A Design-Rule-Based Constructive Approach To Building Traceable Software

by

Arbi Ghazarian

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2009 by Arbi Ghazarian
Abstract

A Design-Rule-Based Constructive Approach To Building Traceable Software

Arbi Ghazarian

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

The maintenance of large-scale software systems without trace information between development artifacts is a challenging task. This thesis focuses on the problem of supporting software maintenance through a mechanism for establishing traceability relations between the system requirements and its code elements. The core of the proposed solution is a set of design rules that regulates the positional (e.g., package), structural (e.g., class), and behavioral (e.g., method) aspects of the system elements, thus establishing traceability between requirements and code. We identify several types of requirements each of which can be supported by design rules. We introduce a rule-based approach to software construction and demonstrate that such a process can support maintainability through two mechanisms: (a) traceability and (b) reduction of defect rate. We distinguish our work from traditional traceability approaches in that we regard traceability as an intrinsic structural property of software systems. This view of traceability is in contrast to traditional traceability approaches where traceability is achieved extrinsically through creating maps such as the traceability matrices or allocation tables. The approach presented in this thesis has been evaluated through conducting several empirical studies as well as building a proof-of-concept system. The results we obtained demonstrate the effectiveness and usefulness of our approach.
To Ivette, for all her love, support, inspiration, and understanding.
Acknowledgements

I would like to take this opportunity to express my gratitude to all who have helped in the development of this thesis. First and foremost, I would like to thank my supervisor, Professor Dave Wortman, for his extraordinary support in all aspects of my graduate life. His kindness, encouragement, wise advice, knowledge, and professionalism have made working with him a great pleasure and honor. I cannot think of a time when I needed help and he was not there to help me. He generously devoted so much of his time to help me succeed. He passed on to me sound technical knowledge and a positive attitude towards research, education, and academic life that will accompany me wherever I go. I could not have asked for a better adviser to guide me through my doctoral studies.

I am grateful to the other members of my supervisory committee, Professors Steve Easterbrook, Hans-Arno Jacobsen, and Greg Wilson, for their careful reviewing of my work during its several phases. Their suggestions and comments have had a significant impact on this thesis. A large part of this thesis is based on empirical studies. I am indebted to Steve and Greg for teaching me so much about conducting empirical studies in Software Engineering. I thank Arno for his advice on all academic matters and for bringing a new perspective to my work. I am grateful to the external members of my thesis committee, Professors Eric Yu and Eleni Stroulia, for taking the time to read my thesis and provide valuable comments.

I feel extremely fortunate and privileged to have had the opportunity to undertake my doctoral studies at the Department of Computer Science, University of Toronto. I am grateful to the University of Toronto for providing an intellectual and stimulating research environment and for financial support. Special thanks to the administrative staff, Linda Chow, Julie Weedmark, and Joan Allen. They were very helpful and friendly. I will certainly miss my graduate student days here.

I cannot thank enough my family for their tremendous support and encouragement. There is no way I can express enough my gratitude to my parents Aghavni Gharibian...
and Ghazar Ghazarian. My brother, Arin, my grandparents, Angel and Johnny, and all my in-laws were truly supportive. My grandfather’s dream for me was to obtain a high education. I wish he could have seen the result of this work.

Most importantly, my deepest thanks are owed to my wife, Ivette Aghajani, for her infinite patience and truly supportive love throughout the hard times. She has stood by and guided me as I worked and reworked the ideas and arguments put forward in this thesis. She is as responsible for the completion of this thesis as I am. I dedicate this thesis to her.
## Contents

### 1 Introduction to the Problem
- 1.1 The Traceability Problem .................................................. 1
- 1.2 The Negative Impacts of the Traceability Problem on Software Maintenance 2
- 1.3 Problems Involved in Practicing Traceability .......................... 6
  - 1.3.1 Difficulties in Establishing Trace Links ......................... 7
  - 1.3.2 Difficulties in Maintaining Trace Links ......................... 7
  - 1.3.3 The High Costs of Practicing Traceability ..................... 7
- 1.4 The Research Problem and the Proposed Method .................... 9
- 1.5 The Organization of the Thesis ......................................... 9

### 2 Background
- 2.1 What is Requirements Traceability? .................................... 14
- 2.2 Why is Requirements Traceability Important? ....................... 16
- 2.3 Types of Requirements Traceability .................................. 19
  - 2.3.1 Pre and Post Requirements Traceability ....................... 19
  - 2.3.2 Forward and Backward Traceability ............................. 20
  - 2.3.3 Traceability Relations and Semantic Link Types ............. 20
- 2.4 The Mechanics for Tracing Requirements ............................. 23
- 2.5 Trace Analysis .................................................................... 24
- 2.6 Tool Support ....................................................................... 25
2.7 The Open Research Problem ............................................. 27

3 An Analysis of Conventional Approaches to Traceability 28
  3.1 Proactive Vs. After-the-fact Strategies for Capturing Trace Information . 29
  3.2 The Main Characteristics of Conventional Traceability Approaches . . . 34
    3.2.1 Reliance on External Traceability .................................. 35
    3.2.2 Reliance on Capturing Low-Level Trace Information ............... 35
    3.2.3 Separate Capture of Trace Information ............................. 36
  3.3 Conclusion ............................................................... 36

4 A Rule-Based Approach to Traceability 42
  4.1 Source-Code Inconsistencies: Uncoordinated Local Decisions .......... 42
  4.2 Characterizing Design-Rule-Based Traceability ........................ 45
  4.3 A Process for Rule-Based Traceability ............................... 47
  4.4 Examples of Design Rules ............................................. 52
  4.5 Implications of a Rule-Based Approach to Software Construction .... 53
    4.5.1 Traceability of Software Systems ................................. 54
    4.5.2 Reduction in Defect Rate ....................................... 57
  4.6 Issues Needed to Support the Thesis ................................ 57

5 A Proof-Of-Concept System 59
  5.1 Classes of Requirements with a Common Solution ..................... 59
    5.1.1 Input Data Requirement Type ..................................... 61
    5.1.2 Application Rule Requirement Type ............................... 61
    5.1.3 Output Data Requirement Type ................................. 63
    5.1.4 Data Persistence Requirement Type ............................... 64
    5.1.5 User Interface/Interaction Requirement Type .................... 64
    5.1.6 Action Requirement Type ....................................... 64
  5.2 Regulating the Implementation of the Identified Requirements Classes .. 65
6.2.3 Subject Selection ................................................. 93
6.2.4 Data Collection and Analysis ................................. 93
6.2.5 Results .......................................................... 99
6.2.6 Threats to Experimental Validity ............................. 105
  6.2.6.1 Construct Validity ........................................ 105
  6.2.6.2 Internal Validity .......................................... 106
  6.2.6.3 External Validity .......................................... 107
  6.2.6.4 Reliability ................................................ 108
6.3 Summary .......................................................... 108

7 A Case Study of Defect Introduction Mechanisms 112
7.1 Introduction ...................................................... 113
7.2 Case Study ....................................................... 115
  7.2.1 Context of the Case Study .................................. 115
  7.2.2 Description of the Case ..................................... 117
  7.2.3 Case Study Process and Data Collection .................. 118
  7.2.4 Results ....................................................... 120
    7.2.4.1 The Impact of Adopting New, Unfamiliar Technologies
            on Defect Rate .............................................. 121
    7.2.4.2 The Impact of Incomplete Requirements Specifications on
            Defect Rate ................................................ 122
    7.2.4.3 The Impact of the Lack of Requirements Traceability on
            Defect Rate ................................................ 124
    7.2.4.4 The Impact of Not Proactively Defining and Enforcing
            the User Interface Consistency Rules on Defect Rate .... 126
    7.2.4.5 The Impact of Software Size on Defect Rate ........... 127
  7.2.5 Summary of the Results ..................................... 128
  7.2.6 Implications of the Findings ............................... 129
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Requirements definition and management shown as parts of the waterfall model. Figure adopted from [53].</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Pre and Post-traceability. Figure adopted from [53].</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Forward and Backward Traceability. Figure adopted from [53].</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>The Three Requirement-Requirement Traceability Relations. Figure adopted from [53].</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Trace Types and Their Interrelationships. Figure Adopted From [22].</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the Planning and Design Methodology. Figure Adopted From [59].</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Input requirement template contents in PDM. Figure Adopted From [59].</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of a PDM input requirement for a hypothetical communications trace function. Figure Adopted From [59].</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Problem definition template contents in PDM. Figure Adopted From [59].</td>
<td>40</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of a partial PDM problem definition for a hypothetical communications trace function. Figure Adopted From [59].</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Steps for Achieving Rule-Based Traceability</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Requirement-Component Rule for the Input Data Requirement Type</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>Requirement-Component Rule for the Application Rules Requirement Type</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Requirement-Component Rule for the Output Data Requirement Type</td>
<td>68</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.4 Requirement-Component Rule for the Data Persistence Requirement Type</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>5.5 Requirement-Component Rule for the User Interface Requirement Type</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>5.6 Infrastructure Rule 1</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>5.7 Infrastructure Rule 2</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>5.8 Source Code for the AddPaperApplicationRules Java Class</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>6.1 Subject 1’s Performance on Two Different Versions of the Source Code</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>6.2 Subject 2’s Performance on Two Different Versions of the Source Code</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.3 Subject 3’s Performance on Two Different Versions of the Source Code</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.4 Subject 4’s Performance on Two Different Versions of the Source Code</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.5 Subject 5’s Performance on Two Different Versions of the Source Code</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>6.6 Subjects’ Total Performance on Two Different Versions of the Source Code</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction to the Problem

1.1 The Traceability Problem

The Software development process can be viewed as a sequence of model transformations. The earlier models (a.k.a., artifacts or work products) such as the conceptual models, software requirements specifications, use case models, and analysis class diagrams represent the problem to be solved (i.e., the problem space), whereas the subsequent ones such as the architectural documents, design class and sequence diagrams, and the source code represent the solution to the problem (i.e., the solution space). Each phase in the development process uses the models built during the previous phases and produces new models. The goal of this process is to move from a real-world problem toward a software solution (i.e., source code) such that the resulting source code holds certain desired quality properties such as correctness, performance, maintainability, reliability, etc.

Due to the fundamental conceptual gap between the problem space and the solution space, current widely used software development methodologies take an opportunistic approach to transition from the problem space artifacts to the solution space artifacts. For instance, in object-orientation, which is the dominant software development approach in the industry, it is partly unclear how an object-oriented analysis (OOA) relates to
an object-oriented design (OOD) [49]; experience is still a necessary part of this transition. In spite of all the benefits attributed to the object-oriented software development paradigm, it has been unsuccessful in addressing one of the fundamental software engineering problems, which is finding a way to systematically transition from the problem space to the solution space. The guidelines provided by current development methodologies for transitioning from the problem space to the solution space are mostly based on heuristics. These development processes heavily rely on human side of software development, emphasizing factors such as creativity, experience, and individual developers’ knowledge of software design and programming. This high degree of design and implementation opportunism (i.e., a large design and implementation space in the absence of objective development rules or guidelines) leads to software development processes that are unrepeatable, unpredictable, lengthy, and consequently uneconomical.

A major problem with the opportunistic or experienced-based nature of the current software development approaches is that the artifacts created during the development process are not readily traceable to their predecessor and successor artifacts. The lack of trace links between various software artifacts results in a fundamental problem in software engineering, which is known as the Traceability Problem. This problem was identified early in the 1970s, and soon after that numerous tools for tracing software requirements emerged. However, due to the economic and technical difficulties involved, the traceability problem has remained largely unsolved. Inadequate traceability has been identified as a major factor in project over-runs and failures [20] [52].

1.2 The Negative Impacts of the Traceability Problem on Software Maintenance

The traceability problem exhibits its negative effects during the various stages of the software development process, most notably the software maintenance phase. It has been
Chapter 1. Introduction to the Problem

frequently mentioned in the software engineering literature that maintenance costs are the dominant costs of software systems. Studies have shown that change accounts for 40 to 90 percent of the total development costs [7] [28] [5] [19] [15]. When a change is initiated in a software requirements model, it should be propagated into the subsequent models, down to the source code and the test cases that verify the correct implementation of the change. Trace information, if available, can precisely identify the parts of the system (both models and the source code) that are affected by the change. Without trace information, one has to investigate and understand a larger part of the software system in order to perform a change task. What makes the situation even worse is that often the people who maintain a software system are not the same people who developed it. Given the large size and the complexity of today’s software systems, software maintenance, without traceable artifacts, becomes a hard task. This maintenance problem clearly indicates the importance of addressing the traceability problem.

Unfortunately, because of the difficulties involved in capturing trace links between software requirements and the components that satisfy these requirements, traceability is often neglected in practice and therefore maintenance developers are left with the time-consuming and difficult task of investigating the source code of a system in order to identify the subset of the system that is relevant to a change task at hand. In general, a software change task includes three phases [8]:

- Understanding the existing software,
- Modifying the existing software, and
- Revalidating the modified software.

Before applying any changes to a software system, a developer must first identify the parts of the software system that are relevant to the change task at hand. To identify the change set, developers must investigate the system documentation (if available and reliable) and the source code. Without trace information, this system investigation can
be a challenging task to accomplish. The followings are some of the factors that make it hard for developers to locate the subset of the system relevant to a change task, thus increasing the overall difficulty of changing software systems:

- **Size** - size is a major factor in the maintainability of software systems. The larger the size of a system’s source code is, the larger the search space for a change task on that source code will be.

- **Design Complexity** - understanding how a software system works (a.k.a., software comprehension) is a prerequisite to performing a change task on the system. The more complex a system is, the harder it is to understand it. Complex designs make software systems less maintainable.

- **Multi-personal construction of software systems over a long period of time** - large-scale software systems are multi-person projects, and are developed over a long period of time. Therefore, maintaining a large-scale software system translates into being able to understand the thoughts of the software developers who have worked on the project (e.g., design and implementation decisions made by developers) and perhaps are not available anymore. Software developers have different levels of experience and various backgrounds. Moreover, there are no standard methods for performing the various software engineering tasks including requirements engineering, software architecting, and software design. Taking into consideration the radical degree of variation in both developers’ cognitive behaviors, which is reflected in their works, and the tools, techniques, and processes that are used in today’s software development settings, it comes with no surprise that maintaining large-scale software systems is a great challenge.

- **The variety of technologies used in software projects** - as the applications become more complex, more technologies are incorporated into them. These technologies are introduced into the software projects in various forms including third-party li-
braries and Application Programming Interfaces (APIs), application frameworks, middleware software, messaging infrastructures, web-based frameworks, etc. Incorporating these technologies into software systems makes it even harder to understand the system. In addition to understanding the core complexities of the systems, developers need to understand how each of the incorporated technologies works. The fact that these technologies are changing quickly makes the situation even worse.

- **Scattered Implementations** - parts relevant to a change task are often scattered across the system. Thus, in a large system, locating the subset of the system that is relevant to a change task requires an extensive search in a large search space.

- **Poor Documentation** - in the absence of reliable documentation for a software system, the system’s source code becomes the main source of investigation for maintenance purposes. Reliance only on source code makes the performance of maintenance tasks in large-scale software systems more time consuming and mind-intensive.

- **Not conforming to conventions and standards** - implementation inconsistencies in a system’s source code make the comprehension and maintenance of the system harder. Coding conventions and standards, such as the language-specific naming conventions, if strictly followed, can ensure a certain degree of implementation uniformity. This implementation uniformity is particularly important for the maintenance of systems developed by large and distributed project teams, where tens or hundreds of developers, each with their own coding style, are working on the same code base.

- **Coding Style** - the way in which code is written has a significant impact on the understandability and maintainability of software systems. Coding best practices, such as refactoring techniques [29], identify coding styles that are detrimental to the
maintainability of software systems, and provide a more maintainable alternative to writing the code, while preserving its behavior.

- Quality characteristics of the design - design attributes, such as coupling and cohesion, are important factors in the maintainability of software systems. Software modules with a high coupling and low cohesion are not only difficult to understand in isolation, but also difficult to change because change in one module forces a ripple of changes in other modules.

The importance of requirements traceability in managing change (e.g., change impact analysis) has been frequently emphasized in the literature [86] [51] [67] [68] [70] [71] [16]. Of the following six types of trace relations mentioned in [53],

- Requirement-Source,  
- Requirement-Requirement,  
- Requirement-Rationale,  
- Requirement-Component, and  
- Requirement-People,  
- Requirement-Verification

the requirement-component class of trace links is of particular importance to facilitating software change tasks. If this information is available, it can help to precisely identify the parts of the system that are affected by a change. This will reduce the time and the cost of maintaining software systems, hence contributing toward achieving a more economical model of software development. Unfortunately, capturing the requirement-component class of trace links has proved to be difficult, resulting in high costs. The next section discusses some of the difficulties involved in capturing trace links.

1.3 Problems Involved in Practicing Traceability

There are fundamental problems that make the practicing of traceability in software systems a hard task to accomplish. Below, we briefly consider these problems.
1.3.1 Difficulties in Establishing Trace Links

Large systems are composed of thousands of interconnected components that together satisfy a large set of software requirements. Capturing trace links requires a significant effort even for moderately complex systems [72]. While some automation exists, capturing traces remains a largely manual process [23]. The magnitude of work involved in manually finding the relationships between the requirements and their corresponding components is immense. Kotonya and Sommerville [51] state that documenting comprehensive trace information between requirements and the components that satisfy these requirements is impractical because there is too much information to maintain.

1.3.2 Difficulties in Maintaining Trace Links

Trace links once established, should be maintained so as to consistently represent the current state of the relationships between various system process artifacts. Separate evolution of models and systems makes the trace links obsolete, hence they are mistrusted [63]. Artifacts being traced are constantly changing as the system is developed. Therefore, maintaining a traceability scheme becomes a great challenge [92] [82] [81] [15]. Outdated documentation has always been a problem in software development. There is no use in software modeling if the models do not consistently represent the real software system [21]. Establishing traceability involves creating a set of additional documents such as the traceability matrices, which adds up to the burden of keeping software artifacts updated.

1.3.3 The High Costs of Practicing Traceability

The costs associated with practicing traceability in a development organization are grouped into two major categories: (a) the costs of developing and introducing the traceability practices, which are comparable to the costs of developing and introducing any
other significant quality management procedure [79] [51], and (b) the cost of gathering, documenting, and maintaining traceability information [51]. Establishing, maintaining, and validating traceability information for a large-scale software system has been proved to be extremely time-consuming, labor-intensive, and error-prone, resulting in high costs [22] [40].

Ramesh et al. [71] studied the costs and benefits of requirements traceability at the Systems Technology Branch of the Science and Engineering Division at McClellan Air Force Base, a U.S. Department of Defense (DoD) organization. The project they studied involved the redesign and enhancement of an operational flight program for a jet aircraft. The new system contains approximately 75,000 lines of code and over 3,000 requirements. Planning for the increased workload and documentation required to implement requirements traceability, the initial budget for the project planned for twice the normal documentation costs associated with developing a system of that size and complexity. They report that this estimate still fell far short of the actual costs associated with traceability. Moreover, training the various members of the organization in the use of the traceability CASE tool was both time-consuming and expensive. The original system had very little documentation and no detailed traceability. As a result, 10 employees spent over six months of productive work time in order to discover the lower-level requirements, missing requirements, and design rationale. Ultimately, the staff had to rehire engineers from the initial project, at a significant cost, to assist them with the task. The management in this organization accepted these costs viewing them as reducing total life cycle costs due to development of a higher quality product and reduced maintenance costs. The management also believed that the cost related to the use of the traceability CASE tool is most likely a onetime cost that would pay off as the organization continues to practice requirements traceability in its systems development efforts.

While organizations that develop safety-critical systems are more concerned about quality than productivity, and can afford to practice traceability, traceability is not a
feasible option for many software projects where cost minimization is a concern.

1.4 The Research Problem and the Proposed Method

This thesis focuses on the problem of supporting software maintenance through a mechanism for establishing traceability relations between the system requirements and its code elements. The core of the proposed solution is a set of design rules that regulate the positional (e.g., package), structural (e.g., class), and behavioral (e.g., method) aspects of the system elements, thus establishing traceability between requirements and code. We identify several types of requirements each of which can be supported by design rules. We introduce a rule-based approach to software construction and demonstrate that such a process can support maintainability through two mechanisms: (a) traceability and (b) reduction of defect rate.

1.5 The Organization of the Thesis

The goal of this thesis, as a whole, is to develop an approach that will help to reduce the effort spent on corrective changes in software systems. Throughout this thesis, we will pursue two strategies to achieve this goal: (a) achieving effective and affordable traceability (b) promoting defect prevention. Chapters 1 through 6 focus on traceability, while chapters 7 and 8 look at software defects. The thesis is concluded in Chapter 9, where we summarize our contributions and discuss direction for extending this work. The detailed structure and organization of the thesis is as follows:

This chapter introduces the traceability problem. In particular, it considers the negative impacts of the traceability problem on maintaining software systems. It proceeds by emphasizing the importance of the requirement-component class of traceability links in addressing the maintenance problem, and discusses the difficulties involved in practicing traceability. This chapter concludes with an outline of the dissertation and a list of
publications resulted from this thesis work.

Chapter 2 provides the background information necessary to understand the approach proposed in this thesis. It provides an introduction to the subject area of traceability, and presents a detailed discussion of the benefits of traceability, as well as the existing mechanisms for achieving traceability.

Chapter 3 is a critical analysis of current approaches to traceability. In this chapter, traceability approaches, based on their similarities and differences, are classified into two major categories, namely the after-the-fact and proactive approaches. Each category of approaches is exemplified by a number of examples and research works. We proceed by identifying the major common characteristics of conventional approaches to traceability, and discuss how the challenges posed in practicing these approaches in real-life situations are linked to these characteristics. This analysis gives us insights into the desirable qualities that a potential solution to the problem should embody.

In chapter 4, we introduce a process for rule-based software construction, and discuss how such a process leads to effective and affordable traceability. We also discuss some other useful implications of the proposed rule-based process including reduction in the defect rate of software systems. We conclude this chapter by providing an outline of the issues that are needed to validate our approach.

The rule-based software construction process introduced in Chapter 4 is demonstrated through developing a proof-of-concept system, which is presented in Chapter 5. We provide examples to illustrate how the outcome of the process is a traceable software system. We also report on two experiments conducted to verify the feasibility of the process.

To collect evidence on the effectiveness of the approach proposed in this thesis, we conducted a number of empirical studies, which are reported in chapters 6, 7 and 8. In chapter 6, we describe an empirical study that investigates the relationship between the presence of the rules in the source code and the efficiency of developers performing
change subset identification tasks.

Chapters 7 and 8 report on two industrial case studies of defects. Empirical evidence obtained from these two case studies is used to support the argument that the proposed rule-based software construction process can counteract some of the mechanisms that give rise to defects in software systems.

Chapter 9 concludes the thesis, summarizing the contributions of this work. It attempts to place the work of the thesis in perspective, and examine several implications of the results. Some directions for future research are suggested.
The research presented in this thesis has been published in a number of conference and journal papers:


Chapter 2

Background

A basic premise of this thesis is that traceability aids in the construction and maintenance of software systems. In this chapter, we present a detailed discussion of requirements traceability, and review the benefits attributed to traceability in the literature to support this premise. The reader may skip this chapter if they are already familiar with the subject area of traceability. This chapter is organized as follows: in Section 2.1, we discuss the various definitions and interpretations of requirements traceability. In Section 2.2, we emphasize the importance of requirements traceability in software development by reviewing the benefits attributed to requirements traceability in the literature. In Section 2.3, we present the various classifications of requirements traceability. Section 2.4 is a discussion of the various mechanisms for tracing requirements. Section 2.5 is a discussion of the different types of analyses that can be performed on trace information as well as the problems that can be detected as a result of these analyses. In Section 2.6, we present a classification of different types of automated tools for supporting requirements traceability. We conclude this chapter in Section 2.7 by identifying one of the main open research questions in the area of traceability.
2.1 What is Requirements Traceability?

The term ”Requirements Traceability” was first introduced in early 1970s. Since then, it has been defined in the software engineering literature in many different ways. Each definition focuses on some specific aspects of requirements traceability and no single definition encompasses all of its dimensions [37]. Individual researchers and practitioners have their own interpretations of requirements traceability. Gotel and Finkelstein [37] have argued that little shared agreement concerning requirements traceability is one of the major reasons for the persistence of requirements traceability problems. What follows is some of the definitions for requirements traceability that we encountered during our study of the requirements traceability literature.

Traceability is ”a link or definable relationship between entities” [86].

”Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origin, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)” [37].

Requirements traceability is ”the means whereby software producers can ‘prove’ to their clients that: the requirements have been understood; the product will fully comply with the requirements; and the product does not exhibit any unnecessary feature or functionality” [91].

”Traceability refers to the ability of tracing from one entity to another based on given semantic relations” [69].

”Traceability refers to the ability to cross-reference items in the requirements speci-
Traceability is “the ability to follow a specific item at input of a phase of the software life cycle to a specific item at the output of that phase ...” [25].

Traceability enables “each requirement to be traced to its origin in other documents and to the software component(s) satisfying the requirement” [47].

“A requirement is traceable if you can discover who suggested the requirement, why the requirement exists, what requirements are related to it and how it related to other information such as systems designs, implementation and documentation” [51].

Traceability is “the ability of a software to provide a thread from the requirements to the implementation, with respect to the specific development and operational environment” [41].

“An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation” [42].

“Requirements traceability is a characteristics of a system in which the requirements are clearly linked to their sources and to the artifacts created during the system development life cycle based on these requirements” [73].

Requirements engineering and requirements traceability are strongly related to each other. Kotonya and Sommerville [51] have defined requirements engineering as consisting of two parts: requirements definition, which includes requirements elicititation, analysis,
negotiation, documentation, and validation tasks; and requirements management. As depicted in Figure 2.1, while activities related to requirements definition are carried out early in the development process, requirements management activities span the whole life cycle of the system. Requirements traceability is useful in managing requirements and therefore it is considered to be a part of the requirements management component of requirements engineering [53].

Figure 2.1: Requirements definition and management shown as parts of the waterfall model. Figure adopted from [53].

2.2 Why is Requirements Traceability Important?

"Science and engineering alike stress the importance of being able to reproduce results. A general technique for enabling this is being able to trace one’s steps from inception to transition. If done comprehensively, tracing can outline every step along the way of how a problem is transformed into a solution, including the intermediate results and findings. Software development needs traceability for that same reason" [83] [22].

Inadequate traceability has been identified as a major factor in project over-runs and failures [20]. Neglecting traceability or capturing insufficient or unstructured traces leads to a decrease in system quality, causes revisions, and thus, increases project costs and
time [20]. On the other hand, many benefits have been mentioned in the literature for the requirements traceability. These benefits include:

- **Detecting Inconsistencies** - requirements traceability makes it possible to verify that software requirements have been allocated to their corresponding design, code, and tests [86]. Creating explicit links between the work products of the various software development activities such as the requirements specification document, software architecture, detailed design documents, and test cases makes it possible to detect inconsistencies. By inconsistency, we mean that there is an element in the output of a software development phase or activity, such as a document or source code, that does not relate to an element in its predecessor and/or successor phases. Some examples of such inconsistencies include a software requirement in the requirements specification document that is not designed into the software product (i.e., does not have corresponding design components), a design component that does not have a higher-level requirement associated with it (i.e., it is an extra feature not required by the customer), and a software requirement in the requirements specification document that does not have a test case associated with it (i.e., it is not covered by a test case).

- **Accountability** - linking requirements to design, implementation and verification artifacts helps in understanding why and how the system meets the needs of the stakeholders [67] [68] [70] [71]. Trace data can also be used during internal or external audits to prove that a requirement was successfully validated by the associated test cases [86]. These capabilities enhance our confidence to the software product and improves customer satisfaction. Requirements traceability information can also be used for creating subcontracts [71]. Stehle [80] defines traceability from the perspective of managing a system development effort. Traceability can be employed to promote a contractor and contracted method of working. It helps to demonstrate that each requirement has been satisfied [80] [73]. It also helps to avoid gold plating
Chapter 2. Background

(i.e., the addition of expensive and unnecessary features to a system) [91] [73].

- **Change Management** - documenting the links between requirements and other system artifacts helps in requirements change management [86] [51] [67] [68] [70] [71] [16]. Traceability makes it easier to determine related design elements, and consequently the parts of the source code that are affected as a result of a change request. Therefore, it facilitates change impact analysis. Moreover, it helps to identify the tests that should be rerun to verify the correct implementation of the change.

- **Quantitative Traceability Analysis** - quantitative analysis can be performed on trace data. The results of the analysis can be used as a valuable source of information for managing software projects. For instance, this information can be used to measure the progress of the project (e.g., the number or percentage of individual software requirements that have been designed, implemented, and tested), or plan different releases of a software product. It can also be used to more easily approve project milestones and verify the quality of deliverables [86].

- **Requirements Validation and Reuse** - documenting the source and the reason why a requirement was included in the requirements specification document can help in validating and reusing requirements [67] [70] [71]. Requirements traceability can help to reduce requirements errors, which are the largest class of errors typically found in a complex software project [52].

- **Decreasing Dependence on Software Project Team Members** - typically, members of a software project team know a portion of the traceability information that is related to the parts of the system that they have worked on and therefore are familiar with. When these people leave the project, a portion of the trace information, which is undocumented, is lost and consequently makes the maintenance of the system harder. Moreover, lack of traceability information makes it difficult to integrate
new people into a project. [71] [37]. Requirements traceability decreases the loss of important information when people leave projects.

Traceability is also demanded or recommended by numerous standards including ISO 15504 [45], ISO 9001:2000 [46], Software Engineering Institute’s Capability Maturity Model Integration (SEI-CMMI) [95], DoD std 2167A [18], IEEE std 830-1998 [42], IEEE std 1233 [43], IEEE std 1219-1988 [41], and IEEE std 982.2 [44].

2.3 Types of Requirements Traceability

2.3.1 Pre and Post Requirements Traceability

Gotel and Finkelstein [37] have divided requirements traceability into two fundamental types: pre-requirements specification (pre-RS) traceability and post-requirements specification (post-RS) traceability. The former ”is concerned with those aspects of a requirement’s life prior to its inclusion in the RS (requirement production)”, whereas the latter ”is concerned with those aspects of a requirement’s life that result from its inclusion in the RS (requirements deployment)”. Figure 2.2 exhibits pre and post requirements traceability.

Figure 2.2: Pre and Post-traceability. Figure adopted from [53].
2.3.2 Forward and Backward Traceability

Depending on the direction in which traceability links are traversed, requirements traceability can be divided into two types: forward traceability, and backward traceability. In forward traceability, either the origin of a requirement is traced to the requirement, or a requirement is traced to those system process artifacts that have been affected by it. In backward traceability, either system process artifacts are traced to the requirements, or the requirement is traced back to its origin [53]. Figure 2.3 exhibits forward and backward requirements traceability.

Figure 2.3: Forward and Backward Traceability. Figure adopted from [53].

2.3.3 Traceability Relations and Semantic Link Types

In an abstract view, a trace can be considered as an edge connecting two nodes or trace endpoints. Each node represents an entity being traced. With this in mind, it is possible to classify various types of traceability relations based on the types of the nodes. In requirements traceability, one of the nodes is always a requirement. The other node might represent rationale, people, other requirements, components, verification cases, and requirement sources such as the company policies, development environments, stakeholders, documents, and standards. This introduces the six classes of traceability relations.
presented in Table 2.1 [53]:

<table>
<thead>
<tr>
<th>Pre-Requirements Traceability</th>
<th>Post-Requirements Traceability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement-Source</td>
<td>Requirement-Requirement</td>
</tr>
<tr>
<td>Requirement-Rationale</td>
<td>Requirement-Component</td>
</tr>
<tr>
<td>Requirement-People</td>
<td>Requirement-Verification</td>
</tr>
</tbody>
</table>

Table 2.1: The Six Classes of Requirements Traceability

The categories listed under the first column in Table 2.1 pertain to pre-requirements traceability, whereas the three categories under the second column pertain to post-requirements traceability. Each of these categories can be further classified into subcategories based on the type of the edge (i.e., semantics) that connects the two nodes. For example, the link connecting the two requirements in the requirement-requirement traceability category can be of type derives/is-derived, constrains/is-constrained, or requires/is-required. Figure 2.4 depicts the three semantic link types for the requirement-requirement traceability relation. A comprehensive discussion of the semantic link types is provided in [73].

Links of type requirement-component capture the relationships between the requirements and the components that are designed to satisfy those requirements. The process of partitioning requirements into their corresponding components is called requirements allocation, and is an essential part of creating new architectures [53] [67]. Links of type requirement-component are typically documented in allocation tables. The information in the allocation table can be analyzed to ensure that all requirements are implemented in the system. Similarly, a backward analysis of the requirements-component trace information for missing links can reveal any components that do not contribute to the implementation of any requirements. Requirement-component trace information can also be used as a valuable source of information for management tasks such as creating work
breakdown structures, creating project plans, identifying the riskiest (i.e., critical) components, assigning teams to components based on the required skills, and performing change impact analysis. Moreover, this information can be used to measure the progress of the projects by measuring the percentage of the implemented or tested requirements [53].

Users of traceability information, depending on their roles in the development organization, have different perspectives and consequently different needs. An end user may be interested in the answer to the question of what system components are affected by a requirement. A systems designer may, in addition, be interested in the answer to the question of why and how the components are affected by a requirement [73].

Ramesh and Jarke [73] have proposed reference models for the various types of objects and traceability links that can be captured. These models have been presented in two levels of user sophistication: high-end users, and low-end users. Traceability schemes for high-end users are much richer than those of low-end users. Low-end users simply use traceability to link various components of information without explicitly identifying the semantics of such relationships (i.e., traceability links), whereas the high-end models
support a rich set of semantic link types.

2.4 The Mechanics for Tracing Requirements

A combination of the following techniques are typically used to establish traceability:

- *Traceability Matrices/Tables* - a traceability matrix is a rectangular arrangement of trace information, where rows and columns represent traceable elements in two development artifacts. The information recorded in a given cell of the matrix, located at the intersection of row \(i\) and column \(j\), indicates whether or not a trace link exists between element \(i\) from the first artifact and element \(j\) from the second artifact. For instance, a '1' in a cell of the traceability matrix can be taken to denote the existence of a trace link between two elements, and '0' for no trace link between the two element. Traceability matrices are one possible approach for establishing pairwise traceability. Documentation and test matrices are examples of traceability matrices. Traceability matrices make it possible to perform both forward and reverse analyses. They can be used to verify if a relationship exists between elements in two different software artifacts (predecessor-successor and successor-predecessor relationships) [86]. A documentation matrix shows the relationships between individual software requirements and their realizations in lower-level software artifacts such as design components, whereas a test matrix shows the relationships between individual software requirements and the test cases that verify their correct implementation.

- *Unique Identifiers* - traceability mechanisms rely on being able to uniquely identify individual requirements in a requirements set, such as a requirements specification document as well as traceable elements in other software artifacts. This can be accomplished by applying some type of numbering or tagging scheme that enables cross referencing between individual software requirements and related elements in
other software artifacts. Each traceable element is assigned an identifier, which is a unique name or reference number. Computer Aided Software Engineering (CASE) tools that support requirements traceability are usually backed up with a relational database management system where unique identifiers assigned to discrete requirements are used as keys to maintain traces between individual software requirements and other elements in successor or predecessor software artifacts.

- **Attributes** - the term attribute refers to those characteristics that may have several values [53]. Attributes are frequently used in requirements management tools for documenting requirements characteristics such as priority, creation date, version, and status (not implemented, implemented, tested, etc.). The same technique can be used for documenting traceability information. The attribute technique is particularly appropriate for documenting pre-requirements traceability such as the source and rationale information because they allow for documenting long, verbal explanations [53].

- **Lists** - a list can be considered as a table with two columns, each representing one or more elements in a software artifact. For example, each row under the first column might represent a group of software requirements, while the rows under the second column represent a group of components that implement those requirements. In contrast to tables, which are convenient for documenting many-to-many relations, the list technique is best suited for documenting one-to-many traceability relations.

2.5 Trace Analysis

Trace information can be analyzed to detect various types of problems in the system. Some possible analyses include:

- **Forward Requirement-Component Trace Analysis** - a missing link or trace between
a separately identifiable software requirement and one or more design components that satisfy the requirement is an indicator of a missing function in the software product.

- **Reverse/Backward Requirement-Component Trace Analysis** - a missing trace between a design component and a higher-level software requirement is an indicator of an additional feature not required by the software requirements specification.

- **Forward Requirement-Verification Trace Analysis** - a missing trace between a software requirement and one or more test cases that verify the correct implementation of that software requirement indicates that the requirement is not covered by any test cases.

- **Reverse/Backward Requirement-Verification Trace Analysis** - a missing trace between a test case and one or more software requirements that were used to derive the test case indicates that the test case does not verify or cover any part of the software product under development.

### 2.6 Tool Support

Gotel and Finkelstein [37] have classified the various tools that provide automated support for requirements traceability into four major groups:

- General-Purpose tools
- Special-Purpose tools
- Workbenches
- Environments and beyond

General-purpose tools are not specifically designed to support requirements traceability. These tools are usually single-user office tools that can be used to store an
electronic version of otherwise paper-based information. These include hypertext editors, word processors, spreadsheets, and database managements systems. These tools must be hand-configured to provide some sort of support for requirements traceability. The electronic nature of the information stored by these tools makes it easier to search and update the trace information.

The emphasis of special-purpose tools is on supporting single and well-defined activities related to requirements engineering. Some sort of traceability might be achieved as a result of using these tools. For instance, a special-purpose tool for creating test cases might provide traceability between requirements and test cases.

Workbenches are composed of a number of different tools that collectively support a coherent set of activities. For instance, a workbench might consist of various tools for supporting different activities in a requirements engineering work flow such as defining, approving, prioritizing, base lining, and testing requirements. In requirements traceability workbenches, all the constituent tools are designed to ensure requirements traceability. These workbenches provide traceability through supporting the work steps in a specific requirements engineering process.

Environments contain various tools that are integrated to support all aspects of software development. They provide traceability through adherence to the underlying development philosophy. Requirements traceability provided by environments can assist in performing a wide set of life cycle activities. These include maintenance tasks such as change impact analysis, and project management tasks such as progress reporting. Open environments can be integrated with external tools. Consequently, they provide more flexibility in the choice of development processes such as the requirements engineering approach.

Automated support by proprietary or commercial readily available tools is central in some traceability processes. For instance, in his approach, Egyed [22] uses a commercial software monitoring tool to generate trace information when test scenarios are executed.
on the software system. As another example, Hayes et al. [40] have introduced a proprietary prototype tool that uses information retrieval techniques to trace requirements in textual documents on the basis of their similarities.

2.7 The Open Research Problem

Our study of the requirements traceability literature clearly revealed that there is a shared agreement among researchers on the following points:

- Requirements traceability is important to project success.
- Implementing requirements traceability in large-scale systems is extremely time-consuming, error-prone, and costly.

Finding a cost-effective solution for addressing the traceability problem is still an open research problem for the traceability research community.
Chapter 3

An Analysis of Conventional Approaches to Traceability

Over the past decades, various approaches to traceability have been developed. In the first part of this section (Section 3.1), we present a broad classification of these approaches, and discuss example approaches from each class of traceability approaches. In spite of the efforts invested in the research and development of these approaches to traceability, practicing traceability is still not considered a feasible option for most projects in the software industry. Traceability is only practiced in cases where it is required by contractual obligations or in systems such as the safety critical software systems where quality is more important than productivity or cost minimization. In order to understand the root causes of the problems involved in practicing traceability in software systems and to obtain insights into potential solutions to the problem, we performed an analysis\(^1\) of conventional traceability approaches and as a result identified a number of problematic characteristics that were common among these approaches. Section 3.2. provides a discussion of these characteristics. Based on this analysis, we conclude this chapter in Section 3.3 by proposing a set of desirable characteristics that an acceptable solution to

\(^1\)The analysis presented in this chapter has been published in [32].
the traceability problem should possess.

3.1 Proactive Vs. After-the-fact Strategies for Capturing Trace Information

Depending on the strategy taken for capturing trace information, various approaches to traceability can be studied under two general categories: the proactive approaches, and the after-the-fact approaches.

In after-the-fact approaches, the assumption is that the software artifacts and the software system are available, but trace information between them is missing. This means that, traceability, in this category of approaches, is not practiced as part of the normal development process. Instead, it is postponed to the end of the software development process. Establishing, maintaining, and validating trace information for a large-scale software system, at this stage, is extremely time-consuming, labor intensive, and error-prone, resulting in high costs. Therefore, the research in this area is focused on developing processes, models, techniques, and especially automated tools for improving the efficiency of establishing, maintaining, and validating after-the-fact traceability information. Automated support by proprietary or commercial readily available tools is central in some after-the-fact traceability processes. [40] and [22] are prime examples of this line of research.

In [22], Egyed introduced an iterative approach for generating trace information, which is based on observing and recording the internal activities of software systems at the source code level, when test scenarios are executed on the running software systems. In this approach, a commercial software monitoring tool is used for spying into software systems during their execution. Here, the observable run-time information of software systems is used to detect trace links between software systems and their models. In
this approach, first, trace information is generated between the running system and scenarios. Then, the generated trace information is compared with the hypothesized traces. Based on this comparison, new trace information is generated, and the existing ones are validated. This approach is capable of generating and validating four types of trace information:

- Traces between scenarios and system,
- Traces between model elements and system,
- Traces between scenarios and model elements, and
- Traces between model elements

The proposed approach consists of four major activities: hypothesizing, atomizing, generalizing, and refining. The first activity, hypothesizing, is the only manual activity in this process. It involves reasoning about traces that may exist between model elements, scenarios and model elements, and finally, model elements and the system. System documentation and models can be used during the hypothesizing process. Figure 3.1 depicts the trace types and their interrelationships. During the second step of the process, atomizing, a graph-like representation of the observable trace information, called footprint graph, is built. The footprint graph shows the atomic footprints that any two scenarios have in common. In generalizing, the resulting footprint graph is traversed starting at its leaves and trace information is propagated to the parent nodes. In refining, in contrast to generalizing, the footprint graph is traversed starting from the root and trace information is propagated to the leaves. In [22], this approach is illustrated through a proof of concept example of an Inter-Library Loan System (ILL).

As another example of a research work that falls into the "after the fact" requirements traceability category, Hayes et al. [40] have introduced a proprietary prototype tool called RETRO (REquirements TRacing On-target) that uses information retrieval algorithms
such as Tf-Idf vector retrieval [39], vector retrieval with a simple thesaurus [39], and Latent Semantic Indexing (LSI) [17] to determine requirement similarity and automate the tracing of textual requirements.

In contrast to the scenario-driven approach proposed by Egyed [22], which can be used to generate trace links between elements in different models such as data flow diagrams (DFD), class diagrams, scenarios, and the system, the approach proposed by Hayes et al. is specifically designed for generating trace links between textual documents. The rationale behind using information retrieval techniques in the proposed approach is that the problem of tracing requirements is reduced to the problem of determining if each pair of requirements from high and low-level requirements documents are similar. This is very similar to the standard problem of information retrieval which is: given a document collection and a query, determine those documents from the collection that are relevant to the query.

The proposed tool does not attempt to achieve full automation. In fact, the analyst
has critical responsibilities in this process. The tool suggests candidate links, but it is
the analyst’s responsibility to make the final decision on whether those candidate links
should be accepted or rejected.

A second group of researchers have taken a proactive approach to requirements trace-
ability. Here, the idea is that some form of traceability can be achieved as a byproduct of
using certain languages, notations, models, and methods for development [37]. Software
artifacts developed by following development approaches that are more prescriptive or
formal have a higher potential for traceability than those developed through opportunistic
approaches.

The planning and Design Methodology (PDM) [59], and Quality Function Deploy-
ment (QFD) [87] [84] are examples of proactive approaches to traceability. In these ap-
proaches, traceability is considered as an integral part of the development process. Taking
a proactive approach to traceability involves incorporating traceability practices into the
various steps of the development process, making it more formal, and consequently more
heavyweight. In practice, this translates into larger investments (both financial and hu-
man resources), and requires the adoption of major process changes on the development
organization’s side. These barriers to practicing proactive approaches to requirements
traceability have made them an infeasible option for smaller development organizations.

Planning and Design Methodology (PDM) [59] is a requirements planning process
that supports the following activities:

• Requirements collection,

• Definition of the underlying problems,

• Development of an external functional description that addresses the problems, and

• Development of system and designs from the external functional descriptions.

It is a formal requirements process that supports for traceability of the requirements
from initial receipt through implementation. Planning and Design Methodology is shown in Figure 3.2.

![Planning and Design Methodology Diagram](image)

**Figure 3.2:** Overview of the Planning and Design Methodology. Figure Adopted From [59].

In PDM, there is a predefined template for each of the process major work products. These include templates for input requirements, problem definition document, solution definition document, and design direction document. Each document template includes a section for control information. The information recorded in the control section of each document supports for both backwards and forwards traceability. Figure 3.3 shows the content of the information recorded for each input requirement. Figure 3.4 shows a typical input requirement. Figures 3.5 and 3.6 depict the template used for documenting problem definitions and a sample problem definition, respectively.

Quality Function Deployment (QFD) is another product planning technique that facilitates concurrent capture of both problem domain and solution domain knowledge.
without requiring formalism [84]. QFD focuses on the quality of the product under
development solely from the customer’s perspective (as opposed to engineering perspec-
tive). Traceability matrices are used in QFD to capture the relationships between cus-
tomers/users needs (“whats”) and product performance characteristics (“hows”). Insur-
ing the traceability of the design to the requirements is the most crucial aspect of QFD
in delivering high quality products. The planning process consists of the following six
basic activities and is documented in the form of matrices:

- Identify customer’s attributes or requirements,
- Identify technical features (counterpart characteristics) of the requirements,
- Relate the customer’s requirements to technical features,
- Conduct an evaluation of competing products,
- Evaluate technical features and specify a target value for each feature, and
- Determine which technical features to deploy in the remainder of the production
  process.

Finding a cost-effective method for establishing, maintaining, and validating trace
links is still an open research question for both after the fact and proactive lines of
traceability research.

3.2 The Main Characteristics of Conventional Trace-
ability Approaches

Our study of numerous current approaches to traceability observed that regardless of
their reliance on a proactive or after the fact strategy for capturing trace information,
they all share the following three characteristics:
Chapter 3. An Analysis of Conventional Approaches to Traceability

- Reliance on external traceability
- Reliance on capturing low-level trace information
- Separate capture of trace information

In the following sections, we consider each of these characteristics. Understanding the nature and the implications of these characteristics in practicing traceability will help us to better understand the problems involved in practicing conventional approaches. This will provide valuable insights into potential solutions to the problem.

3.2.1 Reliance on External Traceability

An analysis of the nature of the problems reported in the literature for capturing trace information reveals that conventional traceability approaches are based on external traceability (as opposed to internal traceability). The form of traceability offered by these approaches is considered external in the sense that traceability is not an internal characteristic or property of the systems’ design or source code. This means that the design or the source code of the system is not traceable in its own rights. Instead, traceability is achieved outside the design and the source code by creating external maps of the system that attempt to capture traces between traceable entities in the system. It is precisely the inefficiency inherent in the process of discovering existing links between traceable entities in the design or the source code of the system that makes conventional approaches to traceability an infeasible approach in real-life situations where software systems are to incorporate thousands of components to satisfy a large number of requirements.

3.2.2 Reliance on Capturing Low-Level Trace Information

Another characteristic of current approaches to traceability is that they rely on capturing low-level trace information. For instance, these approaches attempt to capture trace links between individual requirements and concrete source code constructs such as classes
or methods that satisfy these requirements. As a result, they have to deal with large amounts of information. This is primarily because in a large scale software system with thousands of requirements and components that satisfy these requirements, there are too many entities that need to be traced. Managing this large volume of trace information poses a great challenge for conventional traceability approaches.

### 3.2.3 Separate Capture of Trace Information

In conventional approaches to traceability, trace information is not generated as a byproduct of following the development process. As a result, trace information needs to be captured separately. In proactive approaches, the additional steps required to capture trace information are integrated into the development process thus making the development process more heavyweight. In after-the-fact approaches, the additional steps required to capture trace information are postponed to the end of the development process at which point it is much harder to capture trace information. The additional steps required for capturing trace information in conventional approaches (both proactive and after-the-fact approaches) put heavy burden on project resources.

### 3.3 Conclusion

Based on our discussion of the problematic characteristics of conventional traceability approaches in the previous section, we suggest that an efficient approach to the traceability problem should address all of the mentioned problems. Therefore, an acceptable solution to the problem should conform to the following properties:

- Reliance on internal traceability
- Reliance on capturing higher-level traceability information
- Generation of trace information as a byproduct of following a certain process
In the following chapter, we propose a process for achieving traceability in software systems that satisfies the conditions mentioned here for an acceptable solution.
Control Information:

- Input requirement control number
- Source reference information, such as document number and location within document
- Submitter information, such as name, address, date received, and submitter’s assessment of the requirement’s priority
- Name, address of person recording the requirement, and the date recorded
- Assigned priority
- Status and disposition information
- Response to the submitter and response date
- Forward reference to the problem definition(s) that address this requirement
- Category and subcategory of the requirement, and associated search keywords. The categories and keywords permit the grouping of similar requirements. The categories for communications management requirements include, for example, problem management, configuration management, change management, operations management, and ease of use.

Abstract of the requirement
The requirement text as originally received

Figure 3.3: Input requirement template contents in PDM. Figure Adopted From [59].
Control Information:

- Input requirement number: PR0102
- Source reference: PASR N18723
- Submitter information: J. Duncan, Branch Office 245, Chicago, IL
- Submitter’s priority: Medium
- Author (person recording): R. G. Mays, Raleigh
- Assigned priority: High
- Status: Promoted to problem analysis
- Response to submitter: Accepted, 2/18/85
- Category/subcategory: Problem management/problem determination
- Search keywords: Trace, operator commands
- Problem definition document: PR1071

Abstract: Allow keyword abbreviations in the trace command.

Requirement text: It is very difficult to key in the long trace commands to start up a communications trace. Customer desires the ability to use one- or two-character abbreviations for the command keywords. For example, L= for LINE=; N= for NODE=; EV= for EVENT=.

Figure 3.4: Example of a PDM input requirement for a hypothetical communications trace function. Figure Adopted From [59].
Control information:

- Problem definition control number
- Author of the problem definition and date written
- Assigned priority
- Status and disposition information
- Category and subcategory of the problem and associated search keywords
- Backward reference to the input requirements which this problem definition addresses and forward reference to the solution definition document(s) which address this problem

Abstract of the problem, with a description of specific cases of the problem, if applicable

Current system function description, including a description of input and output data associated with the system function

User task description and end-user profile

Problem conditions associated with the system function and user tasks, including the impact of the problem condition on the customer and IBM

Environment description

Value assessment

Figure 3.5: Problem definition template contents in PDM. Figure Adopted From [59].
Control information:

- Problem definition number: PR1071
- Category/subcategory: Problem management/problem determination
- Author: R. G. Mays
- Status: Customer review completed
- Input requirements addressed: PR0085, PR0102
- Solution definition document: PR2045

Abstract: The communications trace function is too complex to use and error prone.

Current function: The trace function consists of entering trace commands at the operator console to control the starting and stopping of line traces and formatting and printing of the traces for output.

User tasks: The trace commands are generally entered by the system administrator. The operator does not enter them because the commands are complex, and a significant level of expertise is required to enter them properly. The system programmer generally looks up the command, writes it out, enters the command at the console, and waits for an indication that the desired communications event has occurred. The system programmer then enters commands to format and print the trace.

Problem conditions:

- The trace facility requires too high a skill level. Only experienced system programmers can reliably do communications traces. The impact of this is that customers who do not have an experienced staff are unable to do adequate problem determination.
- Trace commands are frequently keyed in erroneously. Trace commands are long strings of parameters that are subject to keystroke error. If an error is made, the only recourse is to re-key the command, a condition that can be very frustrating.
- There is no on-line help facility. The user must frequently reference the trace manuals and often guesses at actions and command formats.
- The system programmer must guess whether the desired communications event has occurred. The impact of this is that large quantities of excess trace data are often collected, formatted, and printed needlessly.

Figure 3.6: Example of a partial PDM problem definition for a hypothetical communications trace function. Figure Adopted From [59].
Chapter 4

A Rule-Based Approach to Traceability

4.1 Source-Code Inconsistencies: Uncoordinated Local Decisions

In the previous chapter, we explained that one of the problematic characteristics of conventional traceability approaches is that they rely on capturing low-level trace information from software systems. In large software systems, this approach leads to the creation of large volumes of trace links that need to be continuously updated as the system evolves.

What drives the conventional traceability approaches to capture low-level concrete trace information (as opposed to high-level trace relationships) is that most parts of software systems are irregular in the sense that often no stable patterns of traceability relationships exist among the elements within and across the various software artifacts (e.g., models and source code). For instance, the relationships between individual requirements in a requirements specification document and the source-code components that satisfy these requirements are somewhat arbitrary, and is influenced by individual
developers’ experience and style; given a set of software requirements, it is quite likely for two individual developers to come up with different solutions in terms of design characteristics such as the components that participate in the solution, the dependency between the participating components, the distribution of responsibilities over the participating components, the arrangement of the participating components (i.e., the locations of the components within their encompassing system), the names of the participating components, the sequence of messages passed between components, etc.

During the construction of a software system, software developers make a large number of local decisions regarding the design and implementation of the parts of the system they are working on. There is often no explicit mechanism in place to coordinate (i.e., unify) these design and implementation decisions. Given the large possible design space and the large variation in individualistic characteristics of software developers, inconsistencies are introduced in almost all systems. Although existing best practices such as naming conventions and coding standards help to bring some degree of uniformity to software projects, they alone are not enough to prevent the many different forms of inconsistencies that are caused by developers’ individuality.

These inconsistencies, if they exist, prevent us from making useful generalizations about the various dimensions of software systems such as the structure of a class of software components or the traceability relationships between classes of entities in various software artifacts. Since in the presence of inconsistencies we cannot make statements about design properties that hold true for well-defined classes of entities and their relationships within the system’s source-code, we are led to treat each entity or relation on a case-by-case basis (e.g., a concrete trace link between two entities). This means that since there are no observable patterns or rules within and between entities, we must collect and maintain low-level information about entities and the relationships between them.

It is worth noting that inconsistencies can be introduced into the systems, even if
individual developers apply the industry best practices or proven solutions, such as the design patterns and programming idioms, while developing a system. This is because there is often no formal coordination or consensus among developers as to which best practices should be applied to which situations. As a result, even if each individual developer locally applies best practices to the subset of the system they are working on, the total effect can be an inconsistent (i.e., irregular) system.

There is often no technical justification as to why the multiple instances of a particular concern are implemented differently in a system. For instance, our inspection of a number of commercial business applications showed that, among other things, business rules were implemented inconsistently in different parts of these systems. Our further investigations revealed that these inconsistencies were caused by developers’ individualistic styles and a lack of explicit unifying rules or guidelines to coordinate developers in making their design and implementation decisions.

Our position is that the coordination of design and implementation decisions among the members of a development team can be leveraged to derive trace information about software systems. An example will demonstrate this point. Assume that the developers working on a software system consistently and strictly adhere to an architectural convention that requires that for every use case of the system there be a single dedicated component in the system that contains the source code that implements the business rules pertaining to that use case. This convention might further require system developers to place the component in a prescribed location, implement a specific interface, and follow a specific naming convention to make it possible to directly map from a use case of the system to its corresponding business rule component in the source code.

These conventions aim at regulating a dimension of the development problem. In our example, this relates to the implementation of the business rule class of functional requirements. In this system, to make a change to a business rule of a system use case at a later time (i.e., to perform a maintenance task), one does not need to collect trace
information about the business rules and the components that satisfy them. Assuming that the system conforms to the abovementioned architectural conventions, one can easily derive the traceability information needed to perform the maintenance task. That is, the business rules in this system are traceable to their corresponding source code components by construction. An advantage of achieving traceability by construction is that it eliminates the need to gather and maintain hundreds or thousands of low-level pieces of trace information (in our example, traceability links between individual business rules and their corresponding source code components). Instead, all the low-level trace information have been summarized into a design rule, which has been observed throughout the system.

The approach presented in this thesis employs design rules as an enabling mechanism to build traceable systems. A rule-based approach to software construction results in a high-level traceability scheme, which in turn reduces the amount of traceability information that needs to be handled, while providing the benefits of traceability, which we discussed in Chapter 3.

From this discussion, it follows that a possible approach to address the problems associated with conventional traceability approaches is to structure the system such that trace information can be captured at a higher level and in the form of rules describing how identifiable aspects or classes of entities are traced to each other. To do so necessarily requires a rule-based approach to software construction. In the next section, we describe the characteristics of the design-rule-based approach to traceability. In Section 4.3, we will present one such approach.

4.2 Characterizing Design-Rule-Based Traceability

The design-rule-based approach to traceability can be characterized as:

- **Synthetic** - in contrast to analytic approaches to traceability where software
artifacts, such as software requirements specifications and source code, are analyzed to detect, record, validate, and update traceability links between them, design-rule-based approach takes a generative approach to traceability by defining and enforcing design rules that ensure the production of source code that is traceable by its structure, as opposed to traceability by means of external maps such as traceability matrices.

- **Proactive** - in contrast to *after-the-fact* approaches to traceability where establishing traceability is postponed to the end of the development process, design-rule-based approach to traceability is practiced proactively during the construction of the software system. Traceability is achieved as a byproduct of conformance to a set of traceability-related design rules.

- **Constructive** - achieving traceability by construction, which is the central theme in this thesis, is a new approach to traceability in software systems. Existing approaches to traceability are either *cross reference-centered* or *document-centered*. In cross reference-centered techniques, cross references (e.g., unique identifiers) are embedded inside the project artifacts. In more comprehensive traceability schemes, these artifacts are then supplemented by traceability matrices or tables that keep track of cross references. In document-centered techniques, the traceability scheme dictates parts or all of the structure and content of the project documentations to ensure traceability. Whereas the emphasis in existing traceability approaches is on the detection, documentation, and validation of trace links, in the constructive approach to traceability emphasis is placed on the design of software systems to ensure traceability.

- **Just-In-Time** - in design-rule-based traceability, instead of capturing low-level data on the entities being traced, the idea is to capture the high-level generic relationships between them. Once a system’s design, and consequently its source
code, conform to these high-level generic rules, the low-level concrete trace information can be derived from the underlying generic rules when the need arises. For instance, the derivation of the concrete low-level trace information from the high-level rules can be done prior to performing a change impact analysis or a software change task. This means that, instead of capturing trace information beforehand, rule-based traceability provides a *Just-In-Time* approach for computing trace information.

### 4.3 A Process for Rule-Based Traceability

Figure 4.1 outlines the steps necessary to achieve rule-based traceability. In the first two steps of the process, the dimensions or areas of the system that can be regulated along with the rule sets that regulate these areas are identified. These steps are best performed during the architecture phase of the development life cycle, where project standards are defined. However, if needed, new rules can be added and existing rules can be refined throughout the development process in an iterative fashion. Below, we provide guidelines that facilitate these two steps of the process:

- The classes of functional requirements that can be addressed through a common solution, guideline, or framework are possible areas for regulation. The inspection of software requirements specifications in a specific domain can identify the dominating class of requirements in that domain. Each such dominant requirement type represents a potential change type as well and therefore is a candidate area for regulation by one or more design rules. These rules support traceability by ensuring a well-defined mapping between the dominant requirement types and the components that satisfy them. In chapter 5, this approach is illustrated through a proof-of-concept software system, in which the implementation of five classes of functional requirements in the domain of business applications, namely, inputs, out-
1. Identification of areas in software systems that can be regulated (i.e., governed by one or more rules)
   
   (a) Identification of functional areas
   
   (b) Identification of non-functional areas
   
   (c) System decomposition/structure

2. Definition of the rule sets that regulate the identified areas

   (a) Definition of traceability-related design rules as evolution invariants
      
      i. Positional design rules to support package-level traceability
      
      ii. Structural design rules to support class-level traceability
      
      iii. Behavioral design rules to support method-level traceability

   (b) Recording related design rules in the form of traceability rule templates

   (c) Creation of project or domain-specific rule catalogues

3. Consistent application of the rules while building the software systems

4. Validation of the conformance of the system’s source code to the rules

Figure 4.1: Steps for Achieving Rule-Based Traceability

puts, application rules (e.g., business rules and data validations), user interaction, and data persistence requirements have been regulated.

There are both commercial and open source products that aim at bringing uniformity (i.e., regularity) to the implementation of particular classes of functional requirements. Business rule engines, data validation frameworks, and data persistence frameworks are some examples of such products. These products can be incorporated into systems or they can be studied to extract potential rules to regulate the identified functional areas.
The identification of development rules that can help in achieving non-functional or quality requirements is another way to identify areas for regulation within systems. For instance, it is well known that localizing or isolating changes helps in achieving maintainability. The guidelines that lead to the isolation of changes can be formulated as a set of architectural or design rules. As an example, to achieve change isolation, a design rule might be defined and enforced that forbids business logic components from containing SQL code, and requires that all SQL code be placed in dedicated data access components. As another example, a programming rule that requires to check references against the null value before accessing them is known to reduce the number of defects (i.e., null pointer faults) in software systems, hence contributing to achieving the reliability non-functional requirements.

Looking at existing standards and best practices (e.g., architectural patterns, design patterns, and refactoring techniques) can also help to identify candidate areas for regulation. These standards exist at varying abstraction levels from low-level coding conventions to high-level architectural guidelines, and can be generic or product-specific. For instance Java naming conventions are followed by most professional Java programmers, and are meant to increase the readability of the system’s source code by regulating the naming of the programming constructs such as classes, methods, and attributes in the programs written in the Java language. As another example, a layered architectural style regulates the overall structure of a software system by defining rules on how to partition a software system into logically coherent sections. It also defines a set of rules that govern the interactions between the components in various layers of the system (e.g., components at layer \( n \) can only communicate with components at layer \( n + 1 \)). As noted earlier, these conventions alone are not strong enough to build traceable systems. However, they can supplement the creation of stronger forms of rules (e.g., rules that define the relationships or mappings from entities in the problem space to entities in the
solution space) that are required to ensure traceability.

Depending on the needs, trace links from software requirements to source code can be established at different levels of detail. Detailed trace information is more useful in performing maintenance tasks as it provides more accurate information about the parts of the system that are relevant to the maintenance tasks. However, more detailed traceability schemes produce larger amounts of information, which has to be continuously updated as the system evolves. In our rule-based approach to traceability, this is less of an issue since the evolution of the system is constrained by the rules that are defined in the second step of the process. In this approach, rules are evolution invariants.

In object-oriented systems, traceability information are typically established at the levels of packages, classes, and methods. Therefore, in the second step of the process, among all the potential rules that might be part of a solution applicable to an area under regulation, we select those rules that govern the positional (i.e., the packaging structure or the location of the components that participate in the solution), structural (i.e., the components that participate in the solution and their dependencies), and behavioral (what the responsibilities of the components are and how responsibilities of a component are distributed over its methods) aspects of the solution to that area. Positional design properties of a system support package level traceability, whereas structural and behavioral design properties support more detailed traceability at the level of classes and methods, respectively. These are the kind of rules that prescribe how source code is organized into packages, classes, and methods.

In step 2(b) of the process, the set of design rules that are intended to govern an identified area of the system are documented in the form of requirement-component traceability rule templates. In Section 5.2 of Chapter 5, we provide several examples of requirement-component rule templates. As we will show in these examples, where needed, the rule templates can take parameters to specialize the components they specify. The rule templates defined in this step are the main models used during the construction
phase of the development process.

In step 2(c) of the process, a set of such traceability rule templates that are applicable to a specific project or domain are put together to form a rule catalogue. The development of a software project is then driven by its corresponding rule catalogue.

Developers occasionally apply design rules while working on a software project. However, since systematic use of these rules is not formally part of the conventional development processes, their application is ad hoc, individualistic, and therefore inconsistent. The third step in the process addresses this problem by requiring that all project stakeholders agree on and consistently apply the set of rules that are defined for the system under development. This step can be seamlessly integrated with widely-used iterative development approaches such as Rational Unified Process (RUP) and Extreme Programming (XP). In each iteration of the development process, the system’s code base is evolved to incorporate new functionalities or rectify defects in existing functionalities. However, this system evolution is constrained by the rules that are defined during the first two steps of the process. This constrained evolution of the system ensures that certain traceability related design properties hold throughout the system. These design properties along with context information from maintenance tasks are then used to derive low-level traceability information that is required to perform maintenance tasks. In Section 5.2.2 of Chapter 5, we will provide examples to illustrate how trace information are derived from design rules that govern the system.

For the rule-based approach to work effectively, developers must consistently adhere to the rules defined during the process. Step (4) of the process is in place to ensure the conformance of the code to the rules. This can be accomplished as part of code inspections, and can be facilitated through automated rule checker tools.

A limitation of the rule-based approach is that since it is constructive it should be used with new software projects (i.e., applied during the construction of the system). It is possible to refactor a legacy system to make it conform to the rule sets that ensure
traceability. However, depending on the case, the costs of such refactoring might not be justifiable.

4.4 Examples of Design Rules

Table 4.1 shows some typical examples of design rules. These rules unify various kinds of design and implementation decisions about a system, and can be project-neutral (i.e., applicable to all projects) or project-specific. In Chapter 5, we will demonstrate how a set of related rules concerning particular architectural components within a system can be grouped together in the form of component templates. These rule templates along with their corresponding requirement types form requirement-component rules, which drive the development process. Rules 9 and 2 in Table 4.1 are examples of design rules that support package and class level traceability, respectively. Rule 1, on the other hand, supports method level traceability by requiring the implementation of a certain interface that dictates how responsibilities of the implementing class should be divided among the methods defined by that interface. Rule 6 is a supplementary rule that employs a specialized naming convention to facilitate method level traceability. Rules 3 and 5 make use of design patterns to support both class and method level traceability.

There are other types of design rules that do not provide support for traceability. For instance, many of the design rules incorporated in many design patterns help to fulfill non-functional requirements other than traceability. Furthermore, there are other types of rules, such as the compatibility and process rules that can be applied to projects. As we will discuss in the next chapter, these rules can be incorporated into our rule templates as well. However, the focus of our work is on the rules that provide support for traceability.
Table 4.1: Sample Design Rules

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All data access classes should implement the Persistable interface.</td>
</tr>
<tr>
<td>2</td>
<td>All the business rules related to a use case X should be grouped together in a module called XRules.</td>
</tr>
<tr>
<td>3</td>
<td>All model classes (in a Model-View-Controller pattern) should be implemented as instances of the Value Object pattern [1].</td>
</tr>
<tr>
<td>4</td>
<td>All object instantiation in component X should be done through a Factory Method pattern [30], and not constructors.</td>
</tr>
<tr>
<td>5</td>
<td>Each use case of the system should be represented as an instance of the Command design pattern [30].</td>
</tr>
<tr>
<td>6</td>
<td>All data validation methods should be prefixed with &quot;check&quot;.</td>
</tr>
<tr>
<td>7</td>
<td>All access to private class member attributes should be done through accessor methods.</td>
</tr>
<tr>
<td>8</td>
<td>All mutator methods should make a call to the validate() method.</td>
</tr>
<tr>
<td>9</td>
<td>All view files (e.g., JSP and HTML files) should be placed in the WebApp/UI directory.</td>
</tr>
<tr>
<td>10</td>
<td>All implementations of the accept methods in the Visitor design pattern [30] should use a double dispatch protocol.</td>
</tr>
</tbody>
</table>

### 4.5 Implications of a Rule-Based Approach to Software Construction

In the rule-based approach to software construction, the implementation of the system becomes more systematic and less subjective (i.e., less dependent on individual’s style). As more rules are defined, the implementation of more dimensions of the system are regulated and consequently the development of the system becomes more prescriptive and less opportunistic.
The systematic application of design rules leads to a number of important implications in software systems. Below, we briefly discuss two of these implications namely, the traceability of software systems and the reduction in defect rate, which are the focus of this thesis. Through a series of studies, we provide empirical evidence to support these two implications. There are other possible implications to the rule-based approach to software construction, which are discussed in the future work section of this thesis. The design and implementation of empirical studies to validate these other implications is future work.

4.5.1 Traceability of Software Systems

An important implication of the uniformity of design achieved by conforming to such rules in developing a software system is that it provides a built-in mechanism for traceability of software requirements into source code. The collection of requirement-component rules defined for a software project form a framework for building a traceable software system. The framework takes as input the type of the requirement being addressed, and zero or more parameters (e.g. for representing concepts like the use cases or features of the system) and gives, as its output, the specification of the corresponding components that address the requirement. The type of the requirement being addressed determines the specific rules in the collection of rules that must be applied.

In contrast to the conventional approaches to traceability in which trace information must be collected and maintained up front, this rule-based framework makes it possible to compute trace information when the needs arise, providing a just-in-time approach to the traceability of requirements into source code. Section 5.2.2 in Chapter 5 provides examples of maintenance tasks and demonstrates how traceability information required to perform the tasks can be derived from the requirement-component rules observed in the system.

The traceability matrix for the software requirements that have corresponding requirement-
component rules in the system can be derived from the rules. An important property of this approach is that regardless of the growth of the software system in terms of the number of requirements, the number of requirement-component rules remain fairly small and constant. This is because a single requirement-component rule is capable of handling all the requirements that fall under the requirement class that it addresses. This means that the framework makes it possible to apply a small number of requirement-component rules to implement a large number of software requirements.

An advantage of our constructive approach to traceability is that it is a more natural fit for agile software development. Both cross reference-centered and document-centered approaches achieve traceability by creating a more comprehensive set of documentation for software projects. This emphasis on documentation is in sharp contrast with the agile philosophy, where emphasis is placed on the production of source code, and not documentation. Our design-rule-based approach to traceability focuses on the structure of the source code, which is the main artifact in agile development methodologies. In [35] and [34], we have presented a design-rule-based model for traceability in agile software development.

The systematic application of design rules makes software systems traceable without incurring the cost associated of conventional traceability approaches. This implication is the focus of this thesis. Table 4.2 contrasts the cost items associated with the conventional and rule-based traceability approaches. Costs associated with conventional traceability approaches were discussed earlier in Section 1.3 of Chapter 1. The costs of defining and maintaining the rules in the rule-based traceability approach are relatively small compared to the costs of capturing and maintaining traceability links in conventional approaches. Our investigations in the domain of business software systems show that only a small number of requirement-component rules are required to achieve traceability in these systems. Moreover, the rule-based approach to traceability makes it possible to create reusable and domain-specific rule catalogues, which projects can use as a starting
point. Therefore, only the rules that are specific to a particular software project need to be defined. The use of predefined rule catalogues can help to reduce the overall cost of defining the rules. Also, in conventional approaches, the introduction of new requirements to a software system results in an increase in the size of the traceability matrix for that system. In contrast, in the rule-based approach to traceability, new requirements do not necessarily translate into new requirement-component rules; only the introduction of new requirements types can result in the addition of requirement-component rules. The empirical study described in Section 5.3 of Chapter 5 demonstrates the feasibility of training developers to follow the rules. The cost of controlling that developers are actually following the rules can be reduced by means of automated rule checker tools.

Table 4.2: Cost Items in Conventional Vs. Rule-based Traceability Approaches

<table>
<thead>
<tr>
<th>Conventional Traceability</th>
<th>Rule-Based Traceability</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The cost of creating complete correct, and consistent system documentation</td>
<td>• The cost of defining the rules</td>
</tr>
<tr>
<td>• The cost of gathering and documenting traceability information</td>
<td>• The cost of maintaining the rules</td>
</tr>
<tr>
<td>• The cost of maintaining traceability information</td>
<td>• The costs of training developers to follow the rules</td>
</tr>
<tr>
<td>• CASE Tool and training costs</td>
<td>• The cost of controlling that developers actually follow the rules (i.e., enforcing the rules)</td>
</tr>
</tbody>
</table>

Throughout a series of empirical studies, reported in the next chapters, we will demonstrate that a rule-based approach to software construction leads to affordable and effective traceability.
4.5.2 Reduction in Defect Rate

Violation of design and programming rules is the cause of a considerable portion of defects in software systems. This argument is supported by the industrial case studies of defects conducted as part of this thesis. The rules whose observance is known to prevent defects in software systems can be incorporated into the rule sets that drive the development process in our rule-based software construction approach. A reduction in defect rate through observing such rules leads to less corrective maintenance activities in the delivered software, which translates into a lower total development costs for software systems. Therefore, a rule-based approach contributes toward achieving a more economical way of software production.

4.6 Issues Needed to Support the Thesis

The main argument in this thesis is that a rule-based approach to software construction helps to achieve maintainability through the following two mechanisms:

1. Traceability

2. Reduction of defect rate

Below, we provide an outline of the issues that are needed to support this thesis:

1. Developers can be trained to consistently follow the rules.

2. The effect of following the rules is recognizable in the source code (i.e., the effect is real).

3. Traceability can be achieved by conforming to the rules.

4. Violation of the rules can lead to the introduction of defects in software systems.
The rest of this thesis reports on a number of empirical studies that provide evidence to support these issues. To demonstrate how a rule-based approach to software construction leads to effective and affordable traceability, we implemented a proof-of-concept system, which is presented in Chapter 5. Through a series of examples, we illustrate how trace information needed to perform various change tasks can be derived based on the rules deployed in the proof-of-concept system. We also report on two experiments that provide evidence that not only developers can be trained to consistently apply the rules, but also a system that follows a set of rules is recognizable from one that does not.

We further demonstrate the effectiveness of the rule-based traceability scheme in an empirical study of the effect of rules on maintainability, which is presented in Chapter 6. We demonstrate the impact of the rules in the reduction of the defect rate in software systems through two empirical studies of defects in an industrial Enterprise Resource Planning software system. These studies are presented in Chapters 7 and 8.
Chapter 5

A Proof-Of-Concept System

To demonstrate how a rule-based approach to software construction, like the one presented in the previous chapter, can be used to achieve traceability in software systems, we reimplemented an existing conference automation software system called Program Committee Assistant (PCA). PCA, which is used for automating parts of the paper selection process in a typical conference, helps in keeping track of the papers submitted to the conference, as well as the referees who evaluate the submitted papers. The requirements of the system include 13 different use cases, covering a range of activities typically performed in a conference. Table 5.1 shows the list of use cases that make up the PCA software system. The size of the system is appropriate for our purpose of building a proof-of-concept system, since we need to reimplemented the entire system.

5.1 Classes of Requirements with a Common Solution

Our approach to software construction is based on definition and enforcement of unifying design rules that govern the implementation of the various areas within the software system. As mentioned in the previous chapter, one possible way to achieve this is through
Table 5.1: List of Use Cases in the PCA System

<table>
<thead>
<tr>
<th>Use Case ID</th>
<th>Use Case Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add Paper</td>
</tr>
<tr>
<td>2</td>
<td>Add Referee</td>
</tr>
<tr>
<td>3</td>
<td>Delete Paper</td>
</tr>
<tr>
<td>4</td>
<td>Delete Referee</td>
</tr>
<tr>
<td>5</td>
<td>Display Reports on Papers</td>
</tr>
<tr>
<td>6</td>
<td>Display Reports on Referees</td>
</tr>
<tr>
<td>7</td>
<td>Assign Papers to Referee</td>
</tr>
<tr>
<td>8</td>
<td>Unassign Papers from Referee</td>
</tr>
<tr>
<td>9</td>
<td>Evaluate Paper</td>
</tr>
<tr>
<td>10</td>
<td>Accept Paper</td>
</tr>
<tr>
<td>11</td>
<td>Reject Paper</td>
</tr>
<tr>
<td>12</td>
<td>Defer Paper</td>
</tr>
<tr>
<td>13</td>
<td>Quit the System</td>
</tr>
</tbody>
</table>

the classes of functional requirements that can be addressed through a common solution, guideline, or framework. In the proof-of-concept system presented in this chapter, we took this approach to develop the entire PCA software system. We started our development process by assigning a unique identifier to each individual requirement encompassed by each of the system use cases, and labeling each individual requirement by its requirement type. This labeling process effectively classified the software requirements into the following requirements classes:

- Input Data
- Application Rule (e.g., Business Rules and Data Validations)
- Output Data
• Data Persistence

• User Interface (UI)

• Actions

Table 5.2 exhibits a partial list of individual requirements from the Add Paper use case of the PCA software system, which has been classified under these requirements classes. This form of requirements categorization works fine for well known information systems such as business applications. The rationale for choosing these requirements classes was the observation that almost all of the requirements in the system could be clearly classified under these requirements classes. Below, we consider each of these requirements types:

5.1.1 Input Data Requirement Type

Requirements of type input data specify the data items that need to be entered into the software system, without specifying any conditions, constraints or rules on the validity of the entered data. In other words, a requirement of type input data merely specifies a list of data items that are to be entered into the software system. These data items are typically entered into the system by the users of the system (e.g., actors of the use cases). Requirement 1 from Table 5.2 is an example of input data requirement type.

5.1.2 Application Rule Requirement Type

Requirements of type application rule specify the constraints and the rules that must be enforced by the software system to ensure the correctness of the operations supported by the software system. Requirements of this type usually take the form of input data validations, business rules, and calculations. There is often one or more requirements of this type for every data item that is entered into the system by the users (i.e., the
### Table 5.2: Sample Requirements and Their Classes from the Add Paper Use Case

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Add Paper command shall ask the user for the following information: (a) a list of author names, (b) the title of the paper, and (c) the received date.</td>
<td>Input Data</td>
</tr>
<tr>
<td>2</td>
<td>The paper number (passed as a command-line argument to the Add Paper command) must be greater than or equal to 1 and less than or equal to 1200.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>3</td>
<td>The system shall respond to an invalid paper number argument (i.e., does not conform to requirement 2) by showing the following error message: &quot;Invalid Paper Number - Paper number must be in the range 1 and 1200.&quot;</td>
<td>Output Data</td>
</tr>
<tr>
<td>4</td>
<td>Each paper must have at least 1 and at most 15 authors.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>5</td>
<td>The system shall respond to an invalid number of author names (i.e., does not conform to requirement 4) by showing the following error message: &quot;Invalid Number of Authors - Each paper must have at least 1 and at most 15 authors.&quot;</td>
<td>Output Data</td>
</tr>
<tr>
<td>6</td>
<td>The prompt sequence for the Add Paper command should be as follows: (a) Author? (variable, repeats until all author names have been entered) (b) Title? (once), (c) Received Date? (once). The system should repeatedly prompt for the author name until it receives an input line that contains only optional white spaces. This input will signify the end of the variable information.</td>
<td>User Interaction</td>
</tr>
<tr>
<td>7</td>
<td>The system shall record the following information about the paper: (a) one or more author names, (b) the title of the paper, (c) the received date, (d) Current Status = &quot;Received&quot;.</td>
<td>Data Persistence</td>
</tr>
</tbody>
</table>
data items specified by a requirement of type input data). The minimum and maximum allowed lengths of data items, mandatory data items, the correct format of the data items, and the uniqueness of the data items in the system are some typical examples of data validations performed in software systems. Requirement statements that express such data validations are all considered application rules. Some data validations can be performed on data items independently of other data items, whereas other validations on entered data might be meaningful with regards to other entered data items or the data already available in the system.

Business rules, which are certain types of application rules, specify the rules of the domain in which the software system operates. Accounting, warehouse management, order processing, payroll management, banking, insurance, health care, personnel management, sales, and manufacturing are examples of domains of business. Software systems should incorporate the rules governing the operations, business processes and work flows involved in each of these domains. Business rules may also be specific to an organization. Software requirements that specify this type of rules also fall under the category of application rules. Requirement 2 and 4 from Table 5.2 are examples of application rules.

5.1.3 Output Data Requirement Type

Requirements of type output data specify the content and the format of the output (e.g., messages) that is displayed by the software system in response to various situations. Warning messages in response to erroneous data entry, error messages in response to fatal system-level errors, messages that acknowledge the successful completion of operations in a software system, and the output produced as a result of executing operations in software systems are examples of such situations. For every requirement of type application rule, there is often a corresponding requirement of type output data that specifies the message that should be emitted by the system if the application rule is violated. Requirements 3 and 5 from Table 5.2 are examples of output data requirement type.
5.1.4 Data Persistence Requirement Type

Requirements of type data persistence, as the name suggests, specify the data persistence aspects of software systems. It includes requirement statements that specify the details of database operations such as create, read, update, and delete operations on the newly entered or existing data of the system. For every requirement of type input data, there is often a requirement of type data persistence that states that the data items entered into the system by the user of the system, and already validated through requirements of type application rule, must permanently persist in the system. Requirement 7 from Table 5.2 is an example of data persistence requirement type.

5.1.5 User Interface/Interaction Requirement Type

Requirements of type user interface specify the behavior of the command line or graphical interface of the system. These requirements typically specify the look and feel of the system, the navigation between the system screens, and the sequence in which the user of the system interacts with the system. Requirement 6 from Table 5.2 is an example of user interface requirement type.

5.1.6 Action Requirement Type

Requirements of types action specify certain types of processing that involve the communication of the software system with an external agent, device, or another software system as well as data format conversion and data exporting. These requirements specify actions such as sending a confirmation email to a related party, or in the case of a control software system, sending a control signal to an external device connected to the computer through one of its ports to trigger an action in the external device. Generating files such as creating a pdf file containing a certain report based on the data of the system and conversion between various file formats are other examples of action requirements.
Requirement 11 from Table 5.3 is an example of action requirement type.

5.2 Regulating the Implementation of the Identified Requirements Classes

For each class of requirements in the system under development, we devised a set of design rules to address that specific requirements class. As part of the design rules, we also devised appropriate component naming rules in order to maintain a direct relationship between the dimension of the outside-world problem that is being addressed (i.e., classes of requirements) and the source code components that address these requirements classes. We refer to each class of requirements, along with its corresponding design rules as a requirement-component rule, and present them using a uniform rule template. In our rule templates, we use a combination of natural language, pseudo code, and code snippets written in the Java programming language.

Each requirement-component rule maps an identifiable class of development problems (i.e., requirement types or dimensions of the problem space) to the structural, positional, and behavioral design properties of the components that satisfy that specific class of development problem. As we mentioned in the previous chapter, positional, structural, and behavioral design properties support package, class, and method level traceability, respectively. The requirement-component rules specify what components should be created, where the components should be placed, how the components should be named, what the responsibilities of the components are, and how each component should interact with other components. The definition of the requirement-component rules for the system under development corresponds to the second step in the rule-based software construction process described in the previous chapter. Figures 5.1, 5.2, 5.3, 5.4, and 5.5 depict the requirement-component rules, corresponding to the categories in our requirements taxonomic scheme, that we used in developing the PCA software system. In
addition to these five requirement-component rules, we used two infrastructure rules that regulate the architectural aspects of the system. Figures 5.6, 5.7 depict these two rules.

Once the requirements were classified and the requirement-component rules for each category of the requirements were defined, we proceeded to the implementation of the system through building concrete components that conformed to their underlying requirement-component rules. This corresponds to the third step in the rule-based software construction process described in the previous chapter.
Requirement Type: Application Rule

Location

Package: pca.usecase.<X>

Class: <X>ApplicationRules

Where <X> is the use case name under development

Attributes:

/** contains a list of error messages registered by
 * checkArgument(...) or checkRules(...) methods.
 */

(1) private Vector errorList = new Vector();

Methods:

/** checks the validity of the command line arguments
 * passed to the command. An error message is registered
 * in errorList for every detected error.
 */

(1) public Vector checkArgument(String argument)

/** checks the validity of the entered data as the command
 * interacts with the user. An error message is registered
 * in errorList for every detected error.
 */

(2) public Vector checkRules(List input)

/** Extract Method Refactoring
 */

(3) Any number of private helper methods with prefix "check" for
 checking individual rules. These helper methods are called by
 checkArgument(...), checkRules(...), or other helper methods.

Figure 5.2: Requirement-Component Rule for the Application Rules Requirement Type
Chapter 5. A Proof-Of-Concept System

Requirement Type: Output Data

Location

Package: pca.usecase.<X>

Class: <X>OutputHandler

Where X is the use case name under development

Attributes:

(1) Any number of

public static final String <Y> = "<Z>";

Where <Y> is a constant name referring to the actual message text <Z>

Methods:

(1) Zero or more private helper methods with prefix "format" for formatting the output of the command

(2) Zero or more public methods with prefix "show" for displaying the formatted output of the command

Figure 5.3: Requirement-Component Rule for the Output Data Requirement Type
Chapter 5. A Proof-Of-Concept System

Requirement Type: Data Persistence

Location

Package: pca.usecase.<X>

Class: <X>DataAccessObject

Where X is the use case name under development

Attributes:

(1) Zero or more

private String <Y> = "<Z>";

Where <Y> is a constant name referring to the SQL statement <Z>

Methods:

(1) One or more public methods throwing DAOException for performing database-related operations

Figure 5.4: Requirement-Component Rule for the Data Persistence Requirement Type
Requirement Type : User Interface

Location

    Package : pca.usecase.<X>

    Class : <X>InputHandler

    Where X is the use case name under development

Attributes : N/A

Methods :

    /** interacts with the user to collect all the required
     * input data for the execution of the command.
     */

    (1) public static List handleInput()
### Application Root Package: pca

### Application Main Class : PCA under the pca package

### Infrastructure Components:

**Package:** pca.infrastructure

#### Interfaces :

1. **Command interface from the Command Design Pattern.** This interface must be implemented by each use case in the system.

#### Classes :

1. **CommandProcessor**

2. **DBManager**

3. **DAOException**

4. **UtilityMethods**

Figure 5.6: Infrastructure Rule 1.
Use Cases

Location

Package : pca.usecase.<X>

Classes :

(1) Zero or one <X>DataObject class

(2) Zero or one <X>ApplicationRules class

(3) Zero or one <X>InputHandler class

(4) Zero or one <X>OutputHandler class

(5) Zero or one <X>DAO class

(6) One <X>Command class implementing the Command interface

Attributes:

(1) Zero or one private String commandArgument;

Methods :

(1) public constructor for initializing the commandArgument attribute, if the commandArgument attribute exists.

Where <X> is the use case name under development

Figure 5.7: Infrastructure Rule 2.
Chapter 5. A Proof-Of-Concept System

5.2.1 Rule-Based Software Construction: Some Examples

The third step in the rule-based development process described in the previous chapter involves building software components that comply to their corresponding rules. To demonstrate this step of the process, we will discuss the application of the rules defined in the previous section for the PCA software system in developing one of the system use cases namely the Add Paper use case. The application of these seven rules to the set of requirements specifying the Add Paper use case results in the following six Java classes, all of which are located under the pca.usecase.addpaper package:

- AddPaperDataObject
- AddPaperApplicationRules
- AddPaperOutputHandler
- AddPaperDataAccessObject
- AddPaperInputHandler
- AddPaperCommand

Each of these classes contain the source code for handling a well-defined dimension of the system and together form a complete implementation for the Add Paper use case of the system. As an example, since requirements 2 and 4 in Table 5.2 belong to the application rule category of requirements, they were both implemented using the requirement-component rule depicted in Figure 5.2. The application of this rule results in the creation of the AddPaperApplicationRules Java class, which is depicted in Figure 5.8.

Observing a rule such as the one presented in Figure 5.2 throughout the system ensures that, regardless of the differences in the implementation styles of the developers
Figure 5.8: Source Code for the AddPaperApplicationRules Java Class

```java
package pca.usecase.addpaper;
import java.util.*;
import pca.infrastructure.*;
public class AddPaperApplicationRules {
    private Vector errorList = new Vector();
    public Vector checkArgument(String argument){
        int paperId = 0;
        argument = argument.trim();
        if (argument.length() == 0){
            errorList.add(AddPaperOutputHandler.
                WARNING_EMPTY_COMMAND_ARGUMENT);
            return errorList;
        }
        StringTokenizer st = new StringTokenizer(argument);
        if (st.countTokens() != 1) {
            errorList.add(AddPaperOutputHandler.
                WARNING_INVALID_COMMAND_USAGE);
            return errorList;
        }
        try {
            paperId = Integer.parseInt(argument);
            if (paperId <= 0 || paperId > 1200) {
                errorList.add(AddPaperOutputHandler.
                    WARNING_PAPER_NUMBER_NOT_IN_RANGE);
                return errorList;
            }
        } catch(NumberFormatException e){
        }
        return errorList;
    }
}
```
errorList.add(AddPaperOutputHandler.
  WARNING_PAPER_NUMBER_NOT_INTEGER);

  return errorList;
}
AddPaperDAO dao = new AddPaperDAO();
try {
    if ( !dao.isPaperIdUnique(paperId)) {
        errorList.add(AddPaperOutputHandler.
          WARNING_DUPLICATE_PAPER_NUMBER);
    }
} catch(DAOException e) {
    UtilityMethods.printDatabaseException(e);
}
return errorList;

public Vector checkRules(List input) {
    errorList.clear();
    Vector authors = (Vector)input.get(0);
    String title = (String)input.get(1);
    String dateReceived = (String)input.get(2);
    checkAuthorsList(authors);
    checkPaperTitle(title);
    checkReceivedDate(dateReceived);
    return errorList;
}
private void checkAuthorsList(Vector authors) {
    if ( authors.size() == 0) {
        errorList.add(AddPaperOutputHandler.WARNING_EMPTY_AUTHOR);
        return;
    }
if ( authors.size() > 15) {
    errorList.add(AddPaperOutputHandler.
        WARNING_INVALID_NUMBER_OF_AUTHORS);
}

private void checkPaperTitle(String title) {
    if ( title.trim().length() == 0) {
        errorList.add(AddPaperOutputHandler.WARNING_EMPTY_TITLE);
        return;
    }
    if ( title.trim().length() > 150) {
        errorList.add(AddPaperOutputHandler.WARNING_INVALID_TITLE_LENGTH);
    }
}

private void checkReceivedDate(String date){
    date = date.trim();
    if (date.length() == 0) {
        errorList.add(AddPaperOutputHandler.WARNING_EMPTY_RECEIVED_DATE);
        return;
    }
    if ( !UtilityMethods.isDateFormatValid(date) ) {
        errorList.add(AddPaperOutputHandler.WARNING_INVALID_DATE_FORMAT);
    }
}
who implement the system, a dimension of the development problem such as the implementation of the application rules in various use cases of the system are handled uniformly.

In the particular example of the Figure 5.2, the rule corresponds to the application rule category of functional requirements. The parameter \(<X>\) in this rule is replaced with a string constant that represents the use case under development and as a result gives as output trace information such as the package and the component names. For instance, according to the rule depicted in Figure 5.2, in PCA, all the application rules associated with the Add Paper use case are implemented in a class called `AddPaperApplicationRules` located in the `pca.usecase.addpaper` package. Once this class is located, the attributes and the methods sections of the rule provide even more detailed trace information about the contents of the component.

### 5.2.2 Rule-Based Traceability Scheme: Some Examples

In this section, we will demonstrate how the high-level (i.e., rule-based) traceability scheme, achieved as a result of the conformance of the system’s source code to the requirement-component rules, can be used to compute low-level trace information about software systems. In our example, we will use a hypothetical software change task on the PCA software system.

**Example 1** - Consider the following defect report on the "Delete Paper" use case of the PCA software system, which is registered in a defect database by a member of the testing team:

"When the delete paper command is issued to delete a paper from the system, the record representing the paper and the records about the assignment of referees to the paper must be deleted from the database. Currently, the delete paper command only deletes
the paper record, without deleting the related records in the assignment table. As a re-
sult, the records in the assignment table refer to papers that do not exist anymore. This
incorrect state of the database causes the system to produce incorrect reports about pa-
per assignments."

Given that the requirement-component rules shown in figures 5.1 through 5.7 have
been observed in developing the PCA software system, the maintenance developer who
is assigned to fix this defect can follow a line of reasoning like the following: as ev-
ident from the defect report, fixing the reported problem involves making corrective
changes to the data persistence dimension of the delete paper use case. This means that
the requirement-component rule for the data persistence requirement type (Figure 5.4)
should be used to derive the related trace information. The package section of the rule
states that the relevant component is located in the package pca.usecase.<X>, where
X is the use case name under development. Since the change task concerns the Delete
Paper use case of the PCA system, the corresponding component must be located in
the pca.usecase.deletepaper package in the system’s source code. The class section of
the rule gives us the next piece of trace information, which is the relevant component
(i.e., Java class) within the identified package. According to the rule, this component is
called <X>DataAccessObject. In the case of the delete paper use case, this refers to the
DeletePaperDataAccessObject class. The other sections of the rule provide us with even
more information about what we should expect to see inside the component.

Example 2 - Here, we provide another example of how the observance of a requirement-
component rules such as the one shown in Figure 5.2 in a system’s source code facilitates
maintenance tasks. The example scenario involves making a change to the requirement
4 of Table 5.2. According to this change, each submitted paper can now have a maxi-
mum of only 10 authors. Assuming that the system’s source code complies with the rules
shown in figures 5.1 through 5.7, the maintenance developer can follow a line of reasoning
like this: since the changed requirement is an application rule, its implementation in the source code must have complied with the rule expressed in Figure 5.2. According to the package section of this rule, the component implementing this business rule must reside in the \texttt{pca.usecase.addpaper} package. Moreover, the class section of the rule specifies that the component that implements this requirement is named \texttt{AddPaperApplicationRules}. The package name together with the component name is enough for the maintenance developer to uniquely identify the Java class that must be changed. However, the rule is capable of producing even more detailed trace information at the level of methods within the identified component. In the specific example of Figure 5.2, the \texttt{checkArgument()} and \texttt{checkRules()} methods check the validity of the command line arguments passed to the command and the entered data as the command interacts with the user, respectively. These methods register error messages in the \texttt{errorList} attribute, if they detect errors while performing the necessary business rule checking or data validations. The component can also have any number of private helper methods that check individual rules, and are called by the \texttt{checkArgument()}, \texttt{checkRules()}, or other private helper methods. Since the list of author names is not an argument to the command, it must be implemented in a private helper method with prefix \texttt{check}, which is then called by the \texttt{checkRules()} method. Inspecting the \texttt{checkRules()} method (Figure 5.8) reveals the exact method name, which is \texttt{checkAuthorsList()}. As it can be seen from this example, the requirement-component rules make it possible to derive the trace information, including the package, class, and method names, for individual requirements, thus eliminating the need to collect trace information upfront.
5.3 Empirical Evaluation of the Consistency of Applying Rules

For the rule-based software construction approach to work, developers should be able to consistently follow the rules defined for a software project. Following the instructions encapsulated in requirement-component rules involve basic programming tasks such as creating packages, classes, and methods; implementing certain interfaces; inheriting from certain base classes; overriding methods; and the like. These are the kind of tasks that developers are trained to perform during a programming session, and therefore can not become a source of hindrance to taking a rule-based approach to software construction. However, the approach requires the ability to discern which rule, among a collection of rules defined for a system, should be observed to implement a particular dimension of the system under development. In the case of the requirement-component rules, the problem is reduced to determining the type of the requirement being implemented. Since consistency among developers in applying the right rules is essential to the rule-based approach to software construction, an important question is: can developers be trained to consistently determine the type of the requirements they implement? We answered this question through conducting an empirical study. Our study setup was to ask six developers to individually perform an identical requirements classification task on a set of software requirements. The following sections provide the details of the study.

5.3.1 The Task

The requirements used in the study for the requirements classification task were selected form a publicly available system requirements document for a Global Personal Marketplace (GPM) software [27]. The GPM system is a global web-based marketplace bringing together private individuals and small companies to buy and sell all types of items. All users of the system are required to create a user account with the GPM in order to be
able to access the services provided by the system. The requirements for the user registration are documented under the user registration use case, which enables new users to create an account in the system. During the course of this use case, some information about the user is captured. This information is then validated, and a user account is created in the system. Upon successful creation of a user account, a confirmation email is sent to the user. In the case of erroneous data, the system displays appropriate error messages, asking the user to rectify the problems and resubmit the user information. This is a typical use case in many web-based software systems and retains many aspects of a real-world development task such as capturing input data, input validation, business rule checking, producing output, and data storage.

The task included a list of 20 individual functional requirements from the user registration use case. Subjects were asked to determine the category each requirement belonged to. Table 5.3 shows sample requirements along with their associated types from the user registration use case used in our study. Subjects were asked to read a short training material on their tasks before performing them. The purpose of the training document was to make participants familiar with the requirements taxonomic scheme used to classify the requirements. It contained a description of each of the requirements classes (provided in sections 5.1.1 through 5.1.6), along with some examples of each class.

5.3.2 Subject Selection

Subjects for this study were recruited through sending a group email to the graduate students in the Department of Computer Science at the University of Toronto, advertisements in places frequented by Computer Science students, and through personal contacts. Five graduate students and a professional developer from the industry participated in this study. A general experience with software requirements was required for this study. Subjects were paid for their participation.
Table 5.3: Sample Requirements and Their Types from the User Registration Use Case

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The system shall ask the user for the following information: First Name, Last Name, User Identifier, Email Address, Postal Address (including Street Address, City, State, Zip Code), Telephone Number, Password, and Password Confirmation.</td>
<td>Input Data</td>
</tr>
<tr>
<td>2</td>
<td>The &quot;User Identifier&quot; shall have a maximum length of 20 characters.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>3</td>
<td>The system shall respond to an erroneous data entry for the &quot;User Identifier&quot; field that does not conform to requirement 2 by showing the following error message: &quot;Error in Data Entry: The maximum allowed length for the User Identifier is 20 characters&quot;.</td>
<td>Output Data</td>
</tr>
<tr>
<td>4</td>
<td>The &quot;User Identifier&quot; is mandatory. (i.e., cannot be empty.)</td>
<td>Application Rule</td>
</tr>
<tr>
<td>5</td>
<td>The &quot;User Identifier&quot; must be unique in the system.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>6</td>
<td>The &quot;Password&quot; shall have a minimum length of 6 characters.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>7</td>
<td>There must be at least one digit in the &quot;Password&quot;.</td>
<td>Application Rule</td>
</tr>
<tr>
<td>8</td>
<td>The &quot;Password&quot; and the &quot;Password Confirmation&quot; fields must match (i.e., the entered values shall exactly be the same).</td>
<td>Application Rule</td>
</tr>
<tr>
<td>9</td>
<td>The system shall record the following user information: First Name, Last Name, User Identifier, Email Address, Postal Address (including Street Address, City, State, Zip Code), Telephone Number, Password, Account Status = &quot;Active&quot;, Account History = &quot;&quot;, Account Balance = $0.00</td>
<td>Data Persistence</td>
</tr>
<tr>
<td>10</td>
<td>Upon successful creation of the user account, the system shall display the following message: &quot;Thanks for registering with the system. Your account was successfully created.&quot;</td>
<td>Output Data</td>
</tr>
<tr>
<td>11</td>
<td>Upon successful creation of a user account, the system shall send a confirmation email to the user’s email address.</td>
<td>Action</td>
</tr>
</tbody>
</table>
5.3.3 Data Collection

Participants were provided with an answer sheet on which they were to mark the category they perceived to correspond to the type of each requirement presented in the task sheet. The completed answer sheets were collected from subjects at the end of each session.

5.3.4 Results

After collecting the answer sheets, subjects’ responses were compared to the predetermined correct answers. Four out of the six subjects classified all the requirements correctly. The other two subjects each had only one mistake in their classification. On average, subjects classified over 98% of the requirements correctly, which is an acceptable level of consistency among subjects.

5.3.5 Threats to Experimental Validity

Several factors potentially affect the validity of our findings. We discuss these factors under four standard types of validity threats.

5.3.5.1 Construct Validity

The construct validity criterion questions whether the variables measured and studied truly reflect the phenomenon under study. The proportions of correctly classified requirements by individual subjects are the main measure in our study in determining if the subjects can be trained to discern the types of the requirements. This measure reflects the purpose of the study. Since the correct answers were predetermined before conducting the experiment, measuring individual subject’s success level in completing the requirements classification task was simply a matter of comparing each subject’s responses to the correct answers.
5.3.5.2 Internal Validity

The internal validity of a study is concerned with distinguishing true causal relationships from those effected by confounding variables. Not properly handling the training phase of the study could potentially affect our results. To block the potential effects of unequal training, subjects were trained on their tasks only through written material. This ensured that all subjects received equal training. Before conducting the experiments, we ensured that subjects had read and completely understood the training material. Before conducting the experiment, we gave subjects an opportunity to ask any questions they might have had about the training material. In answering the subjects’ questions, we took care not to reveal any information that could give hints to subjects about the correct answers; we only clarified what was already available in the training material.

5.3.5.3 External Validity

The external validity of a study is concerned with the extent to which the findings of the study can be generalized. We used only six subjects in our study. This small number of subjects can be a limit to the generalizability of our findings. We could gain more confidence in our findings by replicating the study with a larger group of subjects.

Five graduate students and a professional developer participated in our study. All of the graduate student participants had enough software development experience in the industry to be considered as professional developers. Therefore, our study was not subject to the difficulties arising from generalizing from students to professionals, which is a concern for many software engineering studies conducted in academic settings.

The set of requirements used in the requirements classification task in our study were typical of those found in most business software systems. However, without further replications of the study using requirements from other types of software systems, we cannot be sure that similar results can be achieved.
5.3.5.4 Reliability

The reliability criterion questions whether the study can be repeated with the same results. Our study has been reported in this chapter. The set of requirements used in this study was derived from a publicly available system requirements document. A subset of the requirements used in the study, along with parts of the training material has been provided in this chapter. The complete experimental materials are available to interested researchers. Therefore, it should be possible to replicate the study.

5.4 Empirical Evaluation of the Recognizability of the Rules

The empirical study described in the previous section demonstrated that developers can be trained to consistently determine requirement types, and therefore they can consistently follow the requirement-component rules during the development process. A peculiarity of the rule-based approach to software construction is that the rules observed in the system can be used to derive trace information (e.g., prior to performing a maintenance task). However, for this to happen, the maintenance developers should be able to detect the rules that have been deployed in the system’s source code during the development process. To assess the practicality of this prerequisite, we conducted an empirical study to see if the effects of taking the rule-based approach to software construction is detectable in the source code. The following sections provide the details of the study.

5.4.1 The Task

Our study setup was to ask five developers to individually perform a source code investigation task on two different, but functionally equivalent, Java implementations of the PCA software system. The existing version of the PCA had been developed using
a conventional development approach (i.e., non-rule-based), whereas the second version of the system complied to the requirement-component rules shown in figures 5.1 through 5.7.

5.4.2 Subject Selection

Subjects for this study were recruited through sending a group email to the graduate students in the Department of Computer Science at the University of Toronto, advertisements in places frequented by Computer Science students, and through personal contacts. Five graduate students participated in this study. Familiarity with the Java programming language was required for this study and subjects were paid for their participation.

5.4.3 Study Process and Results

Each trial of the experiment was carried out in two phases. In the first phase of the study, we familiarized the subjects with the rule-based approach to software construction. The general steps in the process were explained. However, no information was given about the specific rules that had been observed in the rule-based version of the PCA code. In the second phase of the study, subjects were asked to investigate the two program codes and determine the one that was produced using the rule-based process. Subjects used eclipse [94] for this phase of the study. In explaining the process to the subjects in the first phase of the study, we took care not to reveal any information that could help them to determine the rule-based program code.

All subjects, after investigating the two program codes, clearly differentiated the rule-based source code from the non-rule-based one. We interviewed subjects at the end of each experimental session, where we asked subjects to justify their answers. Subjects’ answers indicated that they had detected many of the rules that were deployed in the rule-based system. The experiment confirmed that a source code resulted from following the rule-based process can be clearly recognized from a source code that is produced by
other processes, and that the detectability of the deployed rules is a valid assumption.

5.4.4 Threats to Experimental Validity

The following sections provide a discussion of the factors that can potentially affect the validity of our findings.

5.4.4.1 Construct Validity

Subject’s ability to discern the rule-based program code from the program code that was produced by a non-rule-based process, as well as their ability to detect the rules deployed in the rule-based version of the program were the main measures in studying the recognizability of the effect of the rule-based process. These measures could be evaluated objectively, and reflect the purpose of the study.

5.4.4.2 Internal Validity

Unequal training could potentially be a confounding factor. We ensured that all subjects received the same training during the first phase of the experiment, and that subjects had a clear understanding of the rule-based development process. To ensure that subjects’ answers were based on solid evidence taken from the source code rather than guess work, subject were interviewed. As part of the interview, we asked subjects to justify their answers based on the evidence they had found in the source code.

5.4.4.3 External Validity

We used only five subjects in our study. This small number of subjects can be a limit to the generalizability of our findings. We could gain more confidence in our findings by replicating the study with a larger group of subjects.

Five graduate students participated in our study. All of the graduate student participants had enough software development experience in the industry to be considered as
professional developers. Therefore, our study was not subject to the difficulties arising from generalizing from students to professionals, which is a concern for most software engineering studies conducted in academic settings.

5.4.4.4 Reliability

Our study has been reported in this chapter. The complete experimental materials including the two program codes are available to interested researchers. Therefore, it should be possible to replicate the study.

5.5 Summary

This chapter demonstrated the rule-based traceability approach, which was described in Chapter 4, through a proof-of-concept system. The dominating requirement types were identified and requirement-component rules were defined to regulate the implementation of the identified requirements types. Sample code was provided to demonstrate how the implementation of the system is guided by the requirement-component rules. We further provided examples to show how trace information required to perform maintenance tasks can be derived from the rules observed in the system. The two empirical studies described in this chapter demonstrated that the assumptions that developers are able to consistently follow the rules and that the rules are detectable in the code are valid thus increasing the practicality of the rule-based traceability approach.
Chapter 6

An Empirical Study of Rule-Based Traceability and Maintainability

6.1 Introduction

Software comprehension is a prerequisite to software maintenance. Before attempting to change a software system due to a change in an underlying requirement, one must first understand how the original requirement was implemented in the software system. This is reduced to the problem of change subset identification. Change subset identification is the activity through which the parts of a software system that are affected by a change request are identified. Once the parts relevant to a change task are identified, they can be analyzed (i.e., change impact analysis), changed, and tested to accomplish the maintenance task (i.e., to accommodate the changed requirement).

Due to the high costs associated with creating and maintaining correct, complete, and consistent documentation for large-scale software systems, it is common for software systems to lack reliable documentation. As a result, the system’s source code is often the main source of investigation for maintenance purposes. Parts relevant to a change task are often scattered throughout the source code, which make it difficult to locate them.
Maintenance developers of large-scale software systems often face the time-consuming task of investigating the system’s source code in order to locate the subset of the source code which is relevant to the change task at hand.

As discussed in Chapter 1, creating and maintaining traceability tables, which are the conventional solution to this problem, incur high costs. In the previous chapters, we introduced a rule-based traceability scheme as a more affordable alternative to conventional traceability approaches, and provided examples of how such a scheme can be used to derive the trace information required to facilitate software change tasks. In this chapter, we put the rule-based traceability scheme into practice in an experimental setting. To collect evidence on the effectiveness of the rule-based traceability scheme in change subset identification and consequently the maintenance of software systems, we conducted an empirical study of five developers performing various software change identification tasks on the non-rule-based (source code A) versus rule-based versions of the PCA software system (source code B) described in the previous chapter. The remaining of this chapter reports on this empirical study.

6.2 The Study

Our study is set up to have five developers to individually perform three change subset identification tasks on two functionally equivalent programs. The requirement-component traceability rules were deployed in only one of the programs. The main purpose of the study was to verify whether the presence of the traceability rules in a system’s source code affect the way developers perform maintenance tasks. We performed both quantitative and qualitative analyses of the collected data. The quantitative analysis was used to verify whether the difference in subjects time performance on the two programs is significant, whereas the qualitative analysis was used to verify that the difference in time performance is actually caused by the traceability rules and not other factors.
6.2.1 Experimental Procedure

The study was divided into four phases. In the first phase of the study, subjects were familiarized with the target software system. This was accomplished by showing a demo of the running system, as well as explaining the use cases of the system. In the second phase of the study, subjects were asked to investigate the source code A and identify the parts of the system that were relevant to each of the change tasks in a set of three change tasks. One change task was presented to the subjects at a time. Each of the three change tasks was representative of a type of typical change request in most development efforts; the first was a defect fix in the system, the second was a request to enhance the functionality of the system, and the third was a change to the existing functionality of the system. In the third phase of the study, subjects were trained with the rules that were used in the rule-based version of the program code. In the fourth phase of the study, subjects were asked to investigate the source code B and identify the parts of the system that were relevant to each of the three change tasks. As in phase two, one change task was presented to the subjects at a time.

In designing the experiment, we used a within-subject design on five subjects to investigate how compliance to the traceability rules impacts subjects’ performances on their tasks compared to the control (non-rule-based version of the program code). An advantage of a within-subject experimental design such as ours is that confounding variables due to differences in participant skills are reduced [12].

As recommended in [50] and [12], to avoid learning bias, the order in which the two programs were presented to the subjects were counterbalanced. In our study design, subjects 1, 2, and 5 started the experiment with source code A, while subjects 3 and 4 first worked on source code B.
6.2.2 Tasks

The change subset identification tasks we requested of subjects consisted of the following changes. The first is corrective [56], while the other two are perfective [56].

1. When the Delete Paper command is issued, the record representing the paper and the records representing the assignment of the referees to the paper must be deleted from the system. Currently, the Delete Paper command only deletes the paper record, without deleting the related assignment records. As a result, the assignment records refer to papers that do not exist anymore. This incorrect state of the system causes it to produce incorrect reports about paper assignments.

2. Currently, the Add Paper command only accepts dates of YYYY-MM-DD format for the received date of the paper. The program should be extended to support a second date format of DD-MM-YYYY.

3. The Add Paper command checks the paper identifier number for a new paper to ensure uniqueness. If the paper number already exists in the system, a warning message is displayed to the user. Currently, the warning message is not clear enough. The command should be modified to display a more clear warning message.

To further reduce the learning bias, the three change tasks were selected to be independent of each other. The tasks were carefully selected to make it possible to objectively evaluate the correctness of the change subsets identified by the subjects. We performed a detailed analysis of both programs to determine the subsets relevant to each of the change tasks. Since the correct subsets relevant to the changes could be determined objectively, establishing whether a subject completed a task successfully was a matter of comparing the subsets identified by the subject with the correct subsets.
6.2.3 Subject Selection

Subjects for this study were recruited through sending a group email to the graduate students in the Department of Computer Science at the University of Toronto, advertisements in places frequented by Computer Science students, and through personal contacts. Four graduate students and an undergraduate student participated in this study. Familiarity with the Java programming language and previous experience of software development were required for this study. Subjects were paid for their participation.

6.2.4 Data Collection and Analysis

In our study, we used both the *Think-Aloud Protocol* and *Shadowing* data collection methods as explained in [55]. We encouraged participants to think out loud while investigating the source code to identify the subset of the program relevant to the change task at hand. To ensure that no information is lost, shadowing was done using a screen and voice recording software tool. We collected four types of data from the subjects during the study:

- the subsets of the program the subjects identified as relevant to the change task at hand,
- the subjects’ time to completion for each task,
- a video record of the subjects’ computer screens while performing the source code investigation tasks, and
- an audio record of the subjects’ voices as they were verbalizing their thoughts during the source code investigation tasks.

The video and audio records were captured using the Hypercam [93] screen and audio recording program. As advised in [55], in addition to these data collection techniques, we
discussed the findings of the study with the participants to see if they believe an accurate portrayal of their situation has been achieved.

Table 6.1 presents the time taken (in minutes and seconds) by each subject to complete their tasks on source codes A and B, as well as the total time taken for completing the set of three tasks. The bar charts in figures 6.1, 6.2, 6.3, 6.4, and 6.5 in the Appendix at the end of this chapter depict the difference in each subject’s performance in terms of the time required to complete each of the tasks on the two different versions of the program. Figure 6.6 depicts the total time taken by each subject to complete the set of three change tasks on the two different versions of the program.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A</th>
<th>Source Code B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Task A1</td>
<td>Task A2</td>
</tr>
<tr>
<td>1</td>
<td>02:13</td>
<td>02:24</td>
</tr>
<tr>
<td>2</td>
<td>04:19</td>
<td>04:15</td>
</tr>
<tr>
<td>3</td>
<td>02:10</td>
<td>07:25</td>
</tr>
<tr>
<td>4</td>
<td>00:59</td>
<td>02:49</td>
</tr>
<tr>
<td>5</td>
<td>03:55</td>
<td>02:37</td>
</tr>
</tbody>
</table>

Table 6.1: Time Taken to Complete the Tasks

The steps we took to analyze the collected data qualitatively is inspired by [74] and [77]. We followed a two step process: first we closely examined the data to identify patterns, contrasts, and commonalities in the way subjects performed their tasks on the two programs. In the second step, we derived observations based on the identified patterns, contrasts, and commonalities. To perform a qualitative analysis of the data, we needed an abstraction of the subjects’ actions and thoughts while performing their tasks. For this reason, we transcribed the audio and video data captured from each of the subjects while performing their tasks. The data transcription method used in our study is inspired by [74]. We transcribed each subject’s actions and thoughts into a series
of source code navigation events. For each event, we recorded the following information:

- The time of the event
- The program element (i.e., class, method, or attribute) selected or examined by the subject at the time of the event
- The navigation method used to access the program element
- Comments on each event, where necessary, mostly based on the captured audio. Some comments explain the reasoning behind why the subject decided to examine a specific program element.

Since subjects used Eclipse [94] for source code investigation during the study, the following navigation methods were considered:

- Package explorer - the subject reveals a class or method by selecting it in the Eclipse package explorer view.
- Class outline pane - the subject reveals a method by selecting it in Eclipse class outline pane.
- Scrolling - the subject reveals a method by scrolling up or down in a source file.
- Cross-reference - the subject reveals a method by following a cross-reference in the code (i.e., following a method call).
- Recall - the subject reveals a class or method by returning to an already open editor window.
- Keyword search - the subject reveals a method by performing a keyword search (i.e., a lexical search).
As an example, Table 6.2 shows an excerpt from a transcript from one of the subjects. Note that, for simplicity, we have used codes instead of class and method names. Uppercase letters refer to classes, whereas a lowercase letter followed by a subscript index number refers to a method inside the class. As the transcript excerpt in Table 6.2 shows, at time 00:06 the subject accesses the class A by selecting it in the package explorer. The reason for deciding to examine this class is that based on the similarity of the class name to the use case encompassing the change task, the subject has guessed that this class should be relevant to the change task at hand. Then, at time 00:14 the subject scrolls to method \( a_1 \) within the class A. While examining \( a_1 \), the subject notices a call to method \( b_1 \) inside the method \( a_1 \) (i.e., cross-reference) and decides to examine \( b_1 \). To do so, the subject selects the class B in the package explorer. Since there are several methods in class B, the subject decides to use the IDE’s search facility to find the method rather than manually looking for the method. Accordingly, at time 01:23, she returns back to class A, whose editor window is already open (i.e., recall) to copy the name of the method \( b_1 \). After copying the method name, she returns back to the class B’s editor window (i.e., recall), and pastes the class name in the search dialog box and finds the method \( b_1 \) (i.e., keyword search).

To make sense of the transcribed data, we created useful views of the transcribed data. For instance, tables 6.3, 6.4, and 6.5 show the source code navigation path taken by each of the subjects to perform their tasks on each of the two source codes. The source code navigation path for performing a change subset identification task is the sequence of methods examined by a developer to identify the parts of the system relevant to the change task at hand. The number of methods in the path indicates the length of the path. These views helped us to derive interesting observations (commonalities, differences, and patterns) about the ways the subjects had navigated the two program codes.

Another useful view is the one that compares the time and the number of methods investigated by each of the subjects to complete the tasks on each of the two source
Table 6.2: Transcript Excerpt

<table>
<thead>
<tr>
<th>Time</th>
<th>Subject</th>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:06</td>
<td>A</td>
<td>Package Explorer</td>
<td>Based on the class name</td>
</tr>
<tr>
<td>00:14</td>
<td>$a_1$</td>
<td>Scrolling</td>
<td>noticed a call to method $b_1$ while investigating method $a_1$</td>
</tr>
<tr>
<td>01:13</td>
<td>B</td>
<td>Package Explorer</td>
<td>Cross-reference</td>
</tr>
<tr>
<td>01:23</td>
<td>A</td>
<td>Recall</td>
<td></td>
</tr>
<tr>
<td>01:37</td>
<td>B</td>
<td>Recall</td>
<td></td>
</tr>
<tr>
<td>01:38</td>
<td>$b_1$</td>
<td>Keyword Search</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Source Code Navigation Paths for Task 1 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 1</th>
<th>Source Code B - Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_1$-$b_1$</td>
<td>$m_1$</td>
</tr>
<tr>
<td>2</td>
<td>$a_1$-$b_1$-$b_2$-$b_1$</td>
<td>$m_1$</td>
</tr>
<tr>
<td>3</td>
<td>$b_1$</td>
<td>$m_1$</td>
</tr>
<tr>
<td>4</td>
<td>$a_1$-$b_1$</td>
<td>$m_1$</td>
</tr>
<tr>
<td>5</td>
<td>$a_1$</td>
<td>$m_1$</td>
</tr>
</tbody>
</table>
Table 6.4: Source Code Navigation Paths for Task 2 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 2</th>
<th>Source Code B - Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_1$-$b_2$-$d_1$</td>
<td>$n_1$-$o_1$</td>
</tr>
<tr>
<td>2</td>
<td>$c_1$-$b_2$-$c_1$-$b_2$-$d_1$</td>
<td>$n_1$-$o_1$</td>
</tr>
<tr>
<td>3</td>
<td>$e_1$-$c_2$-$c_1$-$b_2$-$d_1$</td>
<td>$n_2$-$n_1$-$o_1$</td>
</tr>
<tr>
<td>4</td>
<td>$c_1$-$d_4$-$c_1$-$b_2$-$d_1$</td>
<td>$n_1$-$p_1$-$n_1$-$o_1$</td>
</tr>
<tr>
<td>5</td>
<td>$c_1$-$b_2$-$d_1$</td>
<td>$n_1$-$o_1$</td>
</tr>
</tbody>
</table>

Table 6.5: Source Code Navigation Paths for Task 3 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 3</th>
<th>Source Code B - Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>2</td>
<td>$c_1$-$d_3$-$c_1$-$b_3$-$c_1$-$b_3$-$c_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>3</td>
<td>$b_2$-$c_1$-$b_3$-$b_4$-$b_5$-$b_2$-$b_3$-$c_2$-$c_1$-$b_7$-$b_2$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>4</td>
<td>$c_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>5</td>
<td>$c_1$</td>
<td>$p_1$</td>
</tr>
</tbody>
</table>
codes. Tables 6.6, 6.7, and 6.8 show this view of the transcribed data.

Table 6.6: The Time Taken and the Number of Methods Investigated (NMI) by the Subjects for Task 1 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 1</th>
<th>Source Code B - Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NMI</td>
<td>Time</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>02:13</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>04:19</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>02:10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>00:59</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>03:55</td>
</tr>
</tbody>
</table>

6.2.5 Results

Based on a detailed analysis of the data collected during the study, we made several observations about the effects of the traceability rules in performing maintenance tasks. Tables 6.10, 6.11, and 6.12 show the difference in the time taken by each of the subjects to complete identical change identification tasks on source codes A, and B. Table 6.13 shows the same information for the set of three change identification tasks. As exhibited by these tables, in 14 out of 15 pairs of change subset identification tasks performed during the study, subjects completed their tasks in less time on the rule-based program compared to the non-rule-based program. To determine whether the observed improvement in subjects time performance is statistically significant (at the level of significance $\alpha = .05$), we performed statistical hypothesis testing.

In statistical terms, our hypothesis states that, for each of the three tasks, the mean of the paired differences (the time to completion on the non-rule-based code minus the time to completion on the rule-based code) is greater than zero. Accordingly, the null
Table 6.7: The Time Taken and the Number of Methods Investigated (NMI) by the Subjects for Task 2 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 2</th>
<th>Source Code B - Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NMI</td>
<td>Time</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>02:24</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>04:15</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>07:25</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>02:49</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>02:37</td>
</tr>
</tbody>
</table>

Table 6.8: The Time Taken and the Number of Methods Investigated (NMI) by the Subjects for Task 3 on Source Codes A, and B

<table>
<thead>
<tr>
<th>Subject</th>
<th>Source Code A - Task 3</th>
<th>Source Code B - Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NMI</td>
<td>Time</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00:57</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>01:47</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>19:44</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>00:52</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>00:47</td>
</tr>
</tbody>
</table>
hypothesis can be stated as the mean of the paired differences is zero. In other words, the null hypothesis states that the mean time performances are the same for both the rule-based and non-rule-based programs, whereas the research hypothesis states that the subjects performed better on the rule-based program than the non-rule-based program.

A paired t-test was applied to time performances on the two source codes for each of the three tasks. The results are presented in Table 6.9. Before applying the tests, we performed a normality test for each of the samples to ensure that the normality assumption needed for the t-test holds. Since subject 3 did not finish Task 3 on source code A successfully, time performance could not be measured and therefore subject 3 was not included in calculating the test statistic for Task 3.

Table 6.9: Results of the Paired t-test

<table>
<thead>
<tr>
<th>Task</th>
<th>Test Statistic</th>
<th>( H_0 ) Rejection Region</th>
<th>P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.722</td>
<td>test statistic &gt; 2.132</td>
<td>0.026</td>
</tr>
<tr>
<td>2</td>
<td>6.072</td>
<td>test statistic &gt; 2.132</td>
<td>0.002</td>
</tr>
<tr>
<td>3</td>
<td>3.89</td>
<td>test statistic &gt; 2.353</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Since all of the p-values are less than the standard alpha level of 0.05, we conclude that we have statistically significant evidence to reject the null hypothesis and support the alternative hypothesis.

Data from our study confirms that the observed improvement in subjects performance is directly caused by the presence of the traceability rules. Tables 6.6, 6.7, and 6.8 show the number of methods examined by each of the subjects to perform their change subset identification tasks. As illustrated by these tables, all subjects examined fewer or at most equal number of methods on the rule-based program code, compared to the non-rule-based program. The audio records captured from subjects through the think-aloud protocol confirmed that this is because the traceability rules in the rule-based version of
Table 6.10: Change in Time Performance on Task 1

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Time (s)</th>
<th>Percentage of Change in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Code A</td>
<td>Source Code B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>133</td>
<td>32</td>
<td>75.93</td>
</tr>
<tr>
<td>2</td>
<td>259</td>
<td>80</td>
<td>69.11</td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td>49</td>
<td>62.30</td>
</tr>
<tr>
<td>4</td>
<td>59</td>
<td>84</td>
<td>-42.37</td>
</tr>
<tr>
<td>5</td>
<td>235</td>
<td>51</td>
<td>78.29</td>
</tr>
</tbody>
</table>

Table 6.11: Change in Time Performance on Task 2

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Time (s)</th>
<th>Percentage of Change in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Code A</td>
<td>Source Code B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>144</td>
<td>25</td>
<td>82.63</td>
</tr>
<tr>
<td>2</td>
<td>255</td>
<td>41</td>
<td>83.92</td>
</tr>
<tr>
<td>3</td>
<td>445</td>
<td>276</td>
<td>37.97</td>
</tr>
<tr>
<td>4</td>
<td>169</td>
<td>59</td>
<td>65.08</td>
</tr>
<tr>
<td>5</td>
<td>157</td>
<td>71</td>
<td>54.77</td>
</tr>
</tbody>
</table>
Table 6.12: Change in Time Performance on Task 3

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Time (s)</th>
<th>Percentage of Change in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Code A</td>
<td>Source Code B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>57</td>
<td>21</td>
<td>63.15</td>
</tr>
<tr>
<td>2</td>
<td>107</td>
<td>39</td>
<td>63.55</td>
</tr>
<tr>
<td>3</td>
<td>1184</td>
<td>38</td>
<td>96.79</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>13</td>
<td>75.00</td>
</tr>
<tr>
<td>5</td>
<td>47</td>
<td>29</td>
<td>38.29</td>
</tr>
</tbody>
</table>

Table 6.13: Change in Total Time Performance on the Set of Three Tasks

<table>
<thead>
<tr>
<th>Subject</th>
<th>Time (s)</th>
<th>Time (s)</th>
<th>Percentage of Change in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Code A</td>
<td>Source Code B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>334</td>
<td>78</td>
<td>76.64</td>
</tr>
<tr>
<td>2</td>
<td>621</td>
<td>160</td>
<td>74.23</td>
</tr>
<tr>
<td>3</td>
<td>1759</td>
<td>363</td>
<td>79.36</td>
</tr>
<tr>
<td>4</td>
<td>280</td>
<td>156</td>
<td>44.28</td>
</tr>
<tr>
<td>5</td>
<td>439</td>
<td>151</td>
<td>65.60</td>
</tr>
</tbody>
</table>
the program code helped subjects to derive trace information about their tasks, which in turn helped them to focus their investigation on a smaller subset of the program (i.e., a smaller search space), eliminating the irrelevant parts. The traceability rules helped subjects to quickly narrow down their search to the relevant components. As a result, they performed their task in less time.

The analysis of the transcripts revealed a contrasting source code investigation pattern between the two versions of the program code. In the non-rule-based version of the code, all subjects had taken a top-down strategy to perform their change subset identification tasks. This means that they all started their investigation by first examining the classes they thought were relevant, at a high level, to the change task at hand. In all cases, the selection of the starting point of the search was a matter of guessing which class’s name suggests a relationship with the change task at hand. Once they found a class they perceived as relevant to the change task at hand, the subsequent places to investigate were decided upon based on systematically investigating the class’s source code and following the chain of method calls which were conceived as being relevant to the change task at hand. In summary, the subjects’ investigation pattern on the non-rule-based version of the program code can be described as cycles of large-scope code investigation and elimination of irrelevant parts, where the scope of the code investigation in each subsequent cycle is further narrowed down.

This top-down approach of the subjects on the non-rule-based program is in sharp contrast with the approach the same subjects took to perform their tasks on the rule-based program. Subjects investigations on the rule-based program were guided by the traceability rules rather than a wide-scope examination and elimination of the irrelevant parts. As a result, subjects were looking for and examining only specific program elements in the source code (i.e., package, class, method, or attributes derived from the traceability rules). In comparison to the non-rule-based program, the guidance provided by the traceability rules deployed in the rule-based program effectively eliminated the initial
several cycles, which were required in the non-rule-based program to narrow down the scope of the source code investigation to the relevant parts.

6.2.6 Threats to Experimental Validity

Several factors potentially affect the validity of our findings. We discuss these factors as well as the measures we took to minimize them under standard types of validity threats in empirical studies.

6.2.6.1 Construct Validity

The construct validity criterion questions whether the variables measured and studied truly reflect the phenomenon under study. An in-depth analysis of the actions and the thought process of the subjects while performing their tasks, the results of performing their tasks (i.e., the subsets of the programs they identified as relevant to the change tasks), and the times taken to complete their tasks are the three main measures in our study. The parts of the system identified by subjects as relevant to the change tasks and the times taken to complete each of the tasks could be measured objectively. We selected the change tasks based on a detailed analysis of both source codes so as to make it possible to objectively evaluate whether the parts of the system identified by subjects as relevant to the change tasks were actually relevant.

To analyze the source code investigation behavior of developers, we relied on an abstraction of their behavior based on transcribing the recorded video and audio data. To reduce the potential errors of interpretation, we interviewed subjects at the end of each experimental sessions. As recommended in [11], we increased the validity of our analyses by using multiple sources of evidence.

The purpose of the study was to investigate how the conformance of a system’s source code to a set of traceability rules affect the way developers perform software maintenance tasks. Therefore, the variables measured and studied reflect the purpose of the study.
6.2.6.2 Internal Validity

Prior knowledge of the target program used in the study, as well as the guidelines and comments provided by the investigator during the experiments could be possible sources of interference with the study. To eliminate the effects of these factors, we ensured that no subject had prior knowledge of the program used in the study. We also limited the guidelines provided during the experiments, as well as our answers to subjects’ questions. We provided only answers to questions about the material already covered in the training phase of the study, and when necessary clarified the tasks. We took caution not to reveal any information that could give hints to subjects on how to accomplish their tasks.

In most software engineering experiments where the results of applying the treatments by different subjects are compared to each other (i.e., a between-subject study design), subjects’ varying backgrounds and experience levels can be an important confounding factor; therefore, special measures should be taken to ensure that all subjects have similar backgrounds and experience. However, since in our study we were focused on the differences in individual subject’s source code investigation behavior as a result of the changes in the structural properties of the source code (rule-based vs. non-rule-based), the potential effects of experience is reduced. Each subject’s behavior is compared to his/her own behavior after switching to the other version of the source code (i.e., a within-subject study design). This is in contrast with software engineering studies in which the focus is on the contrast between the behaviors of different subjects.

Any difference in the qualities of the two programs not resulted form the absence or presence of the traceability rules in the program codes could be a possible confounding factor. For instance, it is well-known that code comments can improve the maintainability of systems. Therefore, the different qualities of the code comments in the two programs could interfere with the results of the study. Although such factors cannot be completely eliminated, we took measures to minimize them. The two programs used in our study were functionally equivalent. We removed all of the comments from both programs, and
both programs conformed to Java’s standard naming conventions.

6.2.6.3 External Validity

We used five subjects in our study. This small number of subjects can be a limit to the generalizability of our findings. Overall, subjects performed 15 pairs of change subset identification tasks. As with most statistical tests, larger sample sizes yield more reliable results. We performed qualitative data analysis as well to gain more confidence in our results (i.e., multiple sources of evidence).

The findings of the study derived from the qualitative analysis of the context-rich video and audio data collected from subjects while performing their tasks on the two source codes support the result obtained from the statistical hypothesis testing. This combined use of quantitative and qualitative evidence is a strength of our study. Whereas the quantitative analysis confirmed that the observed improvement in subjects time performances is significant, the qualitative analysis confirmed that the presence of the traceability rules is the reason why subjects performed better on the rule-based version of the program code.

The small sizes of the programs used in our study is another limit to the generalizability of our findings. We can gain more confidence in our findings by replicating the study with a larger group of subjects working on larger programs or by putting our approach into practice in real situations. The results of our study can be viewed as early evidence for the effectiveness of the rule-based approach to traceability in improving maintainability. As future work, we plan to evaluate our traceability approach on real-world industrial projects in order to increase the external validity of our study.

All of the subjects had enough software development experience in the industry to be considered as professionals. Therefore, our study was not subject to the difficulties arising from generalizing from students to professionals, which is a concern for most software engineering studies conducted in academic settings. Our study involved three pairs
of tasks representing three different types of software modification problems typically found in real-world software development: fixing defects, changing the existing functionality, and enhancing the existing functionality of the system. This allowed us to collect enough data from subjects to perform a more reliable qualitative analysis. In particular, collecting six different sets of data for each subject made it possible to identify patterns, commonalities, and differences in individual subject’s behavior, as well as across all subjects. Our use of a video and sound recording program to capture all the actions and the thought process of developers while performing their tasks ensured that no information is lost. Therefore, we believe that the rich data set captured through the three pairs of change tasks contributed to the validity of our findings.

6.2.6.4 Reliability

Our study has been reported in this chapter and the experimental material, including the target programs, are available to interested researchers. Therefore, it should be possible to replicate the study. However, we encourage interested researchers to replicate the study using other systems and tasks. Conducting the study in various settings might yield different results. If repeated studies result in similar findings, we can be more confident in the results of this study. Also, researcher Bias, which is commonly understood as inevitable, can be a threat to the validity of the findings. Replication of the study by others can reduce the effect of researcher bias and therefore will increase the reliability of the findings of the study.

6.3 Summary

In this chapter, we reported on an empirical study that investigated the impact of the rule-based traceability in performing maintenance tasks. We observed that subjects per-
formed significantly better on the rule-based version of the program. They took shorter source code navigation paths, and completed their change subset identification tasks in less time. Data collected through the think-aloud protocol shows that the observed improvement in subjects’ performances is a direct consequence of the rule-based traceability. Subjects were able to use the rules deployed in the rule-based version of the program to generate trace information, which facilitated their tasks. The study further increased our confidence in the practicality of the rule-based approach to traceability.

Appendix

The figures in this appendix are derived from Table 6.1. Figures 6.1, 6.2, 6.3, 6.4, and 6.5 compare each subject’s performance in terms of the time required to complete each of the tasks on the two different versions of the source code. Figure 6.6 depicts the total time taken by each subject to complete the set of three change tasks on the two different versions of the source code.

![Figure 6.1: Subject 1’s Performance on Two Different Versions of the Source Code](image)
Chapter 6. An Empirical Study of Rule-Based Traceability and Maintainability

Figure 6.2: Subject 2’s Performance on Two Different Versions of the Source Code

Figure 6.3: Subject 3’s Performance on Two Different Versions of the Source Code

Figure 6.4: Subject 4’s Performance on Two Different Versions of the Source Code
Figure 6.5: Subject 5's Performance on Two Different Versions of the Source Code

Figure 6.6: Subjects' Total Performance on Two Different Versions of the Source Code
Chapter 7

A Case Study of Defect Introduction Mechanisms

In Chapter 4, we stated that, in addition to traceability, reduction in defect rate is another implication of the proposed rule-based process for software construction. Empirical evidence obtained from the industrial case studies of defects, reported in this and the next chapters, will be used to support this argument.

It is well known that software production organizations spend a sizeable amount of their project budget to rectify the defects introduced into the software systems during the development process. An in depth understanding of the mechanisms that give rise to these defects is an essential step towards the reduction of defects in software systems. In line with this objective, we conducted a case study of defect introduction mechanisms on three major components of an industrial software system, which is presented in this chapter.

\footnote{The case study reported in this chapter has been published in [31].}
Chapter 7. A Case Study of Defect Introduction Mechanisms

7.1 Introduction

It has been frequently mentioned in the software engineering literature that maintenance activities are the dominant costs of developing software systems. As discussed in Chapter 1, studies have shown that changes made to software systems account for 40 to 90 percent of the total development costs [7] [28] [5] [19] [15]. These changes can be corrective, adaptive, perfective, or preventive. Corrective changes deal with fixing the defects introduced into the software systems during the development process, and account for a significant portion of the maintenance costs; in their study of 487 data processing organizations, Lientz and Swanson [57] reported that, on the average, about 21% of the maintenance effort is allocated to the corrective maintenance. More recent studies have concluded that, in spite of the advances in software engineering in the past few decades, the maintenance problems have remained the same [64] [88]. These figures clearly indicate the potential economic value that can be gained from leveraging defect prevention techniques. A reduced rate of defects in the delivered software results in a reduction in the corrective maintenance activities, which in turn translates into a lower total development cost.

The importance of defect prevention has been emphasized by quality standards such as the Software Engineering Institute’s Capability Maturity Model (SEI-CMM) [95], where defect prevention is a key process area for the optimizing maturity level (i.e., CMM Level 5). According to CMM [89]:

"Defect prevention involves analyzing defects that were encountered in the past and taking specific actions to prevent the occurrence of those types of defects in the future. The defects may have been identified on other projects as well as in earlier stages or tasks of the current project. Defect prevention activities are also one mechanism for spreading lessons learned between projects."

\[2\]The adaptive, perfective, and preventive maintenance activities have on average a share of 25%, 50%, and 4% of the total maintenance efforts, respectively.
It goes without saying that an important first step to devising tools, techniques, and processes to counteract the mechanisms that give rise to software defects is to gain an understanding of these mechanisms. The work reported in this chapter is an effort in this direction.

Eldh et al. [24] emphasize that it is important to regularly collect and report findings about software defects from real industrial and commercially used systems to keep information in tune with development approaches, software and faults. They identify the lack of recent industry data for research purposes as the key problem. This view is supported by Mohagheghi et al. [61] who argue that there is a lack of published empirical studies on large industrial systems and that many organizations gather large volume of data on their software processes and products, but either the data is not analyzed properly, or the results are kept inside the organization. This situation hinders the spreading of lessons learned between projects in various organizations. The case study reported in this chapter is a response to this need for more empirical studies on industrial systems.

The main purpose of the study is to collect empirical evidence to answer the following two research questions:

1. What are the mechanisms that gave rise to defects in the case under study?

2. How large a role each identified mechanism has played in introducing defects in the case under study?

To answer our research questions, we performed root cause analysis (RCA) on 449 defects from a commercial Enterprise Resource Planning (ERP) software system. Our analysis, backed up by evidence drawn from project data including the defect reports in the defect tracking system, source code, requirements specification documents, and

---

3 Enterprise Resource Planning systems are business software systems that integrate the various information systems required to support the operations in various departments within an organization into a unified system.
test cases, as well as group sessions and individual interviews with the project team members identified a number of defect categories and their root causes, along with their frequencies. The rest of this chapter describes the case study.

7.2 Case Study

7.2.1 Context of the Case Study

The organization where the case study was conducted is a manufacturer of telecommunication devices and an information technology company. For confidentiality reasons, we keep the organization anonymous. For the past five years, the IT department has been actively involved in developing a web-based ERP system comprised of various subsystems\footnote{Throughout this chapter, we use the terms system, subsystem, component, and module (source-code file) as units of system decomposition from largest to smallest, respectively.} including Bookkeeping, Inventory Management, Human Resources, Administration, Manufacturing, Procurement, and Workflow Management. The implementation of the ERP system is carried out in Java programming language, and the software developers involved in the project have an average of 7 years of industry experience.

The IT department follows a customized development process, which borrows concepts from both Rational Unified Process (RUP) and Extreme Programming (XP) methodologies. For instance, most programming is performed in pairs, which is an XP practice, whereas the requirements and analysis phases are conducted through a more traditional RUP-like process using use cases.

At the time of this study, the project team was comprised of 28 individuals in various roles. Table 7.1 exhibits the project team structure in terms of the project roles and the number of individuals in each role. As with most long-term software projects, a number of individuals have left the team during the project, while a few others have joined the team. The project team has had 35 members in its largest size. The development of the
Chapter 7. A Case Study of Defect Introduction Mechanisms

Bookkeeping, Human Resources, Administration, and Inventory Management subsystems has been completed, while the remaining subsystems are still under development. The final product is estimated to contain 200,000 (non-commented and non-blank) lines of code.

Table 7.1: Project Team Structure

<table>
<thead>
<tr>
<th>Project Role</th>
<th>Number of Team Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Manager</td>
<td>1</td>
</tr>
<tr>
<td>System Analyst</td>
<td>5</td>
</tr>
<tr>
<td>Developer</td>
<td>13</td>
</tr>
<tr>
<td>Tester</td>
<td>4</td>
</tr>
<tr>
<td>Graphics Designer</td>
<td>2</td>
</tr>
<tr>
<td>Marketing Representative</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total Team Size</strong></td>
<td><strong>28</strong></td>
</tr>
</tbody>
</table>

The company uses the following process for handling defects. When a defect is detected, a defect report is registered in a defect tracking software tool. Each defect report captures a set of information including a unique defect identifier, a summary of the defect, a detailed description, the date the defect report is created, the subsystem to which the defect is attributed, the reporter of the defect, the assignee of the defect, priority, status, resolution, any number of additional comments by team members, and the date the defect report is last updated.

During system testing, all detected defects are assigned to a single member of the development team (i.e., a point of contact between the testing and development teams), who in turn reviews the reported defects and further assigns them to the appropriate developers (i.e., the developer who has introduced the defect into the system) to be fixed. The idea behind this practice is that the developer who implements a feature is also the
most qualified team member to rectify the defects reported on that feature since he/she is considered to be the most knowledgeable team member about the implementation details of that feature and therefore should be able to rectify the defect more reliably and in less time.

All team members, periodically or upon request by a team member, review all or some of the reported defects, and if they have any specific information or comments that can facilitate the correction of the defects, add them to the defect reports in the defect tracking system. Typical information in the added comments include the cause of the defects, the location of the defects in the source code, how to fix the defects, and comments to clarify the descriptions of the defects. Developers assigned to the defects then use this information to correct the reported defects.

The development team uses the CVS version control system to keep track of the changes made to the source code. They commit their code changes to the central repository after testing them on their local machines. Depending on the lengths of the tasks assigned to individual or pairs of developers, the frequency of code check-ins by developers varies from a few hours to a few days.

7.2.2 Description of the Case

All ERP subsystems in the studied organization are built on top of a shared infrastructure layer, which is composed of a set of reusable libraries and frameworks. The components in the infrastructure layer are either developed in-house or acquired as open-source software. All subsystems follow a three-tier layered architectural style, which is comprised of user interface, application logic, and data access layers. Each layer is considered a distinct component and as such each subsystem is divided into three major components. Our case study concerns the three components of the Bookkeeping subsystem. The target subsystem has been under development for a period of two years from 2005 to 2007. The first five columns in Table 7.2 present the characteristics of the target components in terms
Table 7.2: Characteristics and Metrics of the Target Components

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>JSP</td>
<td>7802</td>
<td>41</td>
<td>419</td>
<td>180</td>
<td>150</td>
<td>330</td>
<td>38.606</td>
</tr>
<tr>
<td></td>
<td>Javascript</td>
<td>678</td>
<td>1</td>
<td>181</td>
<td>190</td>
<td>150</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XML</td>
<td>126</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Logic</td>
<td>Java</td>
<td>9434</td>
<td>24</td>
<td>393</td>
<td>22</td>
<td>41</td>
<td>63</td>
<td>6.677</td>
</tr>
<tr>
<td>Data Access</td>
<td>Java</td>
<td>4602</td>
<td>59</td>
<td>78</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>12.168</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>22642</td>
<td>128</td>
<td>258</td>
<td>191</td>
<td>449</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 7.2.3 Case Study Process and Data Collection

At the time of this study, there were 482 defect reports in the defect tracking system, assigned to the target subsystem that we used in our case study. Of these, 4 defects were labeled as "Duplicate", and 29 defects were labeled as "Not a Bug". The defects labeled as "Duplicate" were reported twice in the defect tracking system, whereas the ones labeled as "Not a Bug" were not actually defects and were initially reported as defects as a result of the incorrect usage of the subsystem or unfamiliarity of the reporter of the defect with the correct behavior of the subsystem. We excluded these 33 defect reports, which left us 449 unique defects to study.

To analyze the distribution of the defects over the target components, the main investigator and a member of the studied organization, who had a thorough understanding of the system, independently followed the analysis process described below to analyze each of the defect reports and attribute them to their corresponding components. This analysis of the distribution of the defects over the three target components was required...
since this information was not readily available; records in the defect tracking system included a data field that captured the attribution of the defects to the subsystems, but no data fields were available to capture the attribution of the defects to the lower-level units such as components and modules. The results from this analysis were required to compute the defect count and defect density\(^5\) measures for each of the target components.

We used the following process to analyze the distribution of the reported defects over the three target components. We started our analysis by checking the information recorded in the "Defect Summary", "Detailed Description", and all of the available "Additional Comments" fields for each of the defect reports. Based on the information recorded in these fields, a group of the defects could be directly attributed to their corresponding components in the source code. For the remaining group of the defects, where the attribution of the defects to the components could not be derived from the information available in the defect reports, we recovered this information through conducting a series of defect review sessions. Each session was attended by two people: a research investigator who facilitated the session and recorded the recovered information, and a developer who had been involved in fixing the defects. During each defect review session, all defects fixed by the participant developer were discussed and attributed to their corresponding component. Where necessary, the system’s source code was consulted to locate the components related to the defects. In addition to the attribution of the defects to their corresponding components, where possible, for each defect, we identified the internal cause of the defect in the source code (e.g. an incorrect database query statement or missing source code statements). We also determined whether the defect was caused as a result of a missing or incorrect implementation.

To ensure the quality of the collected information, the abovementioned analysis process was conducted twice and independently. The results of the two analyses were in

\(^5\)The defect density of a software component is defined as the ratio \(X/Y\), where \(X\) denotes the number of defects in the component, and \(Y\) denotes the size of the component measured in lines of code (LOC) or thousand lines of code (KLOC).
close agreement, which is an indication of the objectivity of the analyses performed. The cases where there was a difference between the two analyses were jointly reexamined to reach a consensus.

We then measured the sizes of the target components in non-blank and non-commented lines of code using a software tool, and computed the defect density for each of the target components. The results are summarized in the last two columns in Table 7.2, which present the Defect Count and Defect Density (in defects per KLOC) for each of the target components, respectively.

We followed our data collection and analysis process by performing root cause analysis of the reported defects through conducting a series of group sessions and interviews with the team members. During these sessions, we used input from project team members and, for each defect, identified the external factor that underlay the internal cause of the defect (e.g., the unfamiliarity of the developers with the new query language underlying the incorrect query statements, or incomplete requirements specification documents underlying the missing source code statements). To ensure the correctness of the identified root causes, the team members frequently consulted the project data including defect information in the defect tracking system, requirements documents, test cases, and the source code, as well as the results of the analysis of the attribution of the defects to the components and the collected project metrics (size, defect count, and defect density). As a result of these sessions, we traced each defect back to its origin, which led to the classification of the reported defects based on their root causes. In what follows, we discuss the findings of the study in the form of a number of observations about the mechanisms that gave rise to defects in the subsystem under study.

7.2.4 Results

Based on a detailed analysis of the data collected during the study, we make several observations about the mechanisms that gave rise to defects in the considered subsystem.
We discuss the impact of each identified mechanism on defect rate of the subsystem under study.

7.2.4.1 The Impact of Adopting New, Unfamiliar Technologies on Defect Rate

The data from our study show that of the 56 defects attributed to the data access component (see Table 7.2), 47 (roughly 84%) were caused by incorrect database query statements. We discussed this finding with the development team and found that the unexpected number of query-related defects in the data access component was due to the adoption of a new database query technology. The development team had adopted an unfamiliar query language to implement the data access component. Since there had been no previous experience and expertise on the newly adopted query language, the team had encountered many problems with this new technology. Only 16% of the defects attributed to the data access component were caused by the code written in the Java language, which constitutes the bulk of the code in the data access component and serves as the host language for the embedded queries.

We excluded the defects directly caused by the introduction of the new technology and recalculated a defect density of 1.955 for the data access component. Comparing this new calculated value for the defect density in the data access component with its current value from Table 7.2 (12.168) clearly demonstrates the negative impact of adopting the new, unfamiliar technology in increasing the defect rate in this component. As a result of adopting the unfamiliar technology, the defect density of the data access component has increased by a factor of 6. The data from our study suggest that:

*The adoption of new, unfamiliar technologies into a software component is a risk factor that has adverse effects on the component’s quality in terms of its defect rate.*
This observation is not surprising. There is an intuitive consensus in the software engineering literature on the correctness of this proposition. However, empirical evidence taken from industrial software systems, like ours, to support it can strengthen our beliefs in this proposition.

7.2.4.2 The Impact of Incomplete Requirements Specifications on Defect Rate

As part of our data analysis, we classified the reported defects under two broad categories of incorrect implementation and missing implementation, and observed that about 57.5% of the defects (258 cases) were caused as a result of incorrect implementations of the requirements in the source code, whereas the remaining 42.5% of the defects (191 cases) were a result of missing implementations from the source code. Columns 6 and 7 in Table 7.2 present the distribution of defects classified as incorrect versus missing implementation over the three target components. We further observed that the majority of the defects classified under the missing implementation category were related to missing business rules and data validations. Defects caused by missing implementations (i.e., the code that is necessary is missing) have also been referred to as “faults of omission” in the literature [24].

Our inspection of the requirements specification documents revealed that in 156 cases (roughly 82% of the defects in the missing implementation category), the system analysis team had not explicitly included the requirements in the requirements specification documents and were consequently missing from the system’s implementation as well. In a series of defect root cause analysis group session with the team members, we reviewed all of the 191 defects in the missing implementation category, and confirmed that the reason for the 156 missing implementation cases which didn’t have corresponding requirements was actually the missing requirements.

We know for a fact that these requirements specification documents were the main
means through which the requirements of the system were communicated to developers. The introduction of this group of defects into the system’s source code can be directly traced back to the incomplete requirements specifications. This conclusion is consistent with the results of the interviews conducted with the development team members. When asked about the relatively large number of defects related to missing implementations, the team members expressed their opinions in statements such as ”what is obvious for the business analysis team is not clear to anyone else in the development team. Consequently, if they fail to communicate some of the business requirements in an explicit manner, we are highly likely to miss these requirements in our implementations” and ”when we start the development of a new use case of the system, we are not provided with all the details. What we initially receive from the business side includes a description of the use case including the main and alternative flows, and some of the major related business requirements, but the documents are not comprehensive enough to cover all aspects of the use cases including some of the data validations and less obvious business rules. Therefore, a number of missing requirements are detected during the system testing”.

An interesting aspect of this observation is that it puts into question the comprehensiveness of the traditional view of a defect as any characteristic of the system that does not comply to its predetermined (proactively and explicitly documented) specification. In our case, we observed that there were no predetermined specifications for a significant number of functionalities in the system and yet the absence of these functionalities from the system were reported as defects, whose rectification were required for the correct operation of the system.

An interview with the testing team revealed that they had partly relied on their implicit knowledge of the system domain to test the software system. For instance, while the requirements documents lacked some of the data validation rules, the testing team, relying on their knowledge of the system, had identified and incorporated some of these missing rules into their test cases. A consequence of this phenomenon is that a part of
software requirements are documented outside the requirements specification documents, and inside the test cases and defect records. This practice, over time, can lead to the loss of parts of the system knowledge as a part of system requirements are buried within test cases and defect reports. Our data suggest that:

\[ \text{Incomplete requirements specifications (i.e., some requirements are not explicitly stated in the specifications) are a possible mechanism for introducing defects into software systems in the form of missing implementations.} \]

As mentioned earlier, in our case, the development team was following a traditional document-centric requirements process. This means that the majority of communication between the business analysis and development teams was taking place through requirements documents. A direct consequence of this reliance on documentation as the main form of communicating system requirements is that the performance of the system developers (e.g., in terms of the number of defects related to missing implementations introduced into the system) becomes partly dependent on the quality of the documents produced by the requirements team (e.g., in terms of the completeness of the system requirements). The results could be different in software projects with an agile requirements process, where the development team mostly relies on verbal communication of requirements.

7.2.4.3 The Impact of the Lack of Requirements Traceability on Defect Rate

An interesting observation was that in the remaining 35 defects classified under the missing implementation category (roughly 18% of the defects in this category), the corresponding requirements were existing somewhere in the requirements documents and were somehow overlooked by developers.

Our inspection of the requirements documents in conjunction with group sessions
with the team members revealed that these cases were related to the business concepts or data items and their associated business rules and data validations that were defined in one requirements document and implicitly referred to in other requirements documents. For instance, consider the case where a step in one of the system use cases states that the user shall enter a data item into the system as part of the data entry for that use case, without explicitly making references to the other use cases of the system where the data validation rules for this data item have been specified. Since there were no explicit links between the parts of a document that referred to external business concepts or data items and the parts of the other documents that actually specified the requirements for that concept (requirement-requirement traceability link), developers either did not realize that some of the requirements pertaining to the feature under development are defined somewhere else, or had to manually navigate between the various requirements documents to capture a complete view of the requirements pertaining to the feature under development.

This process of moving from a requirements document to another in order to collect all the relevant requirements can be problematic when a document has many points of implicit reference (i.e., items mentioned in a document are specified in other external documents, but not explicitly linked to those external documents). In other words, the requirements related to a feature under development were scattered across different documents without an explicit mechanism for cross-referencing, and the human errors involved in the process of navigating between the various requirements documents and collecting the complete set of requirements had led to the introduction of a group of defects into the system in the form of missing implementations. Since, other than system testing, there was no other mechanism in place to verify the completeness of the implementation of a feature under development with regards to its specified requirements, these missing implementation defects were remained hidden until system testing. Our data suggest that:
The lack of traceability between requirements specification documents, when the requirements pertaining to a feature of the system are scattered across multiple documents, plays a role in the occurrence of the cases where the requirements are existing in the requirements documents but missing from the implementation.

7.2.4.4 The Impact of Not Proactively Defining and Enforcing the User Interface Consistency Rules on Defect Rate

Another observation is that the cause of 8% of the defects in the user interface component (or 6% of the total number of defects in the subsystem under study) can be directly traced back to inconsistencies in the user interface. A close examination of this group of defects in conjunction with a group session with developers revealed that in the absence of explicit consistency rules for the unification of the system’s user interface behavior, developers had made individualistic and ad hoc decisions in their implementations, which led to inconsistencies in the user interface of the system. The descriptions given for these defects in the defect tracking system refer to various types of inconsistencies in the user interface of the system including inconsistencies in the screen layouts, user interface navigation methods, fonts, and data formats in various screens and reports displayed by the system. In the subsystem under study, these forms of inconsistencies were considered to be detrimental to the usability of the application and as such any occurrences of such inconsistencies in the user interface were reported as defects. The data from our study suggest that:

The lack of explicit and proactive definition and enforcement of implementation consistency rules leads to defects in cases, such as the user interface, where inconsistent implementations are considered defects.
7.2.4.5 The Impact of Software Size on Defect Rate

The relationship between defect measures such as the defect count and defect density, and software size has been the subject of many studies in the literature. For example, [26], [65], [3], [78], [90], [2], [38], [76], and [58] are some of the studies that have been conducted in this area. Table 7.5 summarizes the key results from these nine studies. Some of the previous studies report a relationship between these parameters, while others do not. What makes the situation complicated is that the results from the studies where a relationship between these two parameters have been observed are conflicting. Some of these studies report a rising trend of defects as the size increases, while other studies report a declining trend of defects as the size grows. Others have tried to explain these rising and declining trends.

The data from our study does not suggest any noticeable dependence between defect rate and component size. 73.5% of all reported defects are attributed to the user interface component. In contrast to this high defect concentration, the business logic, and data access components have a share of only 14% and 12.5% of the total reported defects, respectively. Obviously, this noticeable difference between the defect distributions over the user interface component and the other two components cannot be attributed to the sizes of the components. Although the user interface component is almost the same size as the business logic component, and only twice the size of the data access component, it has at least five times more defects compared to each of the other two components.

We would like to study the relationship between defect measures and module size (as opposed to component size). Unfortunately, no data was available on the distribution of defects over the modules. An attempt to collect these data after the fact would be extremely time consuming and error prone.

Two comments are worth making regarding the relationship between size and defect rate. First, the discrepancy among the reported results and findings in the published literature prevents us from drawing a solid conclusion about the relationship between
these two parameters. Second, generalized models, like the ones proposed by some of the previous studies, cannot take into account the context-specific factors that affect the distribution of the defects over the software components. For instance, the higher than normal defect rate of the data access component in our study, which is a result of the unfamiliarity of the development team with the newly adopted technology, can not be explained by any of the proposed models. Jorgensen and Sjoberg [48] have argued that the variation and dynamics of software engineering are so large that the definition of a population that meets the requirements set by statistical hypothesis testing, which has been the basis of generalization in many of the previous studies, can be extremely difficult. Accordingly, they suggest that instead of trying to generalize from a sample to a larger population, we should try other alternatives such as generalizing from one software engineering study context to other context (i.e., across populations) through building theories.

7.2.5 Summary of the Results

Based on the analysis of the collected data, we can now answer the two research questions posed in the introduction.

1. What are the mechanisms that gave rise to defects in the case under study?

2. How large a role each identified mechanism has played in introducing defects in the case under study?

Table 7.3 summarizes the answers to these questions. The major conclusions and contributions of our study are as follows:

- A significant portion of the defects originate outside the source code. This finding is supported by the evidence that 59% of the reported defects were propagated into the considered subsystem from external sources including incomplete requirements
specifications, adopting new, unfamiliar technologies, lack of requirements traceability, and the lack of proactive and explicit definition and enforcement of user interface consistency rules.

- Specification-related defects represent the largest category of defects (42.5%, of which 34.7% were caused as a result of incomplete requirements specifications, and 7.8% were a result of the lack of traceability between various requirements specifications).

The abovementioned findings suggest areas where effort should be directed.

Table 7.3: Defect Introduction Mechanisms Identified in the Subsystem Under Study

<table>
<thead>
<tr>
<th>Defect Introduction Mechanism</th>
<th>Count</th>
<th>% of Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect implementations not linked to external causes</td>
<td>184</td>
<td>41</td>
</tr>
<tr>
<td>Incomplete requirements specifications</td>
<td>156</td>
<td>34.7</td>
</tr>
<tr>
<td>Adopting new, unfamiliar technology</td>
<td>47</td>
<td>10.5</td>
</tr>
<tr>
<td>Lack of traceability between requirements specifications</td>
<td>35</td>
<td>7.8</td>
</tr>
<tr>
<td>Lack of consistency in the user interface</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>449</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

### 7.2.6 Implications of the Findings

Given the profile of defect sources identified in our case study, in Table 7.4 on page 135, we propose a set of mitigation strategies that software development organizations can employ to counteract these defect introduction mechanisms.
7.2.7 Threats to Validity

Several factors potentially affect the validity of our findings. We discuss these factors under standard types of validity threats in empirical studies.

7.2.7.1 Construct Validity

We use defect count and defect density as surrogate measures for the quality of the software components. These measures are widely used in software engineering studies. All component and module sizes were measured in lines of non-commented and non-blank code. The purpose of the study was to identify some of the mechanisms that give rise to defects in software systems. Our observations involve the calculated measures of size, defect count (for various categories of defects), and defect density. Therefore, the variables measured and studied, truly reflect the purpose of the study.

7.2.7.2 Internal Validity

In our study, the incorrect attribution of the defects to the components could be a threat to the internal validity of our findings. To minimize the potential effect of this confounding factor, the author of the paper and a member of the development team, who had a detailed knowledge of the system and had been actively involved in fixing the reported defects, independently analyzed the defects and attributed them to their corresponding components. For the great majority of the defects, we were able to accurately assign the defects to their corresponding components. The accuracy of the assignment of defects to the components was evident from the closely matching results of the two analyses. The cases where there was a difference in the results of the two analyses were jointly reexamined and resolved. We are confident that the attribution of defects to their corresponding components was accomplished accurately.

We studied the entire population of the reported defects in the subsystem under study. This effectively eliminated any potential sampling bias, which can be a problem for
studies where a selected sample of the population is included in the study. Furthermore, the inclusion of the complete set of defects in our study not only helped us to obtain a complete picture of the defects and their root causes, but also increased the reliability of the conclusions drawn from the analysis of the data.

In studies where components developed in multiple languages are involved, an equivalent code must be calculated for the components, to make the comparison of the component sizes meaningful. The software tool used in our study to measure the component sizes provides an equivalent code size in a hypothetical third generation programming language. To calculate the equivalent sizes, the software tool multiplies the component size in Java with 1.36, Javascript with 1.48, JSP with 1.48, and XML with 1.90. We recalculated all the metric data collected in our study using the equivalent sizes. The results did not change any of our findings or conclusions.

To further validate the findings of the study, we discussed them with the project team members during our interviews and group sessions. They believe that the five findings of the study, as discussed under the Section 7.2.4 of this chapter, capture an accurate portrayal of their situation.

7.2.7.3 External Validity

Our dataset was taken from one product of a development organization, which can be a limit to the generalizability of our findings.

7.3 Related Work

The work of Leszak et al. [54] is similar to ours in intent. They conducted a root cause analysis study of defects in a large transmission network element software, and based on the findings of the study devised countermeasures to either prevent the defects or detect them earlier in the development process. Results from our study contradict their
findings. They concluded that the majority of defects do not originate in early phases. They report that in the system they studied defects were introduced into the system predominantly (71%) within the component-oriented phases of component specification, design, and implementation. In contrast to our study results, in their case, requirements-related defects did not have a significant contribution to the total number of defects. This might be because in the type of software they studied more attention is paid to requirements development compared to business software systems.

Eldh et al. [24] studied and classified the failures in a large complex telecommunication middleware system. They concluded that faults related to unclear specifications (46.6%) dominate among the software faults. In their case, faults of omission and spurious faults accounted for 38.3% and 8.3% of the total faults, respectively. Our study agrees with their finding. In our case, we also observed that specification-related defects represent the largest category of defects (42.5%). Our work is different from theirs both in its motivation and the focus of the study. Our goal is to identify the causes of defects so that appropriate actions can be initiated to prevent the sources of defects. To fulfill our goal, we performed root cause analysis of the reported defects and traced the defects to their sources outside the source code and into the external factors (e.g., specifications, processes, and decisions). In contrast, the main motivation behind the classification of faults in Eldh et al.’s work is to investigate software test techniques through injecting the identified classes of faults into code. As a result, in contrast to our focus on the external root causes of the defects, their focus is on the static origin of the faults within the source code.

From a research methodology point of view, our data selection approach is different from both Eldh et al. [24] and Leszak et al.’s [54] study. In our study, the entire population of defects in the considered subsystem was included for analysis. In comparison, the data used in Eldh et al.’s [24] study were selected by convenience sampling. They selected their sample data set from the defects whose labels in the configuration man-
agement system allowed them to trace the failures back to their origins in the system’s source code. Leszak et al.’s [54] used a combination of manual and random sampling. Another distinction between our study and the two previous studies is that our data is taken from an ERP system, which is a business software system, whereas both previous studies collected data from system-level software products namely, telecommunication middleware software and transition network element software.

Chillarege et al. [14] propose a semantic classification of defects called Orthogonal Defect Classification (ODC), which is comprised of eight distinct defect types, each capturing the meaning of a type of defect fix. The distribution of defects over the ODC classes changes with time, which provides a measure of the progress of the product through the process. The main motivation for ODC is to provide feedback to developers during the development process. In contrast to ODC’s focus on providing in-process feedback, the classes of defects identified by our study, along with the observed distribution of defects over these classes can provide after-the-fact feedback to developers, which can be used to improve the development of the next subsystems within the organization. In this sense, a study like ours can serve as a means for spreading lessons learned between projects. Given the qualitative nature of performing root cause analysis of defects, the resources required to perform the analysis are significant, which might make it impractical for providing in-process feedback.

There have been several studies about the relationship between defect-based measures such as defect count and defect density, and software size. Table 7.5 summarizes the findings of these studies. The data from our study, as discussed earlier in Subsection 7.2.4.5, does not show any noticeable dependence between defect-based measures and component size.
7.4 Conclusion

Defect prevention is a possible way to reduce software maintenance costs. However, to devise tools, techniques, and processes to support defect prevention requires an understanding of the mechanisms that give rise to defects during the development of software system. Case studies of real-world industrial systems are a systematic approach towards gaining such an insight. The study reported in this chapter is meant to serve such a purpose.

Our case study identified four possible defect introduction mechanisms including incomplete requirements specifications, adopting new, unfamiliar technologies, lack of traceability of requirements, and the lack of explicit definition of user interface consistency rules that collectively account for 59% of the defects in the subsystem under study.

For the remaining 41% of the reported defects, no external contributing factors could be found. We will investigate this group of defects in another case study, which is the topic of the next chapter. In our next case study, we analyze historical data from the revision control system to track and analyze the changes made to the software modules to fix the defects in order to understand the nature of these defects.
Table 7.4: Defect Mitigation Strategies

<table>
<thead>
<tr>
<th>Defect Source</th>
<th>Defect Mitigation Strategy</th>
</tr>
</thead>
</table>
| Incomplete requirements specifications            | • Improving the requirements process in terms of the completeness of the requirements specifications  
|                                                   | • Explicit documentation of all business rules and data validations                        |
| Adopting new, unfamiliar technology                | • Thorough evaluation of new technologies before adopting them                             |
|                                                   | • Provision of sufficient training when a decision is made to adopt a new technology        |
| Lack of traceability between requirements          | • Practicing requirements traceability                                                    |
|                                                   | • Automated support for requirements process including tools to help the project team to keep track of the relationships between requirements and requirements status |
| Lack of consistency in the user interface          | • Proactive definition and enforcement of user interface design rules to unify the implementation of user interface look and behavior |
Table 7.5: Summary of Results from the Previous Studies on Defect Measures and Size

<table>
<thead>
<tr>
<th>Study</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fenton and Ohlsson [26]</td>
<td>(a) No significant relation between fault density and module size. (b) A weak correlation between module size and the number of pre-release faults. (c) No correlation between module size and the number of post-release faults.</td>
</tr>
<tr>
<td>Ostrand and Weyuker [65]</td>
<td>(a) Fault density slowly decreases with size. (b) Files including a high number of faults in one release, remain high-fault in later releases. (c) Newer files have higher fault density than older files.</td>
</tr>
<tr>
<td>Basili and Perricone [3]</td>
<td>Larger modules are less error prone, even when they are more complex in terms of cyclomatic complexity.</td>
</tr>
<tr>
<td>Shen et al. [78]</td>
<td>(a) Of 108 modules studied, for 24 modules with sizes exceeding 500 LOC, the size does not influence the defect density. (b) For the remaining 84 modules, defect density declines as the size grows.</td>
</tr>
<tr>
<td>Withrow [90]</td>
<td>A minimum defect density for modules with sizes between 161 and 250 LOC, after which the defect density starts increasing with module size.</td>
</tr>
<tr>
<td>Banker and Kemerer [2]</td>
<td>Proposed a hypothesis that for any given environment, there is an optimal module size. For lesser sizes, there is rising economy, and for greater sizes, the economy declines due to rising number of communication paths.</td>
</tr>
<tr>
<td>Hatton [38]</td>
<td>(a) For sizes up to 200 LOC, the total number of defects grows logarithmically with module size, giving a declining defect density. (b) For larger modules, a quadratic model is suggested.</td>
</tr>
<tr>
<td>Rosenberg [76]</td>
<td>Argued that the observed phenomenon of a declining defect density with rising module sizes is misleading.</td>
</tr>
<tr>
<td>Malaiya and Denton [58]</td>
<td>(a) Proposed that there are two types of defects: module-related defects, and instruction-related defects. (b) Module-related defects decline with growing module size. (c) The number of instruction-related defects rises with growing module size. (d) An optimal module size for minimum defect density is identified.</td>
</tr>
<tr>
<td>Current study</td>
<td>No noticeable dependence between defect-based measures (i.e., defect count and defect density) and component size in the considered subsystem.</td>
</tr>
</tbody>
</table>
Chapter 8

A Case Study of Changes and Faults

The case study described in the previous chapter identified a number of mechanisms that gave rise to defects in the studied enterprise resource planning software system. To obtain further insights into the types and the frequency of software faults introduced into the system during the development process, we studied the evolution of the source code modules in the target system spanning a time period of two years from the initial creation of the source code modules to the release of the software product. In this chapter\(^1\), we describe our case study process, and present the frequency distributions of pre-release changes and faults along with lessons learned from the case study.

8.1 Introduction

In the case study presented in this chapter, we perform a retrospective analysis of historical data obtained from a product’s source code repository in order to characterize pre-release change in the product. Source code changes can be characterized by their types, sizes, and frequency distributions, whose properties are well established and described in the statistical literature. Our study takes this approach to address the following

\(^1\)This chapter has been published in [33] and [36].
four research questions about the studied software system. These four questions capture the goal of the study.

1. What is the frequency distribution of the pre-release source code changes in the case under study?

2. What are the sizes of the pre-release source code changes per change type in the case under study?

3. What is the frequency distribution of the pre-release faults in the case under study?

4. How do the results from the present study in the Enterprise Information Systems (EIS) domain compare to the previous studies in other domains?

The answers to the first two questions will provide insights into how effort is spent on different source code change activities during the development process before the product is released to the market. On the other hand, the answer to the third question will help us to identify the dominant classes of faults in the product studied. This information can provide valuable feedback to the organization in choosing the right fault prevention measures that have the highest return on investment for the next releases of the same product or other products within the organization. Furthermore, the distribution of faults can be used to identify problems both in the development process and its product [6] [13]. The answers to these questions can be followed up with further root cause analyses to understand the underlying forces or mechanisms that caused the observed change and fault distributions.

To answer our research questions, we studied the evolution of a set of software modules (i.e., source code files) selected from a commercial Enterprise Resource Planning (ERP) software system. The description of the studied organization and the target ERP product was provided in sections 7.2.1 and 7.2.2. of the previous chapter.

From a research methodology perspective, our study is different from most previous studies in that in contrast to previous work that relied on collecting data from the change
or fault reports recorded by developers and testers, we obtain and analyze historical source code change data from the revision control system, which provides an accurate and comprehensive record of all changes made to the system. Studies that rely on fault reports have expressed several validity concerns regarding the reliability of these reports as a source of data for empirical research. Several of these concerns, including subjectivity, inaccuracy, and ambiguity of the reports, have been discussed by Mohagheghi et al. [62] and Ostrand et al. [66]. The use of the history of the source code changes has increased the validity of the results obtained from the present study. The rest of this chapter describes the case study.

8.2 Case Study Process and Sample Selection

Of the 128 modules comprising the target subsystem, 27 modules were entirely auto-generated and 3 modules were XML configuration files. Since these 30 modules did not contain developer-written source code, we excluded them from the study, which left us 98 modules as the target population to study. Due to the magnitude of the work involved, it would be impractical to analyze all the available data. Therefore, we randomly selected a sample of 10 modules from the target population, and studied the evolution of the selected modules from their initial check-in to the CVS repository to their final release, which spans a period of two years.

Overall, we analyzed 1520 distinct changes distributed over 335 revisions of the selected modules. The average number of revisions per module in the randomly selected sample of modules is 33.5, with a minimum of 9, a maximum of 114, and a median of 21.5. Statistically, this sample size is large enough to enable us to be 95% confident that the margin of error for the change proportions drawn from our sample is no more than 5.4% when the unit of analysis is module revision, and no more than 2.5% when the unit of analysis is individual changes (see Table 8.1).
For each module in the selected sample, we checked out all the revisions of that module from the CVS repository, and using a difference analysis tool compared each revision of the source code module with its succeeding revision. We then classified each identified change, depending on its nature, into one of the following three categories:

- **New Development** - these changes add new functionality to the source code modules. For instance, adding new attributes or methods to Java classes fall under this category.

- **Corrective Change** - these changes deal with rectifying faults found in the source code modules. All bug fixes fall under this category.

- **Behavior-Preserving Change** - these changes do not add to or change the existing behavior of source code modules. Instead, they are behavior-preserving modifications that are primarily meant to make the source code contained within modules more readable and maintainable. Code refactoring, code indentation and layout improvement, code clean-up, and code restructuring, all fall under this category.

After analyzing the frequency distribution of the changes over the three change classes, we further classified each corrective change, based on the type of fault being corrected, into one of the chosen fault classes, and analyzed the frequency distribution of faults over the fault classes. The fault classification scheme used in our study is described in Section 8.3.

Since the classification of changes and faults can be a subjective task and prone to human errors, we implemented a quality control procedure to ensure the quality of the prepared data before performing the analyses. At certain checkpoints, mostly after the examination of every 250 changes by the author of the paper, a member of the development team, who had a thorough understanding of the system, independently classified a random sample of changes and faults. We then compared the results of the two independent classifications. Cases where there were discrepancies between the results
of the two classifications were closely investigated and resolved. In each case, the cause of the discrepancy along with its resolution was recorded, and a consistency check was performed to ensure that the entire dataset is consistent with the agreed resolution.

We followed our case study process by conducting a series of fault root cause analysis group sessions with the project team members. During these sessions, team members discussed the findings of the study and identified the root causes of the fault classes that have a high frequency. To ensure the accuracy of the root cause analysis performed, where necessary, project artifacts such as the requirements specification documents were consulted.

### 8.3 Fault Classes

The fault taxonomy used in our study is partly derived from the ones used in other studies of faults such as [24] and [14]. However, based on the results obtained from a pilot study undertaken as preparation for this current study, we custom defined our fault classification scheme to tailor the needs of the case under study. This approach is consistent with the view expressed by Eldh et al. [24] that different software domains have different sets of faults and fault distributions. In our classification, we used the following 15 classes of faults:

- **Data Fault** - includes defining a variable, attribute, method parameter, or return type to the incorrect primitive or object type.

- **Spurious Fault** - is caused by extra and unneeded code in the software, and the corrective change involves removing one or more source code statements.

- **Fault of Omission** - is caused by missing source code statements that are necessary for the correct behavior of the software. The missing part might be an entire conditional statement, a condition in a compound conditional statement, a method
call, or a computation.

- **Null Pointer Fault** - is caused by an attempt to access a reference that refers to no meaningful object. For instance, calling a method on a null reference causes a run-time error.

- **Algorithm Fault** - refers to incorrect algorithm or program logic. This type of fault occurs when the algorithm encoded in the program is not a correct solution to its corresponding problem. Incorrect program flow and incorrect checking are examples of algorithm fault.

- **Function Fault** - refers to function-related errors such as calling a function with an incorrect parameter, or calling an incorrect function. Note that incorrect arguments passed to functions with generic parameters can not be caught by the compiler.

- **Assignment/Initialization Fault** - refers to incorrect assignment of values to variables.

- **Database Query Fault** - refers to incorrect database query statements. For instance, an incorrect `WHERE` clause in an SQL statement is a database query fault.

- **Empty Array/List Fault** - is caused by an attempt to access elements within an array or list of objects with a size of zero. This type of fault leads to null pointer fault. However, it is not the array or the list itself which is null, but rather the objects contained within them.

- **Computational Fault** - refers to errors in computational expressions within the program code. Incorrect formula or operators are examples of computational fault.

- **Resource Path/Name Fault** - is caused when a protocol, path, or name used within the program code to access an external resource, such as a data or configuration file, is incorrect or malformed.
• **User Interface Fault** - refers to issues related to the look and feel of the system’s graphical user interface. Typical corrections include changes to the user interface component properties such as the size, color, font, and alignment.

• **Language Pitfall** - refers to the incorrect usage of programming language elements. For instance, an incorrect usage of the java assignment operator `=` in place of the logical equality operator `==` in a conditional statement is a language pitfall. In many cases, problems like this cannot be detected at compile time, since both expressions form valid statements in the programming language.

• **Logical Fault** - refers to the errors in the logical expressions within the program code. Incorrect or missing logical operators are examples of logical fault.

• **Empty String Fault** - occurs when the program code incorrectly assumes that a string variable contains a proper value, whereas in fact the string variable is empty (i.e., the size of the string variable is zero).

To ensure that our classification of faults is repeatable (i.e., not dependent on individual opinion), we specified the fault classes precisely so that each identified fault in the considered subsystem falls into only one class. Moreover, we consistently followed specific guidelines to disambiguate cases where a fault might seem to belong to more than one class. As an example, the algorithm fault class, which is commonly found in most fault classification schemes, can be a source of confusion because it usually overlaps with other fault classes within the same classification scheme. For instance, one might reason that an incorrect computation can lead to an incorrect algorithm. In the absence of precise classification rules, it is not clear whether a case like this is a computation fault or an algorithm fault. In employing our fault classification scheme, one rule that we have consistently followed is that we have classified each fault into the most specific fault class in the classification scheme. Therefore, all incorrect computations have been consistently classified as computational faults and not algorithm faults. The algorithm
fault class has been reserved for cases where the program flow is incorrect or the program does an incorrect checking. These cases lead to incorrect algorithms, and are not covered by any other fault class in the classification scheme.

8.4 Analysis

8.4.1 Change Distribution

To answer our first research question, we analyzed the distribution of the source code module revisions and their associated changes over the change types. Table 8.1 presents the results of this analysis. A revision of a source code module is classified as Corrective, if all the changes contained within that particular revision are corrective changes. Conversely, if all changes in a particular revision of a source code module are new developments, the revision is classified as New Development. Revisions classified under the Mixed New Development/Corrective class contain both new developments and corrective changes. Individual changes contained within revisions have a single change type (corrective, new development, or behavior-preserving). Therefore, in Table 8.1, columns corresponding to mixed new development/corrective change type for individual changes are marked as N/A.

<table>
<thead>
<tr>
<th>Change Type</th>
<th>Revision Frequency</th>
<th>% of Revisions</th>
<th>Margin of Error (%)</th>
<th>Change Frequency</th>
<th>% of Changes</th>
<th>Margin of Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrective</td>
<td>164</td>
<td>49</td>
<td>5.4</td>
<td>839</td>
<td>55.2</td>
<td>2.5</td>
</tr>
<tr>
<td>New Development</td>
<td>68</td>
<td>20.3</td>
<td>4.3</td>
<td>337</td>
<td>22.2</td>
<td>2.1</td>
</tr>
<tr>
<td>Mixed New Development/Corrective</td>
<td>67</td>
<td>20</td>
<td>4.3</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Behavior-Preserving</td>
<td>36</td>
<td>10.7</td>
<td>3.3</td>
<td>344</td>
<td>22.6</td>
<td>2.1</td>
</tr>
<tr>
<td>Total</td>
<td>335</td>
<td>100</td>
<td>N/A</td>
<td>1520</td>
<td>100</td>
<td>N/A</td>
</tr>
</tbody>
</table>

As evident from Table 8.1, corrective changes are the dominant class of changes. Over
half of the changes made to the source code modules (55%) are corrections to the existing code, and just about half of the revisions (49%) contain only corrective changes. New developments are well below the corrective changes; only 22% of the changes made to the source code modules are adding new functionality to the software under development. Another 22% of the changes are behavior-preserving changes, that similar to corrective changes, do not contribute new functionality to the system. Instead, these changes are mostly code clean-ups that are performed to facilitate future changes.

The margins of error, at the 95% confidence level, for the proportions of various revision and change types are also given in Table 8.1. For instance, the first row in table 8.1 shows that in a sample of 1520 changes, 55.2% of the changes are corrective and the margin of error is 2.5%. This means that the confidence interval for the proportion of corrective changes ranges from 55.2% - 2.5% at the lower limit to 55.2% + 2.5% at the higher limit. Therefore, if we take other random samples from the considered subsystem, 95% of the times, we can expect to observe a proportion of corrective changes ranging between 52.7% and 57.7%.

The relatively low percentage of changes associated with new developments compared to the other types of changes in the subsystem under study is an indication of the slow pace of the development process from a functional-value perspective. In particular, the ratio of the corrective changes to the new development changes is 2.49, which indicates that with the submission of every new development change to the source code repository, almost two and half new faults have also been injected into the system which had to be corrected at a later revision.

8.4.2 Change Size

Table 8.2 addresses our second research question. It presents the proportions of the sizes of the changes made to the revision types measured in added and deleted lines of code. The choice of added and deleted lines as a measure of change size is consistent with the
way CVS measures and reports the history of changes made to the source code modules. Although only 20% of the module revisions have a mix of new development and corrective changes, they are the most heavily changed modules in the considered subsystem; they account for 47.5% and 43% of added and deleted lines of code, respectively. The majority of changes in these module revisions are new developments. In addition, revisions containing only new development changes account for 20.6% of added lines and 8.7% of deleted lines. These two categories combined indicate that new developments are the largest type of change in terms of the change size.

Corrective changes are the second largest in size; 26.2% of added lines and 39.3% of deleted lines have occurred in corrective module revisions. In comparison to new development and corrective changes, behavior-preserving changes add and delete fewer lines of code; they account for only 5.7% and 9% of added and deleted lines, respectively.

<table>
<thead>
<tr>
<th>Revision Type</th>
<th>% of Lines Added</th>
<th>% of Lines Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrective</td>
<td>26.2</td>
<td>39.3</td>
</tr>
<tr>
<td>New Development</td>
<td>20.6</td>
<td>8.7</td>
</tr>
<tr>
<td>Mixed New Development/Corrective</td>
<td>47.5</td>
<td>43</td>
</tr>
<tr>
<td>Behavior-Preserving</td>
<td>5.7</td>
<td>9</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

### 8.4.3 Fault Distribution

To answer our third research question, we analyzed the distribution of the faults over the fault classes. The results are presented in Table 8.3, which is divided into three
Table 8.3: Distribution of Faults

<table>
<thead>
<tr>
<th>Fault Class</th>
<th>Fault Frequency</th>
<th>% of Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault of Omission</td>
<td>202</td>
<td>24</td>
</tr>
<tr>
<td>Function Fault</td>
<td>184</td>
<td>21.9</td>
</tr>
<tr>
<td>Spurious Fault</td>
<td>132</td>
<td>15.7</td>
</tr>
<tr>
<td>Data Fault</td>
<td>94</td>
<td>11.2</td>
</tr>
<tr>
<td>User Interface Fault</td>
<td>53</td>
<td>6.3</td>
</tr>
<tr>
<td>Algorithm Fault</td>
<td>46</td>
<td>5.9</td>
</tr>
<tr>
<td>Database Query Fault</td>
<td>41</td>
<td>4.9</td>
</tr>
<tr>
<td>Resource Path/Name Fault</td>
<td>37</td>
<td>4.4</td>
</tr>
<tr>
<td>Assignment/Initialization Fault</td>
<td>18</td>
<td>2.1</td>
</tr>
<tr>
<td>Logical Fault</td>
<td>7</td>
<td>0.8</td>
</tr>
<tr>
<td>Language Pitfalls</td>
<td>6</td>
<td>0.7</td>
</tr>
<tr>
<td>Empty String Fault</td>
<td>6</td>
<td>0.7</td>
</tr>
<tr>
<td>Null Pointer Fault</td>
<td>5</td>
<td>0.6</td>
</tr>
<tr>
<td>Computational Fault</td>
<td>5</td>
<td>0.6</td>
</tr>
<tr>
<td>Empty Array/List Fault</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>839</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>
sections: high fault contributors, medium fault contributors, and low fault contributors. The top section in Table 8.3, the high fault contributors, represents fault classes that have a contribution of higher than 10% to the total number of faults in the considered subsystem. Faults of omission, with a share of 24%, are the largest class of faults found in the case study. Function faults (21.9%), spurious faults (15.7%), and data faults (11.2%) are the next largest classes of faults, respectively. Using input from project team members and project data, we performed root cause analysis of high fault contributors, and found that the root causes for these four fault classes, which collectively account for 72.8% of the faults in the considered subsystem, can be largely traced back to upper stream development activities such as requirements and design.

The middle section in Table 8.3, the medium fault contributors, represents classes that have a fault contribution between 1 and 10 percent. These fault classes together represent 23.6% of the faults in the considered subsystem.

The six fault classes listed in the bottom section of Table 8.3, the low fault contributors, have individual fault contributions of less than 1%. These classes are mostly pure programming errors, and together account for 3.7% of the total number of faults. 23.5% of all the faults could have been prevented by enforcing generic or project-specific design and implementation rules.

8.5 Results and Lessons Learned

Based on the analyses performed, the major conclusions and contributions of our study are as follows:

1. A significant portion of the pre-release changes made to the software modules do not add functional value to the system. This finding is supported by the evidence that 77.8% of the total number of changes in the considered subsystem are either corrective or behavior-preserving; only 22.2% of the changes involve new develop-
ments. Moreover, we observed that, on the average, every submission of new code to the source code repository contained two and half faults. These faults had to be detected, fixed, and verified at a later time, which imposed high costs on the development process.

2. Most revisions to source code modules are made with a single purpose in mind (i.e., adding new functionality, bug fix, or code clean-up). This finding is supported by the evidence that 80% of the source code module revisions contain changes of only a single type; only 20% of the revisions have mixed changes.

3. At least 26.2% of added lines and 39.3% of deleted lines are caused by corrective changes.

4. Of the 15 fault classes identified in the subsystem under study, faults of omission, function faults, spurious faults, and data faults are the major fault contributors. These four fault classes together account for 72.8% of corrective changes. Faults of omission (24%) and spurious faults (15.7%) can be largely traced back to the requirements phase, whereas function faults (21.9%) and data faults (11.2%) are mostly rooted in the design of the system. The high concentration of faults in these four fault classes indicates that upper stream specification-related issues (e.g., incomplete, incorrect, or unclear requirements and design specifications) are the dominant sources of faults. This finding suggest areas where effort should be directed.

5. 23.5% of the faults could be prevented by conforming to design and implementation rules. These faults were caused by the violation of rules regarding the proper use of the language constructs, project-specific implicit programming rules, data and interface compatibility rules, and user interface consistency rules.

6. In contrary to our expectation, language-related faults including logical faults, lan-
guage pitfalls, empty string faults, null pointer faults, computational faults, and empty array/list faults have a share of only 3.7% of the total number of faults. In spite of the low frequency distribution of the fault classes in this category, most automated fault detection tools are focused on this category. This finding suggests a need to shift the focus in tool support from low frequency fault classes to classes with the highest fault frequency.

7. A number of the results from the studied ERP system are consistent with findings from other types of software system, which is an indication of the common forces or mechanisms that underlie the development of software systems. Some of these common findings include: (a) corrective changes are the largest class of changes among all change types (b) most faults are introduced in the specification phases of the software development (c) in terms of the change size, new developments are largest in size, followed by corrective changes and behavior-preserving changes, respectively. In the related work section of this chapter, we have discussed and compared results from some of the previous studies.

8.6 Threats to Validity

Several factors potentially affect the validity of our findings. We discuss these factors under standard types of validity threats in empirical studies.

8.6.1 Construct Validity

We use fault frequency as a surrogate measure for the quality of the software modules. Fault frequency is widely used in software engineering studies. The purpose of the study was to understand the nature of the changes made to the software modules during their evolution from the initial creation to the final release. In particular, we were interested in identifying the fault contributions of the various classes of faults based on the corrective
changes made to the various revision of the software modules during their evolution. To answer our research questions, we analyzed the distributions of changes and faults in a randomly selected sample of source code modules from the ERP subsystem under study. Therefore, the variables measured and studied, truly reflect the purpose of the study.

8.6.2 Internal Validity

In our study, the incorrect attribution of faults to fault classes could be a threat to the internal validity of our findings. To minimize the potential effect of this confounding factor, we followed a quality control procedure, as described in Section 8.2, to ensure the quality of the collected data. The involvement of the members of the project team in the case study process has also increased the internal validity of our findings. We used a difference analysis software tool to accurately identify the changes between the consecutive revisions in the source code modules. We believe that these measures have collectively helped us to achieve an acceptable level of internal validity.

8.6.3 External Validity

Our dataset was taken from one product of a development organization, which can be a limit to the generalizability of our findings.

8.7 Related Work

Post-release (i.e., maintenance) changes to software systems have been frequently studied in the past decades. As a result, the nature of these changes and their distributions over the various maintenance activities are well known. Software maintenance, based on the types of post-release changes involved, is divided into four broad categories of activities, namely corrective, adaptive, perfective, and preventive. These activities account for 21%, 25%, 50%, and 4% of the total maintenance efforts, respectively [56] [57] [88]. These
figures are based on studies of a large number of software projects and are believed to represent the status quo in the software industry. In comparison, fewer studies have investigated pre-release changes.

Basili et al. [4] monitored the changes made during the development of an Ada project, which involved the redesign and reimplementaiton of a portion of a satellite ground control system, and observed that requirements, design, and code changes accounted for 7%, 32%, and 61% of the changes. In addition, 57% of the changes were fault corrections, and 23% were improvements for clarity, maintainability and readability.

Mockus and Votta [60] analyzed the modification requests in a large real-time software system and found that between 34% and 46% of all changes were corrective. They report 45% for adding new feature, 3.7% for restructuring the code to accommodate future changes, and 5.3% for inspection changes, which usually have both corrective changes and code restructuring. In terms of the size of the changes, corrective changes account for 18% to 27% of all added and deleted lines and are the second largest only after new features, which account for 63.2% of the added lines and 55.7% of the deleted lines. Code restructuring accounts for only 3.5% and 5.8% of lines added and deleted, respectively.

The results from the change size analysis in our study are consistent with change sizes reported by Mockus and Votta’s study. In our study, too, we observed that changes related to new developments are the largest in size, followed by corrective changes; Behavior-preserving changes were the smallest in size. Another finding of their study is that, in terms of change difficulty, corrective changes tend to be the hardest.

Both of these studies rely on change reports to collect change data. In contrast, we obtained the history of changes from the source code revision control system. This approach allowed us to study the evolution of the source code modules at a fine-grained level. Our study is consistent with these two studies in that corrective changes have been unanimously identified as the largest class of changes among all change types.

Eldh et al. [24] studied and classified the failures in a large complex telecommunica-
tion middleware system. They concluded that (a) faults of omission are the dominating class of software faults (b) faults of omission (38.3% of software faults), together with spurious faults (8.3% of all software faults) indicate that faults related to unclear specifications dominate among the real software faults in the telecom system studied. In spite of the fact that our case study was performed in a different application domain, results from our study support both of their findings. Our study identified the faults of omission as the dominating class of faults in the enterprise resource planning system studied. In our study faults of omission (24%) together with spurious faults (15.7%) account for 39.7% of all the faults found in the considered subsystem. The strong fault contribution of these fault classes together with function faults and data faults in our study, also confirms that the great majority of faults originate in specifications. They report that 12.7% of the faults in their case were data faults, which is close to the 11.2% observed in our study. In contrast to the relatively small share of function faults in their telecom system (6.4%), we observed that 21.9% of the faults in our ERP system are function faults.

The study reported by Borretzen and Dyre-Hansen [10] is similar to ours in intent: they performed fault analysis to improve software quality. They investigated the fault profiles of five industrial Java/J2EE projects to understand what types of faults are dominant and what types of faults are reported to be more severe. They used a revised version of the Orthogonal Defect Classification (ODC) proposed by Chillarege et al. [14] to classify faults in the systems under study, and found that function faults were not only the dominant fault type, but also the most numerous of the critical severity rated faults. These results are consistent with their previous similar study of faults [9]. Since in ODC [14] function faults are associated with the design process, they proposed that the organization studied should increase effort in the design phase of the development in order to improve software quality.

The results of our study contradict some of their findings. First, whereas in the ERP system we studied a significant portion of faults were originated in the requirements
phase of the development process, requirement-related issues are not an important cause of faults in the systems they studied. Second, they report that graphical user interface faults, with a share of 19.5% of the total number of faults, are the second largest class of faults in the projects they studied. This result is in contrast to the ERP system we studied, where user interface faults account for only 6.3% of the faults.

Although the projects they studied are business-critical systems and, similar to ours, are developed in Java/J2EE platform, some of their results are not directly comparable to ours. This is because the classes of faults used in their study are, for the most part, different from ours. From a research methodology point of view, our data collection is different from theirs in that they rely on the textual descriptions in fault reports provided by developers and testers to classify faults. In comparison, we relied on the history of corrective changes obtained from the source code revision control system of the product under study.

8.8 Conclusion

In this chapter, we have reported on a study of pre-release source code evolution from a commercial enterprise resource planning software system. Historical change data from the revision control system have been classified and analyzed according to our research questions. The numerical details of the distributions of changes and faults in the studied system have been presented, and compared to the distributions reported in previous studies in order to obtain insight into the development process. The results clearly indicate the importance of fault prevention in reducing the development costs.

We found that fault corrections, with a share of 55.2%, are the largest class of pre-release changes made to the source code modules in the product under study. Of the 15 classes of faults used to classify these corrective changes, four fault classes namely, fault of omission, function fault, spurious fault, and data fault are the strongest fault contributors,
and together account for 72.8% of the corrective changes. The predominance of these types of faults in the considered subsystem is an indication of the specification-related problems in upper stream development activities such as requirements and design.

We also found that violation of design and programming rules are the cause of a sizable portion of the faults. The software construction approach introduced throughout this thesis is essentially rule-based. The rules that are known to prevent software faults can be naturally incorporated into the rule templates that drive the construction of the system. This means that, our rule-based approach to software construction not only helps to achieve traceability, but also reduces the rate of faults in software systems.
Chapter 9

Conclusion

Inadequate trace links between various artifacts created during the software development life cycle results in a fundamental problem in software engineering, which is known as the Traceability Problem. This problem exhibits its negative effects throughout the various stages of the software development process, most notably the software maintenance phase. It has been frequently mentioned in the software engineering literature that maintenance costs are the dominant costs of software systems. Given the large size and the complexity of today’s software systems, software maintenance, without traceable artifacts, is challenging. This maintenance problem clearly indicates the importance of addressing the traceability problem.

Unfortunately, achieving traceability in large-scale systems has proved to be extremely time-consuming, error-prone, and costly. Therefore, the development of more affordable solutions to traceability is a top priority in the software traceability research. The contribution of this thesis to the research area of software traceability is a design-rule-based approach to software construction that provides effective and affordable traceability. The following section explains our research contributions in more detail.
9.1 Contributions

- **Achieving traceability by construction** - we introduced a process for rule-based software construction that provides effective traceability, and through several empirical studies, demonstrated that the process is feasible. Existing approaches to traceability are either cross reference-centered or document-centered. In cross reference-centered techniques, cross references (e.g., unique identifiers) are embedded inside the project artifacts. In more comprehensive traceability schemes, these artifacts are then supplemented by traceability matrices or tables that keep track of cross references. In document-centered techniques, the traceability scheme dictates parts or all of the structure and content of the project documentations to ensure traceability. In large systems, reference-centered approaches generate large amounts of trace information that has to be continuously updated as the system evolves. On the other hand, document-centered approaches make the development process more heavy weight. Our constructive approach avoids many of the problem associated with existing traceability approaches, while providing the same benefits. To the best of our knowledge, the work presented in this thesis is the first full-fledged constructive approach to traceability.

- **Provision of affordable traceability** - a characteristics of existing traceability approaches is that they provide external traceability in the sense that traceability, in these approaches, is not viewed as an internal characteristics or property of the systems’ design or source code. This means that the design or the source code of the system is not traceable in its own rights. Instead, traceability is achieved outside the design and the source code by creating external maps of the system, such as traceability matrices or tables, that attempt to capture trace links between traceable entities in the system artifacts. In contrast, our approach to traceability is based on the notion of internal traceability as described in Chapter 3 of this
thesis. In our approach, traceability is viewed as an intrinsic property of the structure of software systems. We developed a rule-based approach to traceability that subscribes to this philosophy, and demonstrated that such an approach provides just-in-time traceability, eliminating the costs associated with the up front collection of trace information in conventional traceability approaches. In our approach, the high-level traceability rules that underlie the structure of the system are used to derive low-level concrete trace information prior to performing change impact analysis or maintenance tasks. Our investigations in the domain of business software systems show that a small number of such rules are enough to achieve traceability in these systems, which makes the rule-based approach to traceability more affordable than conventional approaches.

- *Introduction of parameterized rule templates as Traceability Patterns*
  - we introduced parameterized requirement-component rule templates as a convenient format for documenting the positional, structural, and behavioral design rules that ensure traceability of categories of requirements to their corresponding components. Since these templates are essentially rule-based, other types of design rules, such as the ones that contribute to achieving other non-functional or quality requirements such as maintainability, reliability, etc., can be seamlessly incorporated into the rule templates, making them a complete solution to each of the identified requirement types. In this sense, our approach is in accord with the widely-adopted definition of a pattern in the literature as a generic solution to a recurring problem in a context. Each traceability pattern serves as a protocol for transition from software requirements to source code. The set of all traceability patterns that are applicable to a specific type or family of systems together form a traceability catalogue. These catalogues are composed of a set of problem types (e.g., a requirements taxonomic scheme) along with their solutions (i.e., the specifications of the corresponding generic components in terms of the design constraints
or rules). For instance, a domain-specific catalogue can be designed for Enterprise Information Systems (EIS). Traceability catalogues can be standardized within development organizations and shared across multiple projects in the same domain as reusable artifacts. The traceability catalogues not only enable a template-driven approach to software construction, but also promote the sharing and reuse of development knowledge across the various software projects.

• **Provision of an approach for traceability in agile development processes** - the wide industry adoption of agile development methodologies in the recent years has posed a particular challenge to the applicability of conventional traceability approaches. A characteristic of agile methodologies is that requirements are largely communicated through informal channels, such as discussions with an on-site customer, rather than more formal requirements specification documents. The problem arises from the fact that a prerequisite to conventional requirements traceability approaches (e.g., matrix-based approaches) is the existence of a static medium such as a requirements management tool or requirements specification document, with unique identifiers assigned to individual requirements, that will contain the requirements throughout the life cycle of the project. The informal nature of requirements in agile development methodologies does not satisfy this basic assumption that existing traceability approaches rely on. Consequently, the applicability of existing traceability approaches to agile software projects has been restricted. Both cross reference-centered and document-centered approaches achieve traceability by creating a more comprehensive set of documentation for software projects. This emphasis on documentation is in sharp contrast with the agile philosophy, where emphasis is placed on the production of source code, and not documentation. Our design-rule-based approach to traceability focuses on the structure of the source code, which is the main artifact in agile development methodologies. Therefore, our constructive approach to traceability is a natural fit for agile environments.
• **Identification of defect introduction mechanisms in software systems** - the goal of this thesis was to help to reduce the effort spent on corrective changes in software systems. We pursued two strategies to achieve this goal: (a) effective and affordable traceability (b) defect prevention. Defect prevention is a possible way not only to reduce corrective maintenance costs, but also to produce more reliable systems. However, to devise tools, techniques, and processes to support defect prevention requires an understanding of the mechanisms that give rise to defects during the development of software system. To gain such an insight, we conducted and contributed two industrial case studies of defects to the literature on software defects. Through these case studies, we identified a number of mechanisms that give rise to defects in software systems. We further, provided analyses of the types and frequencies of the software changes and faults. Based on the profile of defect sources identified in our case studies, we propose a set of mitigation strategies that software development organizations can employ to counteract these defect introduction mechanisms.

• **Reduction in defect rate through the rule-based software construction** - we provided empirical evidence obtained from the industrial case studies of defects to demonstrate that a sizable portion of the faults are caused by the violation of rules regarding the proper use of the language constructs, project-specific implicit programming rules, data and interface compatibility rules, and user interface consistency rules. As noted earlier, the software construction approach introduced in this thesis is essentially rule-based. The rules that are known to prevent software faults, such as the ones discovered through our case studies, can be naturally incorporated into the rule templates that drive the construction of the system to counteract some of the mechanisms that give rise to defects in software systems. This means that, in addition to traceability, our rule-based approach to software construction helps to reduce the rate of faults in software systems, and hence contributes toward
building more reliable systems.

9.2 Future Work

The work presented in this dissertation can be extended in several directions. In what follows, we provide suggestions for future work. As mentioned in Chapter 4, the rule-based development process developed in this thesis leads to a number of important implications in software systems. Throughout this thesis, we focused on two of these implications namely, the traceability of software systems and the reduction in defect rate. Through a series of studies, we provided empirical evidence to support these two implications. As future work, it would be interesting to design and conduct further empirical studies to investigate each of the following areas, which we believe are the other implications of the proposed rule-based approach to software construction. Empirical evidence obtained from such studies can help to fill the gap between our intuitive beliefs and the underlying realities.

- Economy of Specification - the set of design rules that govern a software system can be regarded as a partial specification for that system. In contrast to conventional approaches to software specification, where each entity in the system needs to be specified individually, in a rule-based software system, each regular subset of the system can be specified by the rule set that it conforms to. This means that a single rule set is capable of specifying some aspects of a group of software artifacts, thus providing an economy of specification.

- Facilitating Software Comprehension - conformance to a set of design rules makes it easier to understand software systems. This is because in a rule-based software system, the problem of software comprehension is partly reduced to understanding the rules that govern the system. Comprehension has been defined as "establishing a link between what you see and something you’ve seen before" [85]. A fundamental
assumption of cognitive theories is that learning is a process of relating new information to previously learned information. Accordingly, an implication of cognitive theories is that new information is most easily acquired when people can associate it with things they have already learned. A rule-based source code can be viewed as frequent repetitions (i.e., instances) of a set of rules. Assuming that developers are familiar with the rules that govern the system, they can relate a large part of the source code to their corresponding rules. Therefore, once developers learn the set of rules that govern a system, they should be able to understand large parts of that system.

• Promoting Software Quality - in a rule-based software system, solutions to each category of development problems (e.g., the architectural, design, and programming rules that should govern the implementation of a particular category of functional requirements such as business rules) are repeatedly applied to each of the problems in the same category hence promoting design uniformity as well as solution reuse. Moreover, since solutions assigned to problem categories represent the state of the art in addressing the types of problems represented by the categories at the time of the assignment, they improve the quality of the software system.

• Shifting From Creative to Routine Development Tasks - in a rule-based approach to software construction, the creative design process is mostly limited to only the initial phases of the development process where the rule sets for the software system under development are defined. This is in contrast with conventional opportunistic software development approaches where the entire development process is regarded as a creative process. Once rules are defined to regulate the various areas of the system, the rest of the development process can be considered as a routine task. This can help to improve the productivity of the development process.

• Promoting Systematic Software Development - in a rule-based approach to software
construction, the development process is more systematic. This is in contrast with current widely-used opportunistic approaches to software development, which are strongly dependent on human side of software engineering such as the knowledge and experience of the developers. Software engineering best practices can be encoded in the form of a set of development rules. These rules can then be shared among various projects.

- **Predictability** - since in a rule-based approach to software development all similar situations (e.g., requirements of the same type or repeating design problems) are addressed through similar solutions (i.e., a set of rules that prescribes a uniform solution for a class of problems), the resulting software system becomes more predictable. This predictability can then be used to facilitate reasoning over the system’s source code, which is necessary when one performs various software engineering tasks such as software comprehension, change impact analysis, and maintenance.

In addition to verifying the validity of these propositions, the followings are other possibilities for future work:

- **Automation of the rule compliance checking** - the rule-based approach described in this thesis can be tool supported. Automatic detection of rule violations in a system’s source code can reduce the cost of verifying that developers are actually following the rules.

- **Identification of the dominating requirement types in other domains** - as mentioned in Chapter 4, frequently occurring functional requirement types are excellent candidate areas for regulation. In this thesis, we were mostly focused on the domain of business applications. We inspected several public and private commercial requirements specification documents for enterprise information systems and as a result identified the dominating classes of requirements in these systems. As future work,
it would be interesting to inspect requirements specification documents from other domains (e.g., scientific or embedded systems) in order to identify candidate areas for regulation in these domains. These studies will allow us to compile, share, and reuse domain-specific rule catalogues.

- Tool support for fault classes with higher frequencies - in our study of software faults, we observed that in spite of the low frequency distribution of the language-related faults, most automated fault detection tools are focused on this class of faults. As future work, it would be interesting to investigate the feasibility of building automated tools to support the detection and prevention of fault classes with higher frequencies such as function and data faults.

9.3 Closing Remarks

The research work presented in this thesis contributes to the field of software engineering by introducing a novel approach to the traceability problem, as well as addressing several related research challenges. We have presented experimental data that demonstrates that our approach is useful and effective in addressing the traceability problem.

We hope that this thesis will encourage researchers in software traceability to focus on traceability as an intrinsic property of the structure of the software system and develop techniques that will be helpful to the development and maintenance of large-scale software systems.
Bibliography


