a Roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE) - Future of SE Track*, pages 345–355, Limerick, Ireland, June 2000. 2, 134

[PS93] Dewayne E. Perry and Carol S.Steig. Software Faults in Evolving a Large, Real-Time System: a Case Study'. In *Proceedings of the 4th European Software Engineering Conference*, Garmisch, Germany, September 1993. 3, 73

[Raj97] Václav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the 13th International Conference on Software Maintenance*, pages 84–91, Bari, Italy, 1997. 167

[RM02] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002. 129, 170

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975. 72, 210

[RSA99] H. Richter, Pascal Schuchhard, and Gregory Abowd. Automated capture and retrieval of architectural rationale. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, San Antonio, Texas, USA, Feb 1999. 129

[rsy] rsync Home Page. Available online at `http://samba.org/rsync/`. 66

[SCH98] Susan E. Sim, Charles L. A. Clarke, and Richard C. Holt. Archetypal Source Code Searching: A Survey of Software Developers and Maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 180–187, Ischia, Italy, June 1998. 104, 137

[Sch99] Norman F. Schneidewind. Methodology for Validating Software Metrics. *IEEE Transactions on Software Engineering*, 18(5):410–442, May 1999. 183

[SGEMGK02] Paul Schuster Stephen G. Eick., Audris Mockus, Todd L. Graves, and Alan F. Karr. Visualizing Software Changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002. 3, 171

[SGM+98] Nancy Staudenmayer, Todd Graves, J. Steve Marron, Audris Mockus, Dewayne Perry, Harvey Siy, and Lawrence Votta. Adapting to a new environment: How a legacy software organization copes with volatility and change. In *5th International Product Development Management Conference*, Como, Italy, May 1998. 216

[SGPP04] Kevin A. Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a Software Developers Local Interaction History. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, UK, May 2004. 29

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423,623–656, Jul, Oct 1948. 13, 213

[Shi03] Jelber Sayyad Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003. 19, 29, 40, 141, 171

[Sim98] Susan E. Sim. Supporting Multiple Program Comprehension Strategies During Software Maintenance. Master's thesis, University of Toronto, 1998. Available online at `http://www.cs.utoronto.ca/~simsuz/msc.html`. 104

[Sta84] Thomas A. Standish. An essay on Software Reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984. 104

[Ste03] Steven Klusener and Ralf Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proceedings of the 19th International Conference on Software Maintenance*, Amsterdam, The Netherlands, 2003. 34, 48

[SY99] S. Savari and C. Young. Comparing and combining profiles. In *Second Workshop on Feedback-Directed Optimization (FDO)*, 1999. 240

[TG01] Qiang Tu and Michael W. Godfrey. The Build-Time Software Architecture View . In *Proceedings of the 17th International Conference on Software Maintenance*, pages 398–408, Florence, Italy, 2001. 63

[TG02] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 127–136. IEEE Computer Society Press, June 2002. 32, 36, 37

[TGLH00] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt. Architectural Repair of Open Source Software. In *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000. 128

[Tic85] Walter F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985. 72, 210

[VIM] The VIM (Vi IMproved) Home Page. Available online at `http://www.vim.org`. 218

[vMV94] Anneliese von Mayrhauser and A. Marie Vans. Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering*, pages 39 – 48, Sorrento Italy, May 1994. 107

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, August 1995. 104

[VN96] Michael VanHilst and David Notkin. Decoupling Change from Design. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 58–69, USA, 1996. 207

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979. Available online at http://www.dcs.gla.ac.uk/Keith/Preface.html. 146

[WB99] S. Woods and M. Barbacci. Architectural evaluation of collaborative agent-based systems, 1999. 207

[Wei80] Sanford Weisberg. *Applied Linear Regression*. John Wiley and Sons, 1980. 229

[Wei03] Zachary Weinberg. A Maintenance Programmer's View of GCC. In *First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003. 158

[Wey88] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988. 240

[WHH04] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Exploring Software Evolution Using Spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering*, Delft, Netherlands, November 2004. 251

[WS00] JingWei Wu and Margaret-Ann Storey. A multiperspective software visualization environment. In *Proceedings of the 10th Annual IBM Centers for Advanced Studies Conference*, November 2000. 249

[WSHH04] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. In *Proceedings of the 7th IEEE International Workshop on Principles of Software Evolution*, Kyoto, Japan, September 2004. 251

[Yin94] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Thousand Oaks, CA, 1994. 156, 235

[Yin03] Annie T.T. Ying. Predicting Source Code Changes by Mining Revision History. Master's thesis, University of British Colombia, 2003. 40, 171

[YK03] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 419–429, Portland, Oregon, May 2003. ACM Press. 158

[YNTL88] S.S. Yau, R.A. Nicholl, J.J. Tsai, and S. Liu. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 15(7):58–95, 1988. 140

[YSD98] T. J. Yu, V. Y. Shen, and H. E. Dunsmore. An Analysis of Several Software Defect Models. *IEEE Transactions on Software Engineering*, 14(9):1261 – 1270, sep 1998. 208

[ZDZ03] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003. 37

[Zip49] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949. 162

[ZW04] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, UK, May 2004. 50, 64, 67

[ZWDZ04] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, May 2004. 37, 40, 171