1-1-2014

Toward an Effective Automated Tracing Process

Anas Mohammad Mahmoud

Toward an effective automated tracing process

By

Anas Mohammad Mahmoud

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2014

Toward an effective automated tracing process

By

Anas Mohammad Mahmoud

Approved:

Nan Niu
(Major Professor)

Eric A. Hansen
(Committee Member)

David A. Dampier
(Committee Member)

Edward B. Allen
(Committee Member
and Graduate Coordinator)

Jason M. Keith
Interim Dean
Bagley College of Engineering

Name: Anas Mohammad Mahmoud

Date of Degree: May 16, 2014

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Nan Niu

Title of Study: Toward an effective automated tracing process

Pages of Study: 217

Candidate for Degree of Doctor of Philosophy

Traceability is defined as the ability to establish, record, and maintain dependency relations among various software artifacts in a software system, in both a forwards and backwards direction, throughout the multiple phases of the project's life cycle. The availability of traceability information has been proven vital to several software engineering activities such as program comprehension, impact analysis, feature location, software reuse, and verification and validation (V&V).

The research on automated software traceability has noticeably advanced in the past few years. Various methodologies and tools have been proposed in the literature to provide automatic support for establishing and maintaining traceability information in software systems. This movement is motivated by the increasing attention traceability has been receiving as a critical element of any rigorous software development process. However, despite these major advances, traceability implementation and use is still not pervasive in industry. In particular, traceability tools are still far from achieving performance levels

that are adequate for practical applications. Such low levels of accuracy require software engineers working with traceability tools to spend a considerable amount of their time verifying the generated traceability information, a process that is often described as tedious, exhaustive, and error-prone.

Motivated by these observations, and building upon a growing body of work in this area, in this dissertation we explore several research directions related to enhancing the performance of automated tracing tools and techniques. In particular, our work addresses several issues related to the various aspects of the IR-based automated tracing process, including trace link retrieval, performance enhancement, and the role of the human in the process. Our main objective is to achieve performance levels, in terms of accuracy, efficiency, and usability, that are adequate for practical applications, and ultimately to accomplish a successful technology transfer from research to industry.

Key words: Traceability, Information Retrieval, Indexing, Semantics, Clustering, Refactoring, Information Foraging

DEDICATION

To family and friends.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Nan Niu. I am very grateful for your support, encouragement, and guidance throughout the past four years. Your enthusiasm in research has always been my main motivation to continue this work. Thank you for supporting me intellectually and financially and for believing in me during the times I doubted myself. I am deeply grateful for the long discussions we have had at your office, and for proofreading and commenting on countless revisions of my papers. None of this would have been possible without you.

My cordial thanks also go to my dissertation committee members, Dr. Edward A. Allen, Dr. Eric A. Hansen, and Dr. David A. Dampier for their input on this work. Dr. Allen, I still remember walking into your office on my first day of graduate school asking for help. Since then, your office has always been the first place I would stop by whenever I needed help. You have always been there for me, a father figure and a mentor, and I am greatly thankful for that.

My sincere thanks also go to Dr. Gary Bradshaw. Thank you for your insightful comments and constructive criticisms during our weekly brief meetings.

I'd also like to express my deepest appreciation to my research brothers and fellow doctoral students, Sandeep Reddivari and Tanmay Bhowmik. Thank you for the stimulating discussions, for the times we were working together, and for all the fun we have had in the

past four years. We took this journey together and have supported one another every step of the way and I could have not asked for a better company.

I am also grateful to the following former or current staff at the Department of Computer Science for their various forms of support during my graduate study. Shonda Cumberland, Lesli Hutchins, Nicole Ivancic, and Courtney Blaylock. I would like also to acknowledge Mamdouh Barakat and Gibran Hamed from MB Risk Management. My experience as a software engineer at MBRM has always been on my side while conducting this work. Thank you for sharing your knowledge and wisdom with me.

I am so blessed to have a wonderful family that unconditionally supports and loves me. I'd like to thank my father, Mohammad, my Mom Fa'eda, my sisters Hiba and Bayan, my bothers Bilal, Ahmad, and Moad, and my in-laws Fadia and Hani. Without your love, patience, encouragement, and support I would not be where I am today.

At last but not least, I would like to thank my wonderful friends Salah, Abdullah, and Muhammad Dakhlalla, Claire Johnston, Ruth Davis, Cara Prather, Gaiya Yu, Laci Kyles, Trent Ricks, Kevin Roscoe, Athénaïs Boëffard, Srishti Srivastava, and Puntitra Sawadpong. My buddies, Christopher Hicks, Jonathan Tyler Dobbs, Taylor Morris, Shann Moore, Josh Jarriel, Ibrahim Gaber, Youssef Hammi, and Osama AbuOmar. Thank you for your support and care. I greatly value your friendship and I deeply appreciate your belief in me.

I love y'all,

Nash.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This chapter defines traceability, its benefits and applications, and briefly reviews the state-of-the-art in establishing and maintaining traceability in practice. In addition, in this chapter we describe our main research objectives, associated research questions, and experimental settings.

1.1 Software Traceability: A Definition

Traceability has been defined in the literature in various ways. Earlier views on traceability can be traced back to 1978. In particular, Greenspan and McGowan [102] defined traceability as "a property of a system description technique that allows changes in one of the three system descriptions - requirements, specifications, implementation - to be traced to the corresponding portions of the other descriptions. The correspondence should be maintained through the lifetime of the system". In 1984, the term *traceability* appeared in the IEEE Guide to Software Requirements Specifications (IEEE-830) as a technique for "enabling the traceability of a requirement back its origin, and facilitating the referencing of each requirement in future development or enhancement documentation" [122]. In this definition, the notion of bidirectional traceability (backward and forward) was introduced. This notion later appeared in the definition coined by Gotel and Finkelstien in 1994, who

defined traceability as "the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)" [97].

A more recent definition in the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std-91) described traceability as "(1) the degree to which a relationship can be established between two or more products of the development process, and (2) the degree to which each element in a software development process establishes its reason for existing" [123]. In addition, in 2004, Spanoudakis and Zisman defined software traceability as "the ability to relate artifacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other" [231]. Other similar definitions can be found in [71, 110, 215].

In general, majority of the proposed definitions in the literature share the key elements of artifact, traceability relation (link), and link direction. These elements can be described as follows:

1.1.1 Artifact

An artifact is any atomic entity or element of a software system (e.g., a requirement document, a design document, or a source code file) or even entities at lower granularity levels (e.g., a requirement statement, a design component, or a code class or method) [215].

1.1.2 Relation

A traceability relation is any association that can be established between any two artifacts of a software system. Such relations, also known as *links*, can be one-to-one, one-

to-many, or many-to-many (i.e., an artifact can be traced to several other artifacts, or two artifacts can have more than one traceability link connecting them). Several traceability relations have been identified in the literature [231], such relations can be described as follows:

- Dependency: This relation refers to situations where the existence of a certain software artifact depends on another artifact in the system. For instance, an implementation relation, where a certain source code module implements a certain functional requirement, can be described as dependency relations.
- Generalization: Such relations are often used to describe complex and compound structures in the system. For instance, if a certain artifact can be broken down into multiple other artifacts at lower granularity levels (e.g., considering the individual member functions of a class), or a set of artifacts can be combined into a single artifact at higher level of abstraction (e.g., combining individual requirements to produce a requirements document).
- Evolution: This particular relation describes the traceability between artifacts from different releases of the system. The main objective of maintaining evolution relations is to keep track of changes in the system overtime.
- Satisfaction: A satisfaction traceability relation can be describes as follows: an artifact e_i satisfies another artifact e_j if and only if e_i complies with a condition imposed by e_j. For example, when a certain design case satisfies a certain condition imposed by a certain requirement.
- Overlap: Two or more artifacts can be described as overlapping if they refer to, describe, or implement a common concept or a feature of the system. For example, if a certain test case is used to test two different features of the system then these two features can be considered overlapping.
- Conflicting: Relations of this type capture potential conflicts in the system (e.g., conflicting specifications).
- Rationalization: These relations are used to describe the rationale behind the creation and evolution of certain artifacts. For instance, a rationalization relation can be established between a test case and the specifications that led to this particular test.
- Contribution: Such relations are established to keep track of the origin of artifacts. For instance, a certain feature is contributed by a certain stakeholder, or a certain code module is developed, or being maintained by a certain software developer.

1.1.3 Direction

Directionality of traceability links implies that such links should be established throughout the system's life cycle, linking artifacts in both forward and backward directions from their origins at initial phases to their subsequent releases at later phases of the development cycle and vice versa.

Based on this review, in this dissertation, we define traceability as "the ability to establish, record, and maintain relations among the various artifacts in a software system in both forward and backward directions throughout the system's life cycle".

1.2 Why Traceability?

Establishing and maintaining traceability information is vital to several software engineering activities such as program comprehension, Verification and Validation (V&V), software reuse, feature location, impact analysis, reverse engineering, and many other activities related to software maintenance and development. In what follows we describe these applications and the potential benefits different traceability relations might provide to support their operation.

1.2.1 Program Comprehension

Program comprehension is the task of understanding a software program. Comprehension is a cognitive process that starts with a hypothesis in mind. A bottom-up [205], a top-down [30], or a combined strategy [155] is then used to verify that hypothesis. The availability of traceability information among various software artifacts reduces the amount of time required to comprehend the system. For instance, in top-down comprehension,

traceability links start from the initial comprehension hypothesis, down to other system's artifacts, tracking down concepts that either confirm or refute the comprehension hypothesis. In contrast, in the bottom-up strategy, starting from the system artifacts at lower granularity levels, traceability links are used to gradually increase the abstraction level until a comprehension hypothesis or an understanding of the system is formulated [11, 58, 174].

1.2.2 Verification and Validation (V&V)

Verification and Validation V&V is the process of checking that a software system meets its requirements [193]. Verification is concerned with building the product right, from that perspective, traceability relations such as *satisfiability*, are used to establish the confidence that work products meet the development standards at each phase. Validation, on the other hand, is concerned with building the right product, in that sense, traceability links such as *dependency*, provide a formal proof that the system actually implements the desired set of requirements [121, 231] and ensure that test cases have been developed to validate that all the requirements have been implemented [63, 249].

1.2.3 Software Reuse

Software Reuse is the process of creating software systems from existing software rather than building them from scratch [33, 142]. Reuse is not limited to source code, basically any part of a software system can be reused, such as specifications [125], test data [162], and even design knowledge [251]. To that end, traceability information facilitates an effective reuse process by saving cognitive efforts while recognizing and retrieving the set of potentially reusable components in the system [10, 88, 215, 250].

1.2.4 Impact Analysis

Impact Analysis is a critical software activity that is concerned with understanding the full extent of a proposed change request [14]. Tracking the impact of changes in a software system is often described as time-consuming and error-prone process as a certain change might affect several components of the system [158]. To assist in this process, traceability links in the system are used to follow the *ripple effect* of a particular change [91]. In particular, identifying dependency relations among various system artifacts helps to interpret the nature of the impact and assess its full effect [249]. For instance, changes can be modeled as events that are propagated through traceability links according to a set of predefined propagation rules to finds all potentially impacted components [213, 249].

1.3 Generating Traceability

Traceability is achieved in practice in various ways. In general, methods for establishing and maintaining traceability can be classified, based on the level of automation they adopt, into three main categories including: manual, semi-automatic, and fully automated methods [215]. The following is a description of these categories in greater detail.

1.3.1 Manual Traceability

Under this approach, traceability links are established and maintained manually through a Traceability Matrix (TM) that links various artifacts in the system. A simple TM can be visualized as a table in which rows and columns represent the system's artifacts. An entry in the table indicates a traceability relation between the two artifacts at the row and column ends of that entry. Table cells may include more information to convey other traceabil-

ity attributes such as the type of the link and its direction. The process of building and maintaining a TM manually can be described as follows:

- Link discovery: Refers to the process of identifying traceability links among various artifacts in the system. This process starts from the initial phases of the development process, such as establishing *contribution* links between requirements and stakeholders during the requirements elicitation phase, and continues throughout the project's life cycle, including later phases of software testing and maintenance.
- Recording: Refers to storing or documenting traceability links. This process can be handled using basic tool support such as a word processing tool or a spreadsheet.
- Maintenance: As software systems evolve overtime, their internal structure tend to change, as a result, traceability links get outdated [153]. Therefore, analysts maintaining a manual TM have to perform several maintenance tasks (e.g., adding, deleting, and updating links) to reflect these changes in the system, thus making sure that the system's traceability links are up-to-date.

Several third-party requirement management tools are available to facilitate the manual tracing process. Such tools (e.g., DOORS [1], RTM [2], RDT [3]) depend on the availability of traceable references in the system to generate traceability links. These references are often implanted manually through a keyword assignment process. Under this process, a developer intentionally leaves marks, or signs, that can be automatically translated into traceability links. To help the tool follow these links, a glossary of these implanted signs is usually provided. The tool then scans the text for these signs and generates the associated tracks or links [98]. In addition, such tools provide several automated options for recording, editing, and displaying links using various visualization and navigation techniques. However, even with the automated support such tools provide, they are still considered manual in the sense that they just follow manually assigned references.

[1] http://www-142.ibm.com/software/products/us/en/ratidoor/
[2] www.chipware.com
[3] http://www.ccs.neu.edu/home/home/lpb/mud-history.html

Manual traceability represents a practical solution for relatively small systems. However, as software systems evolve over time, this process becomes labor-intensive, boring, time-consuming, and error-prone [63, 97, 98, 121]. Therefore, an effective traceability is rarely established manually in practice [44, 47, 216]. Motivated by these observations, researchers have started looking for other methods that bring a degree of automation to the process. The main assumption is that automating traceability should go beyond recording and replaying traceability links to recovering and maintaining these links [75]. Following is a description of these methods.

1.3.2 Semi-automated Traceability

To reduce the effort associated with the manual approach, semi-automated methods exploit various techniques, such as execution traces and graph models, to derive traceability information in a semi-automated manner. A plethora of such methods have been introduced in the literature. Example of such methods include:

1.3.2.1 Rule-based Traceability (RBT)

This approach uses observations about the runtime behavior of the system to detect associations among its functional scenarios and their executing code. In particular, traceability links are captured from the data flow between the system's modules in the form of a footprint graph, where a footprint is the set of lines of code used to execute a certain scenario. A set of rules are then used to reinterpret this graph to yield new traceability information. Examples of such rules include (a) transitive reasoning, if there is a dependency relation between artifacts A and B and a dependency relation between B and C, then

there is a dependency relation between A and C, and (b) sharing of a common ground: a traceability link exists between A and B if their implementations (code) overlap [72]. Other rules can be found in [167, 232].

1.3.2.2 Process-driven Traceability (PDT)

Using PDT, traceability links are captured as a result of the software development. In particular, a special purpose software is used to monitor the software development process and capture traceability links which result from specific development activities [209]. Such activities include creating, deleting, and modifying system artifacts in response to certain code changes [67].

1.3.2.3 Event-based Traceability (EBT)

EBT establishes loosely coupled traceability links by using publish-subscribe relationships between dependent objects in the system. In participial, under this approach, artifacts in the system use an event server to subscribe to the requirements on which they are dependent [108], when a change event (e.g., a requirement merge, replacement, refinement, or abandonment) occurs an event notification message is published to all the subscribed dependent objects. Therefore, ensuring that all these publish-subscribe relations (trace links) are up-to-date or consistent with such changes. This type of traceability scheme is designed to handle both long-term and short-term changes [44].

1.3.2.4 Policy-based Traceability (PBT)

Such approach is used for automatically updating traceability links every time an architecture or its code base evolves, where traceability links in the system are continuously updated in response to changes. The specific update to be made is determined by an actively specified set of traceability management policies. These policies capture one small behavior of traceability link evolution that matches potential actions that a user may take. Such actions include checking in a new architectural artifact, or removing a source code file. Execution of one policy can result in the triggering of one or more other policies, the result is a set of closely collaborating policies that together are responsible for updating traceability links. *ArchTrace* is an example of a tool that uses PBT [191].

Semi-automated methods help to save a considerable amount of effort that the manual methods often require, however, they are still far from being practical. In particular, most of the these methods need special preparation and constant monitoring throughout the project's life-cycle (e.g., installing required infrastructure such as a publish-subscribe structure or a development monitoring tool). To overcome these limitations, a fully automated approach, based on the utilization of Information Retrieval (IR) methods has been proposed. The following is a description of the IR-based automated tracing process.

1.3.3 Automatic Traceability

To overcome the limitations associated with manual and semi-automatic traceability methods, modern requirements tracing tools employ IR methods to automatically generate traceability links. These methods treat the problem as a standard IR problem. The basic

tenet underlying IR-based tracing is that artifacts having a high textual similarity probably share several concepts, so they are likely good candidates to be traced from one another [10]. Examples of IR methods that have been heavily investigated in the automated tracing literature include: Vector Space Model (VSM) [121], Latent Semantic Indexing (LSI) [178], and the Probabilistic Network Model (PN) [47].

The rational behind utilizing IR methods in software engineering tasks stems from the fact that most of software artifacts have textual descriptions. For example, requirements are usually expressed in free text, and semi-formal languages are often used in design documents and code identifiers and comments. IR methods analyze this textual content in order to classify these artifacts' as relevant or irrelevant to each other [104]. IR methods have been used to support several software engineering tasks such as software maintenance [26, 212], mining software repositories [128], program comprehension [41] and code retrieval [165]. Similarly, in requirements engineering, IR methods are used for reusable requirements retrieval [168], requirements discovery [25], and of course, requirements traceability [121].

The IR-based tracing process, which is also known as the automated tracing loop, consists of four main steps. These steps, shown in Figure 1.1, can be described as follows:

- Corpus Building: A corpus is basically a collection of software artifacts. Artifacts in the corpus represent the search space for the information retrieval method.
- Indexing: Indexing is the process of preparing artifacts in the corpus to be compatible with the underlying retrieval model. The output of the process is a compact content descriptor, or a *profile*, which is usually represented as a vector space model [169].
- Retrieval: IR methods are used to identify a set of traceability links by matching a trace query's profile with other artifacts' profiles in the software repository. Based on the specific IR method employed, links with similarity scores above a certain threshold (cut-off) value are considered candidate links [121].

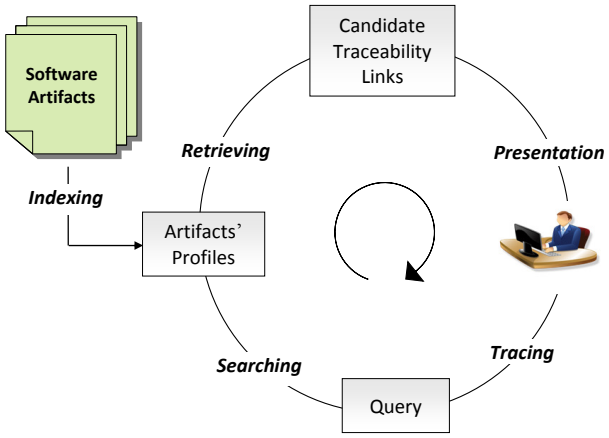


Figure 1.1

Experimental Framework

- Presentation: Candidate traceability links are presented to the human analyst, usually in a list form, for further validation. This process is known as *vetting*. Vetting, in this context, refers to validating and approving the list of candidate links generated by the tool [49, 56].

While the IR-based automated tracing process resembles, to a large extent, a standard Web search problem, it has some key differences that makes it a stand-alone problem by itself. These differences can be summarized as follows:

1.3.3.1 Search Space

The searchable space in tracing consist of individual software artifacts (e.g., requirements, classes, test cases, etc.). Such collection tend to be smaller than natural language collections targeted in a typical Web search or online libraries search [69].

1.3.3.2 Search Query

On the query side, a trace query is composed from the text of a requirement or other software artifacts. They can often be relatively long and may also contain superfluous information [69]. In Web search, a query size is much smaller with specific terms designed specifically to match the user's needs. In addition, a Web search session returns a list of Web documents that best matches the user's query, if the user is not satisfied with the results, the query is altered and the results are regenerated. In traceability, the query is static in the sense that it can not be directly altered by the user.

1.3.3.3 Search Results

Web search engines are often exploratory, which means users do not usually have specific answers for their queries, instead they look for answers that best satisfy their needs. In traceability, a candidate link can be either true or false (e.g., either a particular code module implements a certain requirement or not).

1.4 Research Problem, Objectives, and Plan

Despite major advances in the IR-based automated tracing research, traceability implementation and use is still not pervasive in industry [96]. In fact, several studies have investigated the performance of different IR-based tracing methods and techniques. Converging evidence indicates that most of the exploited methods are equivalent in their performance [1, 121, 172, 178, 199]. In general, IR-based traceability tools still suffer on the accuracy side, and analysts working with such methods still have to spend a considerable amount of time and effort verifying their output, thus, leading traceability to be dropped

as a must-have activity in software development. These observations have motivated a large body of research on automated tracing, with the objective of achieving performance levels, that are adequate for practical applications [43, 47, 55, 96]. The ultimate goal is to accomplish a successful technology transfer from research to industry.

Following this effort, in this dissertation we propose a set of performance enhancement techniques to improve the overall operation of IR-based automated tracing tools. Our main research objectives are to advance the stateoftheart in this field, and to add to the incremental effort of achieving a successful deployment of traceability in practice [96]. Our contributions are based on careful analysis of the current state of research, and the potential areas for improvement. In particular, our research plan can be described as a set of incremental updates over the main phases of the conventional automated tracing process (Figure 1.1), including indexing, retrieval, and presentation. In what follows we briefly describe our research objectives for each of these phases.

1.4.1 Indexing

We start our investigation by looking at the first step of the automated tracing process, known as artifacts indexing. Indexing is a standard, yet a crucial step, in which partial and important information from system artifacts is converted into representations that are compatible with the underlying IR model [89]. Even though indexing has been heavily investigated in related IR fields, such as Web search and documents retrieval, there has not been a thorough investigation of this process and its impact on the retrieval quality in automated tracing literature. To fill this gap, in this dissertation we conduct an intensive

investigation of the different aspects of the indexing process. Our objective is to describe a feature diagram that captures the key components of the indexing process and their relationships in the domain of automated tracing [130]. Our findings are implemented through an adaptive indexer which systematically adjusts indexing settings in such a way that ensures an efficient, yet effective, operation of the automated tracing process.

1.4.2 Retrieval

In this dissertation we investigate several issues related to the utilization of natural language semantics in traceability link retrieval. In particular, several semantically-enabled IR methods, which cover a wide spectrum of semantic relations, are implemented, calibrated, and evaluated. Such methods include:

- Semantic augmented methods such as *Part-of-Speech* aware retrieval [34] and retrieval with thesaurus support [121].
- Latent semantic methods including Latent Semantic Indexing (LSI) [60] and Latent Dirichlet Allocation (LDA) [28].
- Semantic relatedness methods including Explicit Semantic Analysis (ESA) [92] and Normalized Google Distance (NGD) [40].

The main objective is to systematically and collectively evaluate the effectiveness of such methods in supporting IR-based traceability. To that end, our contributions in this domain include: identifying semantic features of software artifacts that have an influence on traceability link retrieval, providing a set guidelines for using semantically-enabled IR methods in requirement traceability tasks, including guidelines for implementing, evaluating, and optimizing such methods, and describing methods for effectively integrating semantics in automated tracing prototypes.

1.4.3 Performance Enhancement

In this part of our research we experiment with several enhancement strategies, beyond the underlying retrieval mechanism, that might impact the overall performance of automated tracing methods. In particular, our contribution in this domain include:

1.4.3.1 Cluster-based Retrieval

We propose a comprehensive analytical study to look at clustering, its operation, and potential benefits in providing retrieval support for traceability tools. We base our research hypothesis on the main cluster hypothesis which suggests that true positives and false positives tend to be grouped into high quality and low quality clusters respectively [175]. The accuracy can then be enhanced by identifying and eliminating low quality clusters.

1.4.3.2 Refactoring Support

IR-based tracing methods track textual signs embedded in the system to establish relationships between software artifacts [96]. However, as software systems evolve, new and inconsistent terminology finds its way into the system's taxonomy, thus corrupting its lexical structure and distorting its traceability tracks [153]. In our research, we argue that the distorted lexical tracks of the system can be systematically re-established through refactoring, a set of behavior-preserving transformations for keeping the system quality under control during evolution [86]. Our main objective is to improve the system lexical structure before indexing. To test this novel hypothesis, the effect of integrating various types of refactoring on the performance of automated tracing methods is investigated. In particular, we identify the problems of missing, misplaced, and duplicate signs in software artifacts,

and then examine to what extent refactorings that restore, move, and remove textual information overcome these problems respectively.

1.4.4 Presentation

Studying human analysts behavior in software engineering is a new research thrust [62, 119]. Building on a growing body of work in this area, in this dissertation we offer a novel approach to understanding requirements analysts information seeking and gathering behavior in a traceability environment. In particular, we try to answer various research questions concerning the way analysts behave when verifying traceability links, and how to improve such behavior in a principled manner. To answer our research questions, we leverage information foraging optimality models to characterize a rational decision process. Our objective is to offer concrete insights into the obstacles faced by requirements analysts. These uncovered discrepancies allow us to define the behavioral problems that are posed by the requirements tracing environment, and suggest multiple directions to advance the fundamental understanding about information seeking in light of the adaptiveness of human behavior.

Figure 1.2 summarizes our research contributions in this dissertation as a set of incremental enhancements over the conventional automated tracing process (Figure 1.1).

1.5 Experimental Settings

IR-based traceability methods retrieve a large number of candidate traceability links in response to each trace query. Some of these links are correct traces, while the majority are

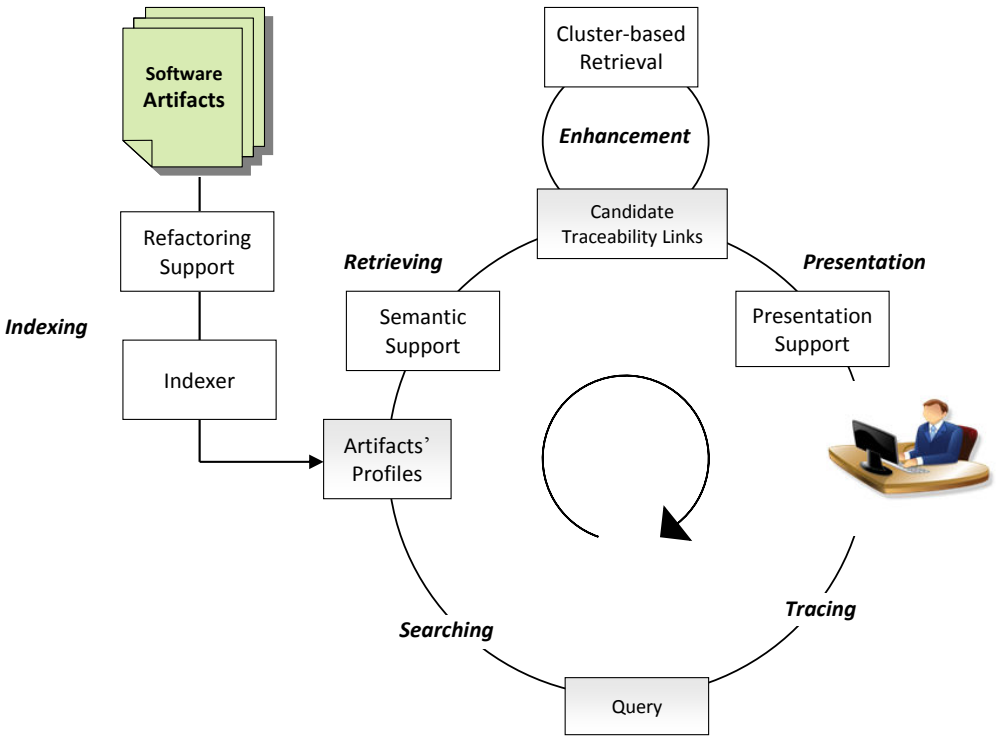


Figure 1.2

Potential Enhancements over the Conventional IR-based Tracing Process

just false positives. Tracing tools are then assessed based on the quality of their output. To illustrate this process, we use the following example.

Example: Assuming (UC1.2) is a use case that describes a functional requirement in one of our experimental software systems. UC1.2 implementation is spread over 30 code classes in the system. UC1.2 was traced against the software system using an IR-based automated traceability tool. The tool retrieved 100 candidate traceability links for UC1.2. Each one of these links represents a code class in the system. These links are usually presented in a list format, arranged in an ascending order based on their similarity to the trace query, where links with higher similarity scores appear at the top of the list. This list is known as the candidate traceability matrix (TM). Furthermore, for browsability purposes, the links in the list are usually presented over multiple pages.

Figure 1.3 shows the candidate TM of UC1.2. The list shows that the tool managed to successfully retrieve 25 of the correct links of UC1.2. Gray boxes represent true positives and white boxes are the false positives.

Multiple primary and secondary performance measures have been presented in the literature to assess the different aspects of IR-based tracing methods performance [238]. These measures can be categorized into two main categories including quality and browsability measures. Following is a description of these categorizes.

1.5.1 Primary Quality Measures

Precision (P), Recall (R), and F-measure are the standard IR metrics often used to assess the quality of the different traceability tools and techniques. Recall measures coverage

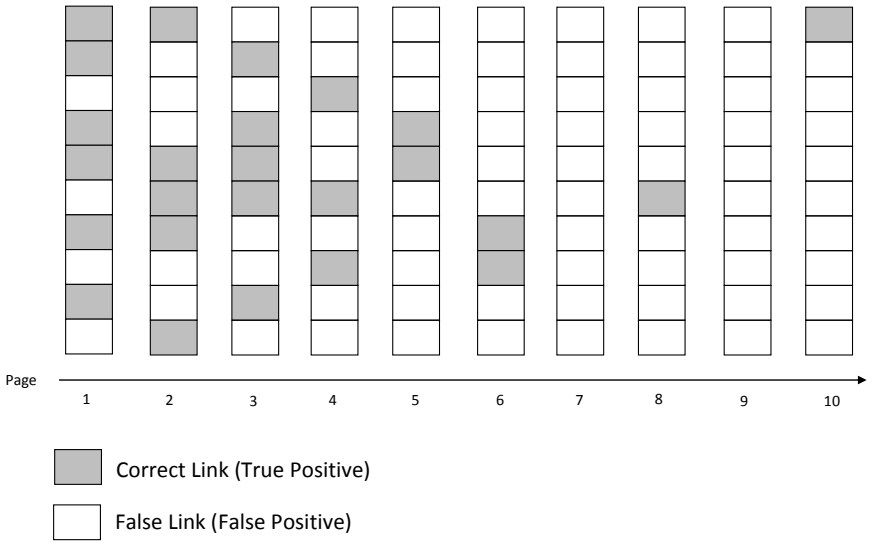


Figure 1.3

The Candidate TM (list of links) of UC1.2 Generated by the Tracing Tool

and is defined as the percentage of correct links that are retrieved, Precision measures accuracy and is defined as the percentage of retrieved candidate links that are correct, and the F-measure is the harmonic mean of recall and precision. Formally, if A is the set of correct links and B is the set of retrieved candidate links, then recall, precision and the F-measure can be defined as:

$$\textit{R (Recall)} = |A \cap B|/|A| \tag{1.1}$$

$$\textit{P (Precision)} = |A \cap B|/|B| \tag{1.2}$$

$$F_\beta = (1 + \beta^2) \cdot (P \cdot R)/(\beta^2 \cdot P + R) \tag{1.3}$$

It is important to point out here that automated tracing methods emphasize recall over precision [121]. The main assumption is that commission errors (distracting false positives) are easier to deal with than omission errors (finding correct links that have not been

retrieved e.g., false negatives). Based on that, F_2 measure, which weights recall twice as much as precision, is usually used. For instance, in the candidate TM in Figure 1.3, the over all recall is 83%, precision is 25%, and F_2 is 47%.

It is common in traceability research to present results over several cut-off points in the list. These points are known as threshold levels. At each threshold level the precision and recall are calculated and the results are then presented in a recall and precision diagram. Several strategies for calculating thresholds have been proposed in the literature [54]. These strategies include:

- Cut-off point: the top *n* links are considered.
- Cut-off percentage: the top *k* percent of the links in the ranked list is considered.
- Constant threshold: links with a similarity score to the query greater than a certain value (λ) are considered.
- Scale threshold: threshold is computed as the percentage of the best similarity score returned by the IR method.

In this dissertation we use a cut-off percentage to calculate our threshold. In particular, precision and recall are reported at ($< .1, .2, ..., 1 >$) threshold values. A higher threshold level indicates a larger list of candidate links, i.e. more links were considered in the analysis [121]. Figure 1.4 shows the recall and precision diagram of the candidate TM in Figure 1.3. Table 1.1 shows the number of links (No. Links), number of true positives (No. TP), recall, precision, and F_2 at each threshold level.

The diagram in Figure 1.4 shows the trade-off between precision and recall. At higher threshold levels, where more links are considered in the analysis, more coverage is expected. However, more false positives are also retrieved which in turn affects the precision negatively.

Table 1.1

Quality Performance Measures at Different Threshold Levels

Threshold	No. Links	No. TP	Recall	Precision	F_2
.1	10	6	.2	.6	0.26
.2	20	11	.37	.55	0.41
.3	30	16	.53	.53	0.53
.4	40	19	.63	.48	0.57
.5	50	21	.7	.42	0.57
.6	60	23	.77	.38	0.58
.7	70	23	.77	.33	0.53
.8	80	24	.8	.6	0.51
.9	90	24	.8	.27	0.48
1	100	25	.83	.25	0.47

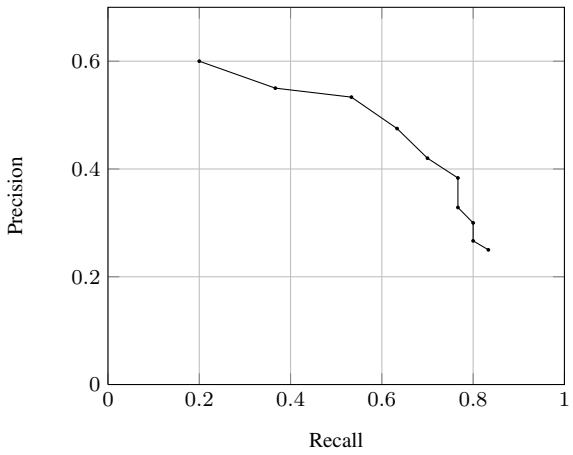


Figure 1.4

Precision and Recall Diagram

1.5.2 Browsability Measures

The second set of measures used in this dissertation are known as the browsability measures. Browsability is the extent to which a presentation eases the effort for the analyst to navigate the candidate traceability links. For a tracing tool or a method that uses a ranked list to present the results, it is important to not only retrieve the correct links but also to present them properly. Being set-based measures, precision and recall do not sufficiently capture information about the list browsability. To reflect such information, other measures are usually used. Assuming *h* and *d* belong to sets of system artifacts $H = \{h_1, \ldots, h_n\}$ and $D = \{d_1, \ldots, d_m\}$. Let C be the set of true links connecting d and h, $L = \{(d, h)|sim(d, h)\}$ is a set of candidate traceability links between d and h generated by the IR-based tracing tool, where $sim(d, h)$ is the similarity score between *d* and *h*. L_T is the subset of true positives (correct links) in *L*, a link in this subset is described as *(d,h)*. L_F is the subset of false positives in L, a link in this subset is described using the notion (d', h'). Based on these definitions, secondary measures can be described as:

1.5.2.1 Average Precision (MAP)

MAP is a measure of quality across recall levels [15]. For each query, a cutoff point is taken after each true link in the ranked list of candidate links. The precision is then calculated. Correct links that were not retrieved (false negatives) are given a precision of 0. The precision values for each query are then averaged over all the relevant links (true positives) in the answer set of that query ($|C|$), producing Average Precision (AP). The Mean Average Precision (MAP) is calculated as the average of AP for all queries in each

dataset [241]. MAP gives an indication of the order in which the returned documents are presented. For instance, if two IR methods retrieved the same number of correct links (same recall), then the method that places more true links toward the top of the list will have a higher MAP. Eq.4 describes MAP, assuming the dataset has Q traceability queries.

$$\textit{MAP} = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{|C_j|}\sum_{k=1}^{|L_{T_j}|} Precision(L_{T_{j_k}}) \tag{1.4}$$

1.5.2.2 DiffAR

Measures the contrast of the list. It can be described as the difference between the average similarity of true positives and false positives in a ranked list. A list with higher DiffAR has a clearer distinction between its correct and incorrect links, hence, is considered superior. Eq. 1.5 describes DiffAR.

$$\textit{DiffAR} = \frac{\sum_{i=1}^{|L_T|} sim(h_i, d_i)}{|L_T|} - \frac{\sum_{j=1}^{|L_F|} sim(h'_j, d'_j)}{|L_F|} \tag{1.5}$$

1.5.2.3 Lag

The average of the number of false positives with higher similarity score that precede each true positive in the ranked list. In other words, the average number of incorrect links that appears before each correct link in the list. Eq. 1.6 shows Lag.

$$\textit{Lag} = \frac{\sum_{i=1}^{|L_T|} Lag(h_i, d_i)}{|L_T|} \tag{1.6}$$

Lag gives an indication of how separated true positives from false positives in a list. A higher lag means that true links are scattered all over the list, which is a sign of poor performance. For the list in Figure 1.3, Lag = 15.28.

1.5.3 Datasets

We use several datasets to conduct our experimental analysis in this dissertation. These datasets have been used in several traceability studies as a benchmark for assessment (e.g., [56, 121, 199]). Following is a description of these datasets and their application domains. Table 1.2 summarizes the characteristics of our experimental datasets. The table shows the size of the system in terms of lines of source code (sLOC), lines of comments (cLOC), type of traceability links, contributor of each dataset, and the application domain.

Table 1.2

Experimental Datasets

Dataset	sLOC	cLOC	Links	Type	Contributor	Domain
iTrust	18.3K	6.3K	314	requirement-to-code	NCSU	Health care
eTour	17.5K	7.5K	394	requirement-to-code	UNISA	Tourism
CM-1	20K	N/A	361	requirement-to-design	NASA	Instrumentation
MODES	20K	N/A	41	requirement-to-requirement	NASA	Instrumentation
WDS	44.6K	10.7K	229	requirement-to-code	Industrial Partner	Job search

1.5.3.1 iTrust

A medical application, developed by software engineering students at North Carolina State University (USA). It provides patients with a means to keep up with their medical history and records and to communicate with their doctors. The system is written in java with approximately 18.3K lines of code. The dataset has 38 requirements and 266 source code files, with 314 requirements-to-source code links.

1.5.3.2 eTour

An electronic tourist guide application developed by final year students at the University of Salerno (Italy). *eTour* was selected as experimental object because its source code contains a combination of English and Italian words, which is considered an extreme case of vocabulary mismatch. The system is written in java with approximately 17.5K lines of code. The dataset has 58 requirements and 166 source code files. The dataset contains 394 requirements-to-source code links.

1.5.3.3 CM-1

Consists of a complete requirements (high-level) document and a complete design (low-level) document for a NASA scientific instrument. The project source code was written in C with approximately 20K lines of code. It has 235 high-level requirements and 220 design elements. The traceability matrix contains 361 actual requirements-to-design traces.

1.5.3.4 MODIS

A dataset that has been constructed from two publically available high-level requirements and low-level requirements documents for NASAs Moderate Resolution Imaging Spectrometer (MODIS). The dataset contains 19 high-level and 49 lowlevel elements with 41 requirement-to-requirement. A typical requirement (high or low-level) is one to two sentences in length.

1.5.3.5 WDS

A proprietary software-intensive platform that provides technological solutions for service delivery and workforce development in a specific region of the United States. In order to honor confidentiality agreements, we use the pseudonym WDS to refer to the system. WDS has been deployed for almost a decade. The system is developed in java and current version has 521 source code files, with 229 requirements-to-source code links, linking a subset of 26 requirements to their implementation.

1.6 Dissertation Outline

The rest of this dissertation is organized as follows:

- Chapter 2: Automatic indexing
- Chapter 3: The role of semantics in automated tracing.
- Chapter 4: Enhancing candidate link generation using clustering.
- Chapter 5: Supporting artifacts indexing through refactoring.
- Chapter 6: On human analyst performance and tool support
- Chapter 7: Concludes the dissertation and suggests venues of future work

CHAPTER 2

INDEXING

Documents indexing is defined as the task of assigning terms to documents for retrieval purposes [89]. This process consists of two generic steps: extracting the subject matter of a document, and expressing that subject matter in index terms to facilitate subject retrieval [173]. Search engines, which search large repositories of textual documents such as digital libraries or the Web, rely heavily on indexing to increase their retrieval efficiency and effectiveness. In software engineering, indexing is used to convert software artifacts into more compact forms known as profiles. A profile is a short-form description of an artifact, is easier to manipulate than the entire artifact, and plays the role of a surrogate at the retrieval stage [166].

In this chapter, we investigate software artifacts indexing for automated tracing. In particular, we experimentally identify the main aspects of the indexing process, and present these aspects in a feature diagram [130]. A feature diagram captures the common and variable components of a certain domain and their dependencies, and organizes such knowledge in a tree-like structure. Our main research question in this chapter is: Which aspects of the indexing process have a statistically significant impact on IR-based traceability link recovery?

2.1 Software Artifacts Indexing

Several methods have been proposed in the Natural Language Processing (NLP) literature for indexing documents expressed in free text [221]. However, the restricted nature of languages used in software development limits the ability of such generalized NLP indexing techniques to perform well when applied to software artifacts. In particular, software artifacts contain a mixture of languages at different levels of formality. Some artifacts (e.g., requirements documentation and manuals) are usually expressed in natural language, while other software artifacts (e.g., design and specifications documents and source code) are expressed in formal languages with some descriptive free text such as code comments. Therefore, indexing software artifacts tends to be less straight forward than indexing free text as these factors have to be taken into consideration.

In our research we focus on source code indexing. Source code represents an extreme case of formal software artifacts, in which code files have a mixture of text at different levels of formality (e.g., comments, code messages, and code identifiers). In particular, source code exhibits certain characteristics that make its indexing a challenging task. Such characteristics include:

- Formality: Source code is highly structured. Developers have to follow strict syntactic rules in order to produce a working code.
- Naming style: There is no guarantee that developers will use genuine words in their code, or follow a well-defined naming convention throughout the project's life cycle. In most cases, developers use combinations of words and abbreviations to name their identifiers [9, 23, 152].
- Reserved words: The majority of the taxonomy in source code are reserved words related to the programming language being used. These words have no direct relation to the application domain.

- Comments: Comments are usually expressed in natural language and have a different lexical structure from source code, thus comments need to be processed separately [248].

2.2 Source Code Indexing

The source code indexing process can be described as a multi-step process. This process starts by extracting textual content (e.g., comments, code identifiers, requirements text) from input artifacts. Lexical processing, stop-words removal, and stemming are then applied to reduce words to their roots. In what follows we describe the main steps of this process in greater detail.

2.2.1 Information Extraction

Domain knowledge and code concepts are embedded in the linguistic aspects of source code, including identifiers names, error messages, and comments [9,152,179,188]. Source code identifiers, such as names of classes, attributes, methods, and parameters, often capture the developers cognitive perception (understanding) of the application domain. The underlying assumption is that developers name their identifiers in such a way that is related to the functionality of their code, and not completely at random [166]. For example, an identifier named `user_id` is expected to hold a user's identification information.

The other source of knowledge in source code is the comments. Comments serve as the internal documentation of the system. In the literature, the utilization of comments in source code indexing has generated some debate. The argument that supports using comments is based on the fact that programmers, under the pressure of approaching deadlines, tend to focus on the functionality of the code with only little attention paid to its style, thus

there is no guarantee that the naming style used by the developers will be good enough to capture the domain concepts. However, comments are commonly written in a language similar to that of the external documentation [229,239,248], and developers often use comments to explain and communicate their code. Therefore, comments are expected to carry valuable information that should not go to waste [166, 178].

Argument against using comments is also supported by several observations. For instance, not all software systems contain comments, quality of comments and their levels of abstraction vary widely among software systems, and comments might be outdated or even redundant to the source code [10, 133]. For instance, in the following line of code, comments add no value to the code concept:

```
Increment++; //incrementing by 1
```

2.2.2 Lexical Analysis

Lexical analysis is used to extract meaningful words from extracted tokens. A *token* is defined to be any alphabetical sequence of characters separated by non-alphabetical characters or by letter capitalization [247]. It is a common practice to define identifiers by concatenating two or more words [10]. Such identifiers can be broken down into units based on commonly used coding standards, such as the location of the capital letter in the identifier name ($firstName \rightarrow first\ name$) or any other separators such as the underscore (_) character. Abbreviations are also commonly used by programmers to name identifiers [151]. Domain specific dictionaries or lookup tables can be used to expand abbreviations to their constituent words. For example:

$$\boxed{hsptlRcrd \rightarrow hsptl\ rcrd \rightarrow hospital\ record}$$

2.2.3 Filtering

The main objective of indexing in IR is to generate a set of index terms that achieve the best performance with IR methods. Stopwords are any words that are irrelevant to the code concept. Such words carry a very low information value and can affect the retrieval process negatively [175]. We identify four categories of stopwords that are usually filtered out of the code profiles. These categories include:

- Generic stopwords: Stopwords that are used in natural language, such as (and, but, the). A list of the most common stopwords in English is available at [87].
- Programming language reserved words: The set of keywords reserved by the programming language, such as (integer, string, class, static).
- Non-textual tokens: Set of language operators and special characters which are used to perform certain arithmetic operations such as (+,-, %, @).
- Other stopwords: Often found in comments that are used throughout the project as references (e.g., author information or license terms). Such comments usually appear in all system files, and often add no distinctive value to the retrieval process and can be removed.

2.2.4 Stemming

Stemming is the process of reducing a word to its root. This process can be accomplished using techniques such as rule-based and dictionary-based stemming. Rule-based stemming uses a large number of language-specific rules to reduce words to their canonical morphological representations. Porter's algorithm [211] is one of the most used rule-based stemmers in IR research. Rule-based stemming is simple to implement and maintain, and has a modest computational cost. However, its quality depends highly on the set of rules applied. In addition, its performance may downgrade when dealing with irregular cases such

as *eat* and *ate*. To overcome this problem, a dictionary-based approach is sometimes used. This approach mainly involves maintaining known morphological word roots that exist as real words in a lookup table. Krovetzs stemmer [141] is an example of a dictionary-based English stemmer where potential root forms are contained in the dictionary.

Stemming has been found to improve the effectiveness and the efficiency of the retrieval system [175]. However, this improvement in the performance does not come without a risk. In particular, it has been observed that as words get stemmed, they lose an important part of their meaning [175]. This risk becomes more obvious in free text retrieval, where different parts of speech carry different information values to the retrieval engine. For instance, it has been reported that nouns and verbs are better discriminators, or more descriptive, to the content of a document than other parts of speech [39, 78].

2.3 A Feature Diagram for Source Code Indexing

A feature diagram is a hierarchy of common and variable features characterizing the set of instances within a domain. It helps in determining the scope of the domain and provides an external view that stakeholders can understand and communicate easily [130]. The analysis in this chapter is concerned with identifying the variabilities and commonalities in the source code indexing domain. In particular, our main objective is to develop a feature diagram for source code indexing to support knowledge management and reuse in the domain of IR-based automated tracing.

Figure 2.1 depicts a feature diagram we use as a basis for our discussion. It is important to note that we do not aim for this domain characterization to be immune from change.

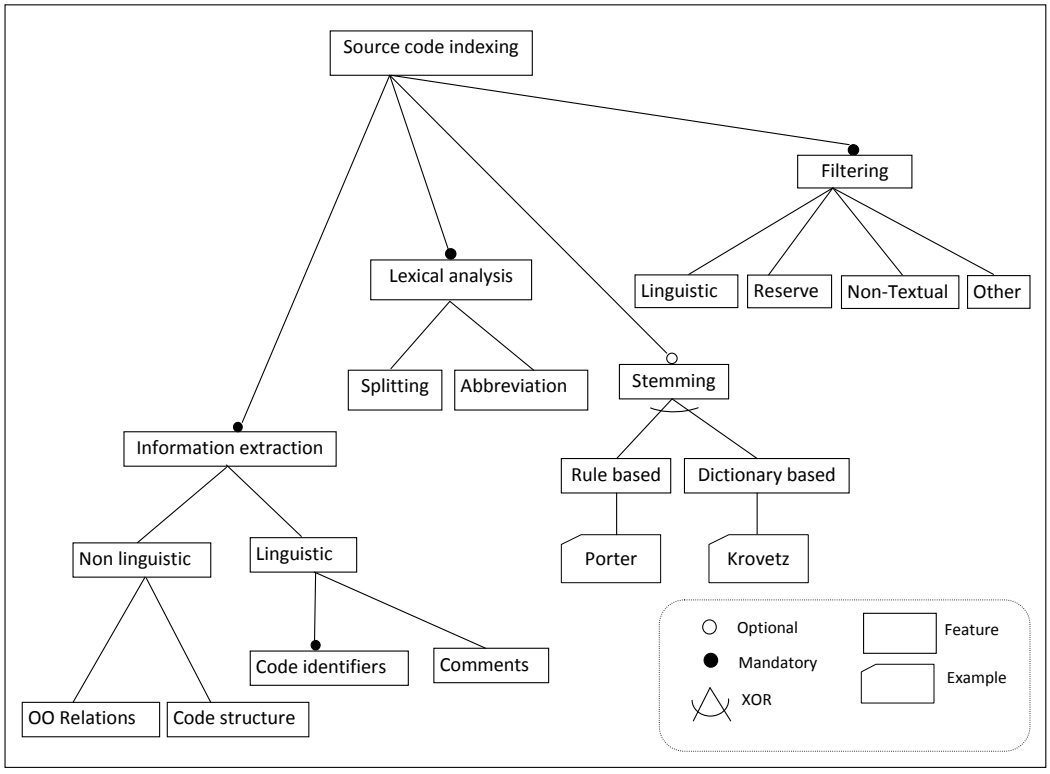


Figure 2.1

A Feature Diagram for the Source Code Indexing Process

In fact, we expect this knowledge representation to evolve as our understanding of the indexing process matures.

The experiment we describe in the next section is an attempt to further our understanding empirically. Our main goal here is to show the vast range of available choices as represented by the current code indexing approaches from a reuse perspective. Figure 2.1 follows the notations defined in [130]. The features (denoted by the boxes) of the concept source code indexing are described, which is located at the top of the feature diagram. The boxes directly connected to source code indexing are the direct sub-features or sub-steps. The little circles at the edges connecting the features define the semantics of the edge. A filled circle means mandatory, thus every code indexing shall perform information extraction, lexical analysis, and filtering. However, since the utilization of comments and stemming in the indexing process have generated some debate in the literature, they are identified as optional features currently (denoted by the outlined circle at the edge). Alternative features indicate an exclusive-or (XOR) choice, so when stemming is performed in practice, a rule-based or a dictionary-based stemmer could be used.

2.4 Experimental Analysis

Experimentally validating all the features in a feature diagram and identifying all their possible dependencies can be tedious [106]. In our experiment, we chose *comments* and *stemming* as our independent variables as they are marked as optional features in the feature diagram in Figure 2.1. Our research questions are: Should comments be considered when tracing source code? And is stemming required?. To answer these questions, we identify

four experimental settings with all the possible comments and stemming combinations. We control the rest of the features shown in Figure 2.1 to void their effect. These settings, summarized in Table 2.1, can described as follows:

- Base case analysis (C): The base case in our experiment includes indexing source code only. Code identifiers are extracted and lexically processed, stopwords are filtered out, no comments are considered and no stemming is performed. This case represents a reference point for comparing other cases performance.
- Stemming the source code (CS): To investigate the effect of stemming on source code traceability, all source code profiles generated in the base case are stemmed using Porter's algorithm [211].
- Considering comments (CC): In this case, source code in is indexed with comments. The comments, in addition to code identifiers, are extracted and lexically processed. All irrelevant stopwords are removed and no stemming is performed.
- Stemming comments (CCS): In this case, all CC profiles from the previous case are stemmed using Porter's algorithm.

Table 2.1

Experiment Settings

Case	Code	Comments	Stemming	Lexical Analysis	Filtering
C	√	×	×	√	√
CS	√	√	×	√	√
CC	√	×	√	√	√
CCS	√	√	√	√	√

The independent variables in our experiment are comments and stemming. We use Porter's stemmer [211] for its computational efficiency. We also use the stop words list available in [87] to filter out unnecessary words. Our dependent variable is the quality

of the generated requirements-to-code traceability matrix. We use precision and recall to assess the quality after applying the different treatments listed in Table 2.1.

We use three requirements-to-code datasets from our dataset collection to carry out our experiment. These datasets include *iTrust*, *eTour*, and *WDS*. We use standard Vector Space Model (VSM) with TFIDF weights as our traceability method. Using VSM, each document is represented as a set of terms $T = \{t_1, ..., t_n\}$. Each term t_i in the set *T* is assigned a weight w_i. The terms in *T* are regarded as the coordinate axes in N-dimensional coordinate system, and the term weights $W = \{w_1, ..., w_n\}$ are the corresponding values. Thus, if q and d are two artifacts represented in the vector space, then their similarity is measured as the cosine of the angle between them (Eq. 2.1):

$$Sim(q, d) = \frac{\sum q_i \cdot d_i}{\sqrt{\sum q_i^2 \cdot \sum d_i^2}} \tag{2.1}$$

where q_i and d_i are real numbers standing for the weights of term i in q and d respectively. Word counts, or term frequencies in documents are often used to assign weights to terms in the document's vector. While this method is computationally efficient, it might represent a bias towards long text documents or frequent words in the corpus. To mitigate this risk, another weighting scheme based on term frequency and inverse document frequency (TFIDF) is used. Using this approach, q_i and d_i in Eq. 1 become $q_i = tf_i(q) \cdot idf_i$ and $d_i = tf_i(d) \cdot idf_i$, where $tf_i(q)$ and $tf_i(d)$ are the frequencies of term *i* in q and d respectively. idf_i is the inverse document frequency, and is computed as $idf_i = log_2(t/df_i)$, where t is the total number of artifacts in the corpus, and df_i is the number of artifacts in which term *i* occurs. TFIDF determines how relevant a given word is in a particular

document. Words that are common in a single or a small group of documents tend to have higher TFIDF, while terms that are common in all documents such as articles and prepositions get lower TFIDF values. A higher TFIDF implies a stronger relationship between the term and the document it appears in, thus if that term were to appear in a query, the document would probably be a correct match.

A prototype is created to carry out the experimental analysis. This prototype has two main functions: a code indexer and a requirements-to-code tracer. The code indexer uses regular expressions to match and capture identifiers and comments in a source code file, an implementation of Porter's algorithm is used to perform stemming, and generic and programming language specific stopwords lists are provided to filter out irrelevant terms. The prototype also has a control panel to allow the user to control the settings of the indexing process, such as whether to include comments or to do stemming. After performing indexing, all the generated profiles are stored in the artifacts database to be used later in the tracing process. For each dataset, we trace all the requirements (use cases) to code classes under different settings. The answer sets of our datasets are used to evaluate the quality of the automatically generated traceability links under the different experimental settings.

2.5 Results and Discussion

Analysis results for both of our datasets are shown in Table 2.2 and Figure 2.2. Standard recall and precision diagrams are used to report the results. Wilcoxon Signed Ranks test is used to assess the effect of the different experimental treatments. This is a non-parametric statistical hypothesis test used to compare two related samples, or repeated measurements

on a single sample, where the null hypothesis is that the median difference between pairs of observations is zero. Wilcoxon test makes no assumptions about the distribution of the data [65, 237], which makes it appropriate for testing our hypothesis. We use $\alpha = 0.05$ to test the significance of the results.

Table 2.2

Wilcoxon Signed Ranks Test Results (***p**-values* at $\alpha = .05$)

	iTrust		eTour		WDS	
	Recall (Z, *p-value*)	Precision (Z, *p-value*)	Recall (Z, *p-value*)	Precision (Z, *p-value*)	Recall (Z, *p-value*)	Precision (Z, *p-value*)
C x CS	(.000, 1.000)	(-1.461, .144)	(-.510, .610)	(-.730, .465)	(-.510, .610)	(-1.000, .317)
CS x CC	(-2.497, .013)	(-2.803, <.01)	(-2.308, .021)	(-2.803, <.01)	(-2.308, .021)	(-2.803, <.01)
CC x CCS	(-2.803, <.01)	(-2.023, .063)	(-1.656, .098)	(-1.095, .273)	(-2.549, .011)	(-1.732, .083)

To assess the effect of indexing comments on the performance, all requirements in all datasets are traced to the CC profiles. The performance, in terms of recall and precision, is compared to the base case performance (C), where no comments are considered. Analysis results show that in all three datasets, the recall and precision improve significantly after considering comments (Table 2.2). This confirms our speculations that comments carry a considerable amount of the domain knowledge, thus they should be considered as a valuable source of information when indexing source code.

To test the effect of stemming on the performance, we initially compare the performance of the base-case (C), where no stemming is performed over source code, to the second case where source code is stemmed (CS). The Wilcoxon test results show no sig-

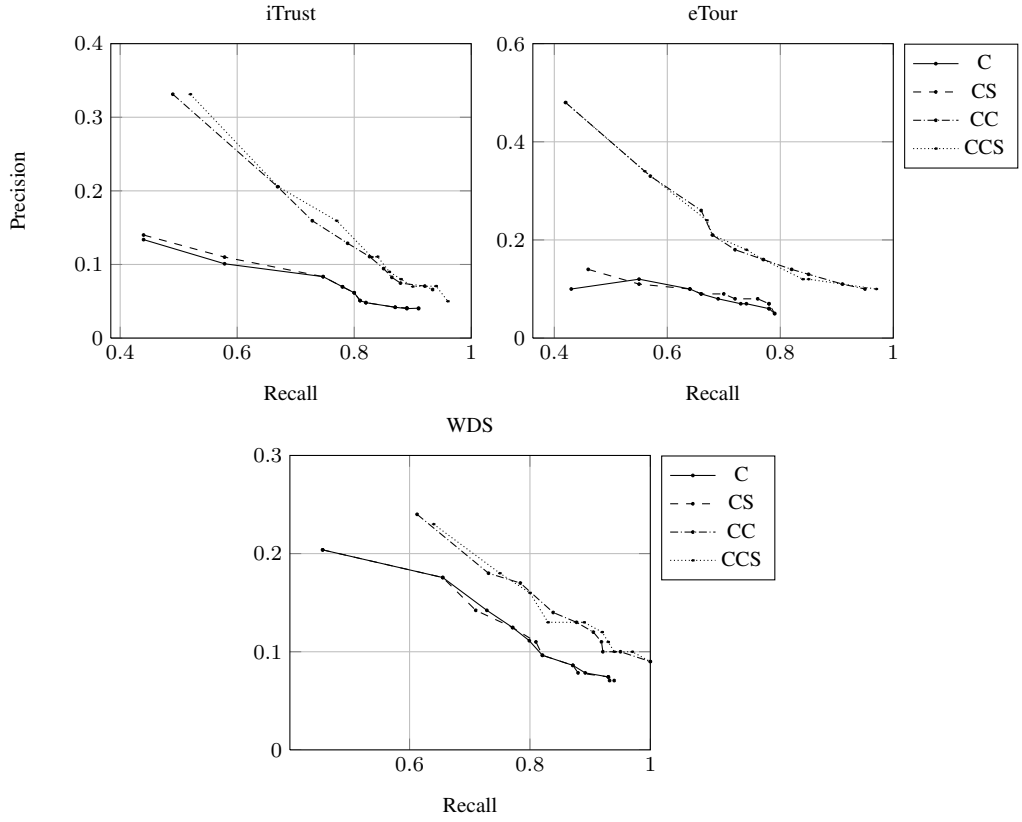


Figure 2.2

Precision and Recall Data for our Experimental Datasets

nificant difference in the performance in all three datasets in terms of recall or precision (CS = C). This leads to the conclusion that if only source code is considered in the indexing process, then no stemming is required.

In the second case, comments are stemmed (CCS). The results are compared to the unstemmed comments case (CC). Analysis shows that applying stemming on comments improves the recall significantly in *iTrust* and *WDS*. However, no significant improvement is detected in *eTour*. In terms of precision, the average precision shows no significant improvement in any of our datasets. In fact, in all three datasets, the precision has actually declined, however this decline was statistically insignificant. This negative effect on the precision can be explained based on the fact that stemming causes loss of information when reducing words to their roots [140], which results in retrieving more irrelevant links. For example, the tenses of verbs may be lost in creating a stem. Therefore, when tracing requirements, where verbs carry the highest information value [194], this could lead to a decline in the accuracy.

To further confirm these findings, we observed the percentage of terms affected by stemming in both datasets. Results are shown in Table 2.3 which shows that the percentage of terms that were affected by stemming in source code where considerably less than comments. This can be explained based on the fact that developers often do not use a fancy language in naming their identifiers, instead they usually stick to the base form of the word, which limits the effect of stemming when applied to source code [9, 23, 152]. However, comments are usually written in natural language and in complete sentences, thus stemming had more obvious impact on comments, and consequently retrieval quality.

In addition, the insignificant effect of comments stemming on recall in *eTour* can be explained based on the fact that this particular dataset includes a mix of Italian and English keywords, which limits the operation of our English-Language stemmer (only 4.2% of the comments keywords in this dataset are affected by stemming).

Table 2.3

Percentage of Terms Affected by Stemming in all Datasets

	iTrust	***eTour***	***WDS***
Comments terms	15%	4.2%	23%
Source code terms	3.3%	1.2%	4.7%

Overall, the experimental analysis results shows that when stemming is applied to comments, it improves the recall significantly. However, if only the code is to be used (for example in cases where the code is not commented), then stemming is unnecessary. Therefore, in the feature diagram in Figure 2.1, comments will be included as a feature in the code indexing process. In addition, we would still keep stemming as an optional feature. However, we would add a *requires* dependency link from *comments* to *stemming*. This indicates that if comments are considered, then stemming is required.

2.6 Threats to Validity

Several factors can affect the validity of our experiment. Construct validity is the degree to which the variables accurately measure the concepts they purport to measure [59]. In our experiment, we feel that the independent variables (*comments* and *stemming*) and

dependent variables (precision and recall) accurately measure the concepts they purport to measure including variabilities in the code indexing process for the independent variables, and quality of automatically generated traceability links for the dependent variables.

Threats to external validity impact the generalizability of results. In particular, the results of this study might not generalize beyond the underlying experimental settings [59]. However, several strategies are used in our experiment to help mitigate these threats. For instance, we choose a representative stemmer and stopword lists in our analysis. In addition, we experiment using midsize datasets, from different domains, and find converging results. This helps generalize our findings to other application domains. However, both *eTour* and *iTrust* were developed by university students and may not be representative of a program written by industrial professionals. It is therefore unknown if the results will generalize to other software systems, other application domains, or larger systems.

2.7 Conclusions

In this chapter, we have tackled the problem of indexing source code for supporting requirements-to-source-code traceability. We introduced a feature diagram to describe the indexing process, and conducted an experiment using three datasets including *WDS*, *eTour*, and *iTrust*, to examine some of the diagram's features and their dependencies. Results showed that considering comments in the indexing process improved the quality of the generated traceability links significantly. Stemming was also found useful when comments were considered. However, if comments were ignored then the overhead of stemming is unnecessary.

Our findings in this chapter emphasize the importance of adopting a good naming convention in software systems. Meaningless names or abbreviations often lead to a vocabulary mismatch between requirements and source code, which often leads to a considerable drop in the retrieval accuracy. Our findings also emphasize the importance of considering comments in indexing. Commented code is not only more understandable, but also easier to be traced.

CHAPTER 3

SEMANTICS IN AUTOMATED TRACING

This chapter investigates the potential benefits of utilizing natural language semantics in automated traceability link retrieval. In particular, we evaluate the performance of a wide spectrum of semantically-enabled IR methods in capturing and presenting traceability links in software systems. Our objectives are to gain more operational insights into these methods, and to provide practical guidelines for the design and development of effective requirements tracing and management tools. To achieve our research objectives, we conduct an experimental analysis using three datasets from various application domains. Results show that considering more semantic relations in traceability link retrieval does not necessarily lead to higher quality results. Instead, a more focused semantic support, that targets specific semantic relations, is expected to have a greater impact on the overall performance of tracing tools. In addition, our analysis shows that explicit semantic methods, that exploit local or domain-specific sources of knowledge, often achieve a more satisfactory performance than latent methods, or methods that derive semantics from external or general-purpose knowledge sources.

3.1 Introduction

The tenet underlying IR-based tracing methods is that artifacts having a high textual similarity probably share several concepts, so they are likely good candidates to be traced from one another [12]. The main assumption is that practitioners use consistent terminology throughout the project's lifecycle. These terms serve as signs that can be traced to produce meaningful tracks in the system [96]. In other words, IR methods assume that the same words are used whenever a particular concept is described [18]. However, extensive research in static code analysis shows that as projects evolve, new and inconsistent terminology finds its way into the system's taxonomy. This textual gap in the system grows gradually to a case where different system's artifacts have information contents with a large degree of variance from each other [9, 23]. This problem is known as the vocabulary mismatch problem, and is regarded as one of the principal causes of poor performance in retrieval engines [90].

In an attempt to alleviate the problem of vocabulary mismatch, researchers have started investigating semantically-enabled IR techniques that look beyond the lexical structure of software artifacts. Unlike lexical methods, which deal with text as strings of tokens, semantic methods capture similarity among various artifacts by exploiting the semantic knowledge embedded in their contents. In requirements engineering, several semantically enabled methods have been exploited to support various related tasks such as, requirements discovery, analysis, modeling, traceability, and reuse [25, 117, 135, 168, 194]. The underlying assumption is that the overwhelming majority of requirements are written in natural language (NL) [164, 198]. Therefore, IR methods that exploit semantics in the NL compo-

nent of software artifacts should be able to discover dimensions that lexical methods often overlook [116].

It is important at this point of our analysis to distinguish between two kinds of semantics: programming language semantics, which refer to the meaning of a program as a state transformer from inputs to outputs, and natural language semantics, which refer to the meaning inherent in the natural language component of artifacts, such as code identifiers' names and comments [23]. In this chapter, the latter is our concern. In particular, the analysis in this chapter addresses several research questions related to the utilization of semantics in traceability link retrieval. Such questions include, how much semantics is needed? What specific effects does semantics have on the performance? What are the merits of different semantic enhancements? And what is the scope of applicability of different methods? Our work not only advances the fundamental understanding about the role of semantics in supporting automated tracing, but also enables principled ways to increase the practicality of requirements tracing and management tools. In particular, the contributions of this chapter are:

- A comprehensive statistical analysis of the merits of a wide spectrum of semantically-enabled IR methods in identifying and capturing traceability links in software systems.
- A systematic categorization of different semantically-enabled IR methods based on their internal operation, the semantic relations they target, and scope of application.
- A set of guidelines for using semantically-enabled IR methods in requirements traceability tasks, including guidelines for optimizing, evaluating, and implementing such methods.

To achieve our research objectives, we analyze the performance of a plethora of semantically-enabled IR methods using three datasets from various application domains. These meth-

ods include basic Vector Space Model (VSM) [222], VSM with thesaurus support (VSM-T) [121], *Part-of-Speech* enabled VSM (VSM-POS) [34], Latent Semantic Indexing (LSI) [60], Latent Dirichlet Allocation (LDA) [28], Explicit Semantic Analysis (ESA) [92], and Normalized Google Distance (NGD) [40]. Following is a description of these methods.

3.2 Semantically-Enabled IR

In our analysis, we experiment with various semantically-enabled IR methods that are based on the algebraic vector space model (VSM). Such methods transform textual documents into more compact representations in the form of vectors. These vectors can hold various types of information, representing different aspects of the semantic knowledge embedded in a text corpus (e.g., words counts in artifacts or latent topical structures in a corpus [28]). Once the vector representation of a document is generated, a simple similarity measure (e.g., the cosine-distance [258]) can be used to calculate the similarity between these vectors, thus determining the relevance of documents.

Based on the underlying IR model used, we identify three categories of VSM-based semantically-enabled IR methods. These categories include semantic-augmented, latent semantic, and semantic relatedness methods. In what follows, we describe each of these categories, its main methods, and their applications in greater detail.

3.2.1 Vector Space Model

We use Vector space model (VSM) with TFIDF weights as an experimental baseline in our analysis. A full description of VSM is available in (Section 2.4). Due to its conceptual and mathematical simplicity, basic VSM has gained a considerable popularity in informa-

tion retrieval research [25]. However, this over simplification often comes with several limitations. For instance, the bag-of-words assumption assumes term-independence. This assumption discards the punctuation information and the words ordering. However, since there are strong inherent associations between terms in a language, this assumption is never satisfied [258]. This often results in loss of information and ambiguity problems as texts should ideally be compared at their topic level, and not based on the specific words that were chosen to express these topics [143].

3.2.2 Semantic-Augmented Methods

This category includes IR methods that semantically *augment* the basic VSM by adding new information to the basic document's vector, thus integrating additional evidence into retrieval. We identify two methods under this category, including Vector Space Model with thesaurus support (VSM-T) and Vector Space Model with *Part-of-Speech* tagging (VSM-POS). Both methods have been investigated before in automated tracing research [34, 121]. Following is a description of these methods in greater detail.

3.2.2.1 Vector Space Model with Thesaurus Support

A very common occurring semantic relation in software artifacts is *synonymy*, or equivalent words. As mentioned earlier, as software projects evolve, developers tend to use different vocabulary, including abbreviations and acronyms, to refer to certain domain concepts [9, 61]. Basic VSM fails to capture these relations as it assumes terms' independence. A simple way to overcome this problem is to equip VSM with a dictionary or a thesaurus that keeps track of such relations (e.g., different acronyms used to describe a

certain concept). Documents are then matched based on their matching keywords, as well as *synonymy* relations found in the supporting thesaurus.

The integration of a thesaurus into VSM is relatively simple. For each pair of synonyms identified (s_i, s_j), a perceived similarity coefficient α_{ij} can be assigned to indicate their equivalence [121]. For each document in the corpus, document vectors are expanded based on these synonym pairs. A similarity coefficient of $\alpha_{ij} < 1$ is usually assigned to distinguish a *synonymy* match from an exact match ($\alpha_{ij} = 1$). The similarity between two documents can then be calculated as:

$$s(q, d) = \frac{\sum q_i d_i + \sum_{(k_i, k_j, \alpha_{ij}) \in T} \alpha_{ij}(q_i d_j + d_j q_i)}{\sqrt{\sum q_i^2 . \sum d_i^2}} \tag{3.1}$$

Based on the type of the integrated thesaurus, two methods of VSM-T can be distinguished under this category including VSM with general purpose thesaurus (VSM-T-WN) and VSM with domain specific thesaurus (VSM-T-DT). VSM-T-WN method uses general purpose dictionaries, such as WORDNET, to derive synonyms. WORDNET, introduced and maintained by the Cognitive Science Laboratory of Princeton University, is a large lexical database of English verbs, nouns, and adjectives, grouped into sets of cognitive synonyms, known as *synsets* [82]. The main advantage of general purpose dictionaries is their high coverage of terms, and the fact that they are constantly being maintained by highly trained linguists. However, with no domain-specific knowledge (context), relying on a general purpose thesaurus to handle abbreviations and acronyms in a software system can become a challenge.

VSM-T-DT methods use a domain-specific thesaurus to handle synonym pairs derived from the project domain's taxonomy. These domain-specific thesauri can deal with cases such as acronyms and abbreviations. However, they can become quickly out-of-date, as keeping track of the changes in the project's vocabulary over time can become an exhaustive task.

3.2.2.2 Vector Space Model with Part-of-Speech Tagging

Part of speech (POS) refers to the syntactic role of terms in written text (e.g., nouns, verbs, adjectives). Research in natural language processing (NLP) has revealed that some parts of speech carry more information value than others. For instance, it has been reported that nouns and verbs are better discriminators, or more descriptive, to the content of a document than other parts of speech [39,78]. These observations have been recently integrated into the IR paradigm. The main assumption is that, favoring certain parts of speech over treating all terms at the same level of importance should improve the overall accuracy of retrieval engines [160, 204, 217].

We build upon these observations to derive a VSM model with *Part-of-Speech* tagging support (VSM-POS) for traceability link retrieval. To calculate the similarity between two artifacts in the system (q and d), initially POS analysis is conducted to identify different parts of speech (e.g. verbs and nouns) in q and d. This process can be achieved using various text tagging tools such as TreeTagger [224]. Only terms which belong to a particular part of speech are then considered in retrieval.

In this chapter we only consider the two linguistic forms of verbs (VSM-POS-V) and nouns (VSM-POS-N) in building term vectors of software artifacts. These two particular parts of speech have been found to carry higher information values than other parts, capturing main actions and objects in software artifacts [79, 127, 168, 228]. Therefore, indexing based on these two linguistic forms is expected to filter out noise resulting from keywords that do not contribute to the artifact's topic.

Limitations of methods based on POS often stem from the complications associated with text tagging, or automatically identifying different parts of speech in a text. While generating linguistic parse trees can be relatively simple for artifacts expressed in natural language (e.g., requirements documents), this process can become more complicated for semi-formal or formal artifacts, such as source code or design documents [2, 24]. In addition, selecting an optimal linguistic form or a combination of forms that best achieves desired performance levels can be computationally exhaustive [160].

3.2.3 Latent-Semantic Methods

While semantic-augmented methods help to exploit basic semantic aspects of artifacts, they reveal a little about the inter or intra semantics in a corpus. To count for such information, another set of methods are often used. Such methods use statistical and probabilistic models to automatically discover latent semantic structures in text corpora. In particular, instead of representing a document as a vector of independent terms, latent methods represent documents and terms as combinations of implicit semantic schemes that are often hidden from other methods. Two methods can be identified under this category: Latent Se-

mantic Indexing (LSI) and Latent Dirichlet Allocation (LDA). In what follows, we briefly introduce both methods and discuss their limitations and applications in greater detail.

3.2.3.1 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is a statistical method for inducing and representing aspects of the meanings of words and passages reflective in their usage. LSI is based on the assumption that there is some underlying (*latent*) structure in words that is partially concealed by the variability of words used to express a certain concept [60]. In particular, using statistically derived conceptual indices, LSI tries to overcome the problem of vocabulary mismatch by capturing the semantic relations of *synonymy* (equivalent words) and *polysemy* (multiple meanings) in software artifacts [218].

LSI is a corpus-based technique that uses Singular Value Decomposition (SVD) to estimate the structure in word usage across documents in the corpus [64]. It starts by constructing a term-document matrix (*A*) for terms and documents in the corpus. This matrix is usually huge and sparse. Word counts are often used to build this matrix. SVD is then applied to decompose *A* into three new matrices $A = USV^T$ where *T* stands for transpose. Dimensionality reduction is then performed to produce reduced approximations of $< U, S, V^T >$ by keeping the top *K* eigenvalues of these matrices. A dimensionality reduction technique takes a set of objects that exist in a high-dimensional space and represents them using low dimensions, often in a 2D or 3D space. These reduced matrices can be described as $< U_k, S_k, V_k^T >$. The best value of *K* that fits a certain corpus can be obtained experimentally; however, a value in the range of [100, 300] is frequently used [60]. From

the newly reduced space, the equation $V = A^T U S^{-1}$ can be derived. Assuming *A* is a matrix with $n > 1$ documents, for a given document vector *d* in *A*, *d* can be expressed as $d = d^T U S^{-1}$. In LSI, the query is also treated as a document, which is the case in traceability, where the query itself is a requirement or a piece of code. The query *q* can be expressed in the new coordinates of the reduced space as $q = q^T U S^{-1}$. In the *K*-reduced space, *q* and *d* can be represented as $d = d^T U_k S_{-1}^k$ and $q = q^T U_k S_{-1}^k$ respectively. The similarity of q and d can then be calculated as the cosine measure:

$$sim(q, d) = sim(q^T U_k S_k^{-1}, d^T U_k S_k^{-1}) \tag{3.2}$$

In other words, retrieval in LSI is performed using the database of singular values and vectors obtained from the SVD analysis. Therefore, a query and a document can have a high cosine similarity even if they do not have any overlapping terms, as long as their terms are semantically similar in the latent semantic space.

LSI has been employed in a wide range of software engineering activities such as categorizing source code files [174], detecting high-level conceptual code clones [177], and recovering traceability links between documentation and source code [178]. The drawbacks of LSI include its huge storage requirements, the computational costs of performing SVD, and the assumption of normally distributed data, which might be inappropriate for handling the word count data of the term-by-document matrix. In addition, it is often difficult to add new documents to the corpus, and determining the optimal *K* can be computationally exhaustive [70].

3.2.3.2 Latent Dirichlet Allocation

LDA was first introduced by David Blei et al. [28] as a statistical model for automatically discovering topics in large corpus of text documents. The main assumption is that documents in a collection are generated using a mixture of latent topics, where a topic is a dominant theme in the corpus.

A topic model can be described as a hierarchical Bayesian model that associates with each document d in the collection D a probability distribution over a number of topics K. In particular, each document d in the collection $(d_i \in D)$ is modeled as a finite mixture over K drawn from a Dirichlet distribution with parameter α, such that each d is associated with each $(t_i \in K)$ by a probability distribution of θ_i. On the other hand, each topic t in the identified latent topics $(t_i \in K)$ is modeled as a multidimensional probability distribution, drawn from a Dirichlet distribution β, over the set of unique words in the corpus (W), where the likelihood of a word from the corpus $(w_i \in W)$ to be assigned to a certain topic t is given by the parameter ϕ_i.

LDA takes the documents collection D, the number of topics K, and α and β as inputs. Each document in the corpus is represented as a bag of words $d =< w_1, w_2, \ldots, w_n >$. Since these words are observed data, Bayesian probability can be used to invert the generative model and automatically learn ϕ values for each topic t_i, and θ values for each document d_i. In particular, using algorithms such as Gibbs sampling [210], a LDA model can be extracted. This model contains for each t the matrix $\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, representing the distribution of t over the set of words $< w_1, w_2, \ldots, w_n >$, and for each document d the matrix $\theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, representing the distribution of d over the set

of topics $< t_1, t_2, \ldots, t_n >$. Once these matrices (vectors) are produced, a similarity measure such as the cosine distance can be used to compute the similarity of two documents by comparing their topic distribution vectors to produce a ranked list of topically-similar documents [113].

Selecting the number of topics (K) that best fits a certain text corpus is computationally expensive. In NLP tasks, often a heuristic of 50 to 300 topics is empirically specified depending on the size of the collection [28, 103, 253]. In some other cases, such values are determined automatically. For example, Teh et al. [240] proposed a non-parametric model known as Hierarchical Dirichlet Processes which extends LDA and seeks to learn the optimal K automatically. However, while such heuristics and methods achieve satisfactory performance in NLP tasks, they are not necessarily optimal for software systems [180,202].

Several methods have been proposed in the literature to approximate near-optimal combinations of LDA parameters (α, β, K) in software systems. Such methods can be *a)* manual, based on a domain expert understanding of the system [7, 180] , *b)* experimentally-determined, in which LDA parameters are tuned until a configuration that achieves acceptable performance over a certain quality measure is reached [16, 22], or *c)* automatically generated using statistical methods or machine learning approaches [101, 202].

LDA has been used to support several software engineering tasks, such as mining semantic topics from source code [159], analyzing software evolution [243], and automated tracing [16, 199]. Several drawbacks of LDA stem from its mathematical complexity. For example, it can be easily misguided by uninformative words, also the use of the Dirichlet distribution limits LDA ability in modeling data with high diversity [202]. Another limi-

tation is the fact that the number of topics that are naturally present in the corpus should be specified ahead, which is, similar to specifying the K in LSI, can be computationally exhaustive [28].

3.2.4 Semantic Relatedness Methods

Semantic relatedness (SR) methods try to quantify the degree to which two concepts semantically relate to each other by exploiting different types of semantic relations connecting them [18]. The main intent is to mimic the human mental model when computing the relatedness of words. The human brain establishes semantic relatedness between words based on their meaning, or context of use [112]. For example, both words <*cow, horse*> refer to a mammal that has four legs, thus they can be considered related. Also, the words <*horse, car*> both refer to a means of transportation for humans, thus they can be considered related from that perspective. Another aspect the brain examines is the frequent association between words. Words that often appear together are likely to be related. For example, the words <*table, chair*> appear together frequently, giving the human brain an indication of relatedness.

A wide range of methods for measuring SR have been proposed in the literature. These methods infer words relatedness by exploiting massive amounts of textual knowledge to leverage all the possible relations that contribute to words similarity (e.g., Table 3.1). Such information is usually available in external knowledge sources including, linguistic knowledge bases (LKB) such as WORDNET [82], collaborative knowledge bases (CKB), such

as the online encyclopedia *Wikipedia* [236], or general Web search results, such as Google search [80].

SR has been applied to several NLP applications such as automated spelling correction [32], text retrieval [83], word sense disambiguation [203], question answering [4], and automatic speech recognition [214]. In what follows, we describe two methods of semantic relatedness that have been heavily investigated in related literature. These methods include Explicit Semantic Analysis (ESA) and Normalized Google Distance (NGD).

3.2.4.1 Explicit Semantic Analysis

ESA represents the meaning of a text as a high dimensional weighted vector of concepts, derived from *Wikipedia* [92]. In details, given a text fragment $T =< t_1, \cdots, t_n >$, and a space of *Wikipedia* articles (*C*), initially a weighted vector *V* is created for the text, where each entry of the vector v_i is the TFIDF weight of the term t_i in *T*. Using a centroid-based classifier [111], all *Wikipedia* articles in *C* are ranked according to their relevance to the text. Let k_j be the strength of association of term t_i with *Wikipedia* article c_j, where $c_j \in < c_1, c_2, \cdots, c_n >$ (*N* is the total number of *Wikipedia* articles), the semantic interpretation vector *S* for text *T* is a vector of length *N*, in which the weight of each concept is defined as:

$$S_i = \sum_{w_{i \in T}} v_i.k_j \tag{3.3}$$

Entries of this vector reflect the relevance of the corresponding articles to text *T*. The relatedness between two texts can then be calculated as the cosine between their corresponding vectors.

Among the different *Wikipedia*-based measures proposed in the literature, ESA has been proven to achieve the highest correlation with human judgment [185, 236]. In addition, ESA compares text fragments. This makes it a suitable approach for traceability tasks or even requirements engineering tasks in general [172]. In fact, due to its flexibility, ESA has been extended to work in cross-lingual retrieval settings, which can be considered as an extreme case of vocabulary mismatch [230]. Limitations of ESA can stem from the complexity of its implementation, as it requires downloading the whole *Wikipedia*, which requires substantial space requirements. In addition to the computational capabilities required for indexing such a large amount of data [92].

Table 3.1

Semantic Relations

Relation	**Description**	**Example**
Synonymy	Equivalent	*<sick, ill>*
Polysemy	Multiple meanings	*<charge>*
Hyponymy	Type-of	*<ambulance, car>*
Antonymy	Opposite	*<male, female>*
Meronymy	Part-of	*<room, hotel>*
Statistical	Co-occurrence	*<patient, hospital>*

3.2.4.2 Normalized Google Distance

The fuzzy set theory suggests that the degree of keywords' co-occurrence can be considered as a measure of their semantic relatedness [18]. Based on that, the Normalized

Google Distance (NGD) provides a method to estimate confidence scores between words using words' co-occurrences collected over Web search results (e.g., Google).

Formally, for each two terms being matched, a Google search query is initiated. The semantic relatedness between two terms $s(t_1, t_2)$ is then measured using the normalized Google Similarity Distance (NGD) introduced by Cilibrasi and Vitanyi in [40] as:

$$s(t_1, t_2) = \frac{\log(\max(D_1, D_2)) - \log(|D_1 \cap D_2|)}{\log(|D|) - \log(\min(D_1, D_2))} \tag{3.4}$$

where D_1 and D_2 are the numbers of documents containing t_1 and t_2 respectively and $|D_1 \cap D_2|$ is the number of documents containing both t_1 and t_2. The assumption is that pages that contain both terms indicate relatedness, while pages that contain only one of the terms suggest the opposite. Equation 3.5 [100] is often used to normalize NGD fit in the range [0 - 1]:

$$nNGD = e^{-2NGD(w_1,w_2)} \tag{3.5}$$

For example, the NGD between the two terms *patient* and *hospital* can be calculated as follows, a Google search is initiated for the terms *patient* and *hospital* separately. The search process returns 573,000,000 and 1,200,000,000 hits for both terms respectively (i.e., D_1 and D_2). Next, a search using the phrase *patient hospital* is requested. Google returns 335,000,000 hits (pages in which both *patient* and *hospital* appear) representing $|D_1 \cap D_2|$. Using Eq. 5, NGD $(patient, hospital) = 0.446$, given that Google search engine indexes approximately ten billion pages $(D = 10^{10})$.

In NGD, the smallest the distance, the closer the terms, hence $NGD(x, x) = 0$, and the distance between two completely unrelated terms (e.g. $|D_1 \cap D_2| = \phi$) is equal to ∞.

To quantify the similarity between different artifacts in the system, we initially calculate the pairwise NGD similarity between all the unique terms in the corpus. These values are normalized to fit in the interval [0 - 1], producing NDG. The value $sNGD = 1 - NGD(x, y)$ is then used to indicate the pairwise term similarity rather than dissimilarity (i.e., 1 means an exact match) [95]. These values are stored in a thesaurus similar to the synonyms thesaurus introduced earlier (VSM-T). Similarity between any two artifacts in the system can then be calculated using Eq. 3.2, where α_{ij} is equal to the sNDG value between the terms t_i and t_j.

NGD has been successfully applied in several NLP tasks such as search query prediction [36] and concepts mapping [95]. However, the quality of NGD can be highly affected by the noise usually returned by search engines due to the inherent ambiguity of some terms, and the lack of context when matching individual terms.

Table 3.2

Categories of Semantically-enabled IR Methods Used in our Analysis

	Description	Semantics	knowledge source	Related work
Baseline				
Vector Space Model	VSM	N/A	N/A	[222]
Semantic Augmented with Thesaurus				
VSM with domain thesaurus	VSM-T-TD	Synonyms	Domain Thesaurus	[121, 246]
VSM with general purpose thesaurus	VSM-T-WN	Synonyms	WordNet	
Semantic Augmented with POS				
VSM with Part-of-Speech tagging (nouns)	VSM-POS-N	Nouns	OpenNLP	[127] [39]
VSM with Part-of-Speech tagging (verbs)	VSM-POS-V	Verbs	OpenNLP	[127] [39]
Latent Semantic				
Latent Semantic Indexing	LSI	Synonyms, Polynyms	Corpus	[60]
Latent Dirichlet Allocation	LDA	Topics Modeling	Corpus	[28]
Semantic Relatedness				
Explicit Semantic Analysis	ESA	Synonymy, Hyponymy	Wikipedia	[92]
Normalized Google Distance	NGD	Antonym, Meronymy co-occurrence	Google	[40]

Summary The collection of methods presented in this section covers a wide spectrum of semantically-enabled IR methods that have been intensively used in software engineering research in general, and traceability research in particular. Some of these methods are focused on certain semantic relations such as *synonymy* (e.g., VSM-T), while other methods expand the range of relations to cover more semantic relations such as *polysemy*, *hyponymy*, and *meronymy*, and statistical associations such as co-occurrence of terms (e.g., LDA, LSI, ESA, and NGD). In addition, some of these methods use local knowledge sources, such as a domain thesaurus or the internal textual structure of the corpus, to derive their similarity scores (e.g., VSM-T-TD, LSI, and LDA), while other methods use external sources, such as WORDNET and *Wikipedia*, to estimate similarity (e.g., VSM-T-WN, ESA, and NGD).

Furthermore, The presented methods can be divided into explicit and latent. Explicit methods are explicit in the sense that they manipulate concepts grounded in human cognition (e.g. ESA and NGD) or import semantics explicitly from an external source such as a domain thesaurus (e.g. VSM-TD), or the grammatical structure of documents (e.g. VSM-POS). On the other hand, latent methods use statistical methods to derive latent semantic structures hidden in the natural language component of the system (e.g., LDA and LSI). Table 3.2 summarizes the attributes of the different methods described in this section.

3.3 Experimental Settings

The main objective of our experimental analysis is to systematically compare the performance of the various methods proposed in Table 3.2 in capturing traceability links

among different types of artifacts. Three datasets are used to conduct the experiment in this chapter including: *CM-1*, *eTour*, and *iTrust*.

To implement the different methods investigated in our analysis, we refer to several state-of-the-art implementations available online. In particular, ESA implementation was guided through several online implementations[1] including pre-processing tools for parsing Wikipedia dumps (e.g., WikiPrep) and carrying out ESA analysis. Wikipedia 2009 dumps were used in our implementation. To implement NGD, the client library Google.NET[2] was used to initiate Google queries and interpret returned responses. For the implementation of VSM-POS we used SharpNLP[3], a port of the Java OpenNLP library written in $C\#$. For VSM-WN we used WordNet.Net[4], a free open source .Net framework library for WORDNET written in $C\#$. For the implementation of LSI we used Bluebit Matrix Calculator[5], a high performance matrix algebra for .NET programming which provides routines for singular value decompositions, eigenvalues, and eigenvectors problems. JGibbLDA[6], a Java implementation of LDA is used for topic modeling. This particular implementation uses Gibbs Sampling for parameter estimation and inference [103].

We use two sample artifacts (q, d) from the *iTrust* dataset as an illustrative example to guide our analysis. These artifacts are shown in Figure 3.1 a and b respectively. q represents a requirement of the system (req. 3.1.1), it describes a basic *login* functionality. d is a method that versifies user's login information. There is a valid trace link between

[1] http://www.cs.technion.ac.il/∼gabr/resources/code/
[2] https://developers.google.com/gdata/client-cs
[3] http://sharpnlp.codeplex.com/
[4] http://opensource.ebswift.com/WordNet.Net/
[5] http://www.bluebit.gr/net/
[6] http://jgibblda.sourceforge.net/

q and d. Some methods require artifacts to be indexed before matching them. Both q and d were indexed using the indexing process described earlier [169]. The output of the indexing process is shown in Figure 3.1 c and d.

Performance of different methods is presented in precision/recall curves over various threshold levels ($< .1, .2, ..., 1 >$) [121]. A higher threshold level means a larger list of candidate links, i.e., more links were considered in the analysis. Wilcoxon Signed Ranks test is used to measure the statistical significance of the results [65]. This test is applied over the combined samples from two related samples or repeated measurements on a single sample (before and after effect). The IBM SPSS Statistics software package is used to conduct the analysis. We use $\alpha = 0.05$ to test the significance of the results. Note that different IR methods are applied independently, so there is no interaction effect between them.

Table 3.3

DiffAR (Eq. 1.4) Values Taken at 0.7 Threshold

	baseline	Thesaurus Support		POS Support		Latent Semantic		Semantic Relatedness	
	VSM	VSM-T-TD	VSM-T-WN	VSM-POS-N	VSM-POS-V	LSI	LDA	NGD	ESA
iTrust	0.3	.21	0.17	0.33	0.33	0.01	0.01	0.02	0.11
eTour	0.22	.16	0.1	0.39	0.32	0.01	0.01	0.07	0.12
CM-1	0.1	.09	0.06	0.21	0.2	0.01	0.01	0.01	0.09

3.1 Functions

...

...

3.1.3 Login:

Each patient should authenticate himself using his credentials. In case of forgotten password see 3.1.4.

3.1.4 Forget Password:

The system should provide a functionality for recovering user's password...

a) SRS document

```
//Validate Patient's email and password
public bool Login_OnClick(object sender)
{
    if
     (ValidateUser(EmailText.Text))
       &&
       (ValidateUser(PasswordText.Text))
           return true;
      else
       {
         Exception.Text ="Login failed";
         return false;
       }
}
```

b) Code Element

<login>
<patient>
<Authenticate>
<credential>
<forget>
<password>

c) Profile of requirement 3.1.3

<patient>
<email>
<password>
<login>
<user>
<validate>
<fail>

d) Profile of Method Login_OnClick

Figure 3.1

Example 1: A Traceability Link

3.3.1 Semantic-augmented Methods

We start our analysis by examining the performance of the semantic-augmented methods including VSM with thesaurus support (VSM-T) and VSM with *Part-of-Speech* tagging (VSM-POS).

3.3.1.1 Vector Space Model with Thesaurus Support

Two methods are investigated under his category: VSM-T-DT and VSM-T-WN. We propose an optimization algorithm in order to specify acceptable approximations for α in Eq. 3.2. This algorithm is based on maximizing the recall. The main assumption is that IR-based tracing tools favor recall over precision. This is mainly because commission errors (false positives) are easier to deal with than omission errors (false negatives) [121]. The algorithm starts from $\alpha = 0$, gradually increasing this value by .05 each time, and monitoring the recall over constant threshold levels. The value of α that achieves the highest average recall at lowest threshold level (i.e., highest possible precision) is considered a local maximum. We run this algorithm over our three experimental datasets. Results show that average similarity coefficients of $\alpha = .43$ and $\alpha = .81$ achieve the best recall in VSM-T-WN and in VSM-TD respectively. However, while this kind of optimization achieves acceptable performance levels, more sophisticated approaches, such as assigning different weights based on human judgment or statistical similarity analysis over WORDNET, can be used. However, such analysis is beyond the scope of this chapter.

Figure 3.4 shows the performance of the two methods in comparison to the VSM baseline in all three experimental datasets. Statistical analysis results are shown in the *Semantic*

Augmented (Thesaurus) section of Table 3.4. Analysis shows that the VSM baseline starts with relatively higher precision and recall at lower threshold levels. VSM-T-TD is able to catch up halfway through, keeping up the good performance until almost achieving a 100% recall at higher threshold levels, while basic VSM stopped at 93.3% recall. Figure 3.4 also shows the fast drop in the precision of VSM, while VSM-TD shows a more gradual decrease in the precision with the increase of the recall. Results also show the poor performance of VSM-T-WN, which performs significantly worst than VSM and VSM-T-TD in all three datasets. While VSM-T-WN is able to hit a 100% recall at higher threshold levels, it retrieves so many false positives taking the precision down to significantly lower levels.

The above analysis shows that the explicit introduction of synonyms in VSM improves the overall recall. It also has a positive effect on the accuracy by keeping acceptable precision levels at higher recall levels. The poor performance of VSM-T-WN in comparison to VSM-T-TD can be explained based on the fact that WORDNET is a general purpose thesaurus; no domain knowledge is available to guide the synonym extraction process. Therefore, this method introduces a high noise-to-signal ratio that leads to retrieving a large number of false positives. In contrast, the domain thesaurus in WSM-T-TD was generated using terms from within the corpus, so noise levels were kept under control. Also, the domain knowledge helped to deal with non-English words that the English dictionary WORDNET fails to handle, especially in the *eTour* dataset where Italian words were used.

To further confirm our findings, we refer to our example in Figure 3.1. The similarity scores between *q* and *d* given by VSM, VSM-T-TD, and VSM-T-WN were 0.54, 0.63, and

0.47 respectively. Using VSM-T-WN, *q*'s vector has been expanded with the following synonyms:

- *credential*: *certificate*
- *authenticate*: *formalize*, *corroborate*
- *validate*: *formalize*, *corroborate*
- *password*: *watchword*, *word*, *parole*, *countersign*
- *user*: *exploiter*
- *fail*: *miscarry*, *neglect*, *die*, *go*, *break*, *break*, *flunk*, *bomb*
- *forget*: *bury*, *block*, *leave*

This list shows that WORDNET introduces so many domain irrelevant terms (e.g., *parole*, *miscarry*). While such enrichment might have a positive influence on the recall, especially in retrieving some of the hard-to-trace requirements [94], it often causes a significant drop in the accuracy, which is reflected in the fast drop in the precision values of VSM-T-WN at higher threshold levels. This is also clearly shown by the DiffAR (Eq. 1.5) values in Table 3.3 which show that VMS-T-WN is the least successful in distinguishing between true and false links.

In contrast, using VSM-T-TD, the following synonym pairs were manually identified based on the domain's context: <*password*, *credential*>, <*email*, *credential*>, <*authenticate*, *validate*>, and <*patient*, *user*>. It is important to point out here that VSM-T-TD is also prone to noise. For example, the synonym pair <*patient*, *user*> might cause some confusion, as in the *iTrust* dataset, the term *user* might refer to individuals other than patients, such as visitors or doctors. However, regardless of that small amount of noise, VSM-T-TD still gives a higher similarity score between *q* and *d*, which is desirable since (q, d) is

actually a correct link. In addition, the DiffAR values of VSM-T-TD, shown in Table 3.3, show that this method achieves an acceptable distinction between true and false links in comparison to VSM and VSM-T-WN.

3.3.1.2 VSM with Part-of-Speech Tagging

Under this category we analyze the performance of VSM-POS-N, in which only nouns are considered in the indexing process, and VSM-POS-V, in which only verbs are considered. POS is applied before indexing to preserve the grammatical structure of the text. In case of source code, this process depends heavily on the availability of free-text comments that can be correctly parsed. After indexing, documents are matched using the standard cosine similarity (Eq. 3.1).

Performance precision/recall curves for running the two VSM-POS methods over our three experimental datasets are shown in Figure 3.5. Statistical analysis results are shown in the *Semantic Augmented (POS)* section of Table 3.4. Results show that traceability link retrieval is heavily affected by the grammatical filters. In both cases, considering only one part of speech has a significantly negative effect on recall in all three datasets. Even though considering only nouns has relatively less negative impact on the performance than considering only verbs, it still fails to match the baseline's recall. The relatively better performance of both methods in *CM-1* can be explained based on the fact that *CM-1* is a requirements-to-design dataset, and free text is used to describe artifacts at both sides of the traceability link. This allowed the POS tagger to generate more accurate lists of candidate links for this particular dataset in comparison to the two other datasets. Results also show

that considering only nouns in the indexing process achieves a significantly higher recall than indexing verbs only. This suggests that nouns carry more information value when retrieving traceability links. However, such information value is not sufficient enough to achieve optimal recall levels.

In general, it can be concluded that this kind of augmentation fails to achieve a satisfactory performance in the domain of automated tracing. However, if high precision levels are favored over recall, these methods can be useful as they tend to filter out a large portion of unwanted noise, usually caused by some irrelevant terms generated by the indexing process. This behavior is clearly reflected in the DiffAR values (Table 3.3) which show that, VSM-POS methods generate the highest values in terms of distinguishing between true positives and false positives. However, this success does not give them an edge over the baseline, as they significantly fail to outperform basic VSM.

Considering our example in Figure 3.1, VSM-POS-V reduces *q* and *d* vectors to <*login*, *Authenticate*, *forget*> and <*login*, *validate*, *fail*> respectively. In contrast, VSM-POS-N reduces *q* to <*patient*, *credential*, *password*> and *d* to < *patient*, *user*, *email*, *password*>. Using VSM-POS-V, *q* and *d* only match at <*login*> taking the similarity down to 0.224, and VSM-POS-N has two matches <*patient*, *password*> also taking the similarity score of the true (q, d) link down to 0.336.

3.3.2 Latent Semantic Methods

Under this category we analyze the performance of the latent methods LSI and LDA. To approximate an optimal value of *K*, LSI is ran iteratively while the *K* value is gradually

increased by 5 after each iteration. *K* values that produce globally better precision/recall, averaged over all the instances of each dataset, are kept. Running this optimization procedure over our three experimental datasets produced *K* values of 35, 40, and 45 for *iTruts*, *eTour*, and *CM-1* respectively.

We follow a similar experimental approach to approximate an optimal number of topics (*K*) for LDA. In particular, K is initially set to 40 topics. The document-topic distribution matrix of each artifact in the system is then generated. A cosine comparison (Eq. 3.2) is conducted to capture matching in the latent topic structures of different artifacts, generating candidate traceability links. The value of K is then increased by 40 and the process is repeated. This particular step size of 40 is the minimum value that yields noticeable changes in the recall. As mentioned earlier, We tie optimality to recall in our analysis. Therefore, we follow a hill climbing approach to monitor the changes in the recall, best recall values were detected at *K* values of 160, 180, and 180 for *iTrust*, *eTour*, and *CM-1* respectively. At this range of K, topics tend to be more distinguishable from each other, which makes these particular values nearly optimal for traceability analysis.

It is important to point out that the complexity of the study grows exponentially with the inclusion of other LDA parameters such as α and β. Therefore, at this stage of our analysis, we fix these values. This strategy is often used in related research to control for such parameters' effect [101, 161, 243]. In particular, values of $\alpha = 50/K$ and $\beta = 0.1$ are used. These heuristics have been shown to achieve satisfactory performance in the literature [103, 253].

The performance of LSI and LDA over our three datasets in comparison to the baseline is shown in Figure 3.6. Statistical analysis results are shown in the *Latent Semantics* section of Table 3.4. The results show that, in comparison to VSM, in all three datasets both methods achieve relatively better recall. However, the only statistically significant improvement in recall over the baseline is achieved by LSI over *CM-1*. The results also show that this improvement in the recall has taken the precision down to significantly lower levels in all three datasets. Further more, a closer look at the recall and precision curves shows that, in *iTrust* and *CM-1*, LSI manages to outperform LDA at lower threshold levels. This difference in the performance is more obvious in the *iTrust* dataset where LSI does significantly better than LDA in terms of precision and recall. In the *eTour* dataset, both LDA and LSI achieve an interchangeably good performance before reaching the maximum recall and minimum precision point, i.e., all the links are retrieved (considered relevant).

Applying the latent methods over q and d in Figure 3.1 shows that both methods produce relatively low similarity scores. LSI returns a similarity score of 0.032 and LDA returns a score of 0.007. These similarity scores were generated at class-granularity level. For instance, using LDA, the topic distribution of the class that contains the function `Login_OnClick` (Figure 3.1-b) was matched with the topic distribution of requirement 3.1.3 (Figure 3.1-a).

The relatively higher score of LSI might be explained based on its operation, such as a detected synonym relation between <*credential*, *password*>. However, there is no clear indication if that is actually the case, or just a mathematical coincidence. In general, the poor performance of latent methods can be ascribed to their internal operation.

When dealing with software artifacts, the amount of knowledge available in a corpus is not expressive enough to produce meaningful document-topic or document-term matrices for LDA and LSI, which makes these methods prone to mathematical noise. This behavior was clearly reflected in the DiffAR values shown in Table 3.3. In all three datasets, both latent methods are the least successful in distinguishing between true and false links, achieving the smallest DiffAR values among other investigated methods.

3.3.3 Semantic Relatedness Methods

Under this category, the methods of Explicit Semantic Analysis (ESA) and Normalized Google Distance (NGD) are investigated. Performance of both methods in comparison to the baseline is shown in Figure 3.7. Statistical analysis results are shown in the *Semantic Relatedness* section of Table 3.4. Results show that in all three datasets, both methods are able to hit a 100% recall at higher threshold levels. However, this improvement over the baseline's recall is statistically insignificant ($\boldsymbol{p}$ = .093 for NGD and $\boldsymbol{p}$ = .515 for ESA). On the other hand, the precision is affected negatively due to the high number of false positives, which is more obvious in NGD where the precision at 100% recall hits a minimum.

The diagrams also show that ESA achieves better precision and recall than NGD in all datasets. This can be explained based on the fact that *Wikipedia*, being a semi-structured source of knowledge, cancels a high ratio of noise usually returned by search engines, thus achieving a higher precision. Another potential reason for the relatively poor performance of NGD is the oversimplification of the problem. While ESA utilizes a smarter approach

for extracting relatedness measures, NGD simply relies on hit counts to derive similarity, ignoring several inherent problems related to term ambiguity.

To gain more insights into these methods, we refer to our example in Figure 3.1. ESA compares vectors of text, thus the domain knowledge is somewhat preserved through the context. For example the word *fail* in the context <*user*, *login*, *fail*> obviously refers to a failure in the login process. In contrast, due to the lack of context in NGD, terms such as *user* and *fail* can refer to so many types of failure. This behavior was reflected in the DiffAR values shown in Table 3.3, which shows that ESA is more successful in differentiating between true links and false links.

3.3.4 Inter-category Comparison

We conduct a comprehensive analysis to compare the performance of the best performing methods from each category. Results are shown in Figure 3.8. Pairwise statistical analysis is shown in the *Inter-Category* part of Table 3.4. In terms of recall, in all three datasets, VSM-T-DT, ESA, and LSI were able to reach a maximum recall at higher threshold levels, with VSM-T-DT achieving significantly higher precision, followed by ESA, which in turn significantly outperforms LSI. In terms of precision, VSM-POS-N achieves highest precision, outperforming all other methods significantly. However, it fails to achieve an acceptable recall.

In terms of browsability measures, Figure 3.2 and Figure 3.3 show the MAP (Eq. 1.4) and Lag (Eq. 1.6) results of the best performing methods from the different categories of semantically-enabled IR methods. In general, results show that methods that achieve

a reasonable performance in terms of quality, such as VSM-T-TD and ESA, also tend to achieve a good performance in terms of browsability.

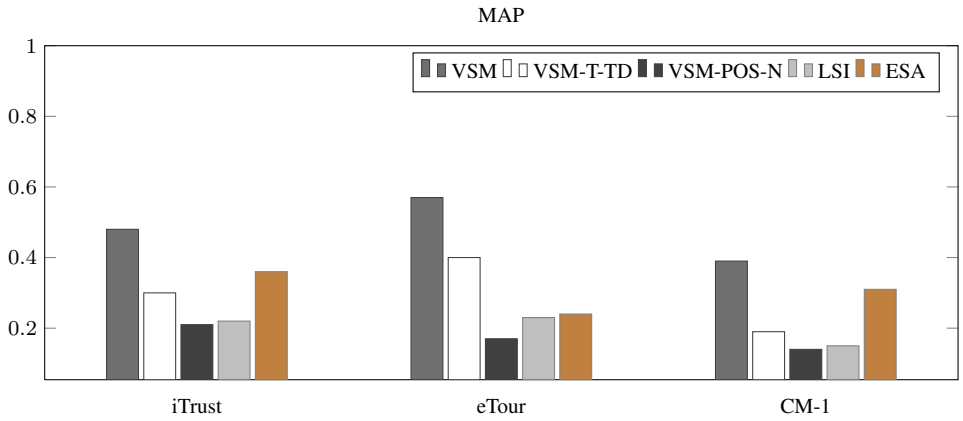


Figure 3.2

MAP Values in iTrust, eTour, and CM-1

In particular, in terms of MAP, Figure 3.2 shows that VSM-T-DT and ESA achieve an interchangeably good performance over the experimental datasets, ESA outperforms VSM-T-TD in *iTrust* and *CM-1* while VSM-T-TD outperforms ESA in *eTour*. The results also show the constantly poor performance of LSI and VSM-POS-N, achieving a significantly lower MAP in all datasets. LSI tends to scatter true positives all over the ranked list of candidate links, with higher concentration of these links at the bottom of the list, thus taking the average precision (AP) value down for each query. VSM-POS-N, on the other hand, captured a smaller number of correct links, thus increasing the number of links with 0 precision (false negatives) in Eq. 1.5.

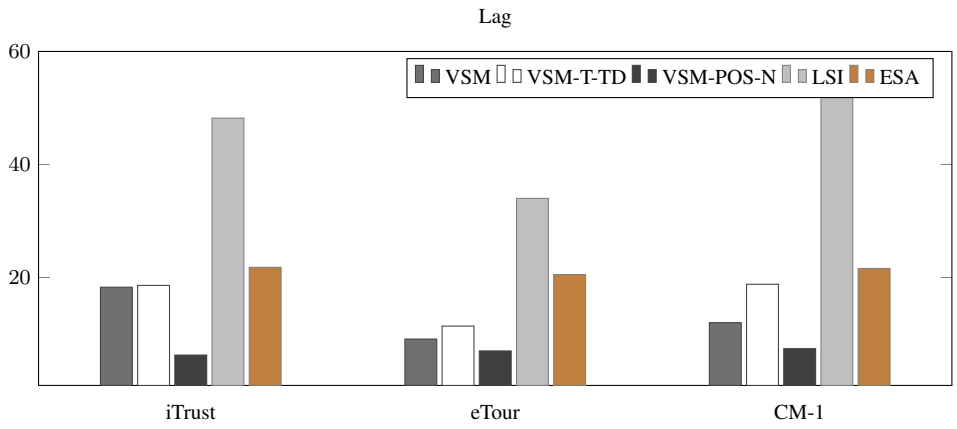


Figure 3.3

Lag Values in iTrust, eTour, and CM-1

Similar patterns are observed in the Lag results, shown in Figure 3.3. The method that achieved the highest precision (VSM-POS-N) acquired the lowest Lag values (i.e., smaller number of false positives separate true positives). Results also show that VSM-T-TD achieves a comparable performance to the baseline, also outperforms ESA and LSI in all three datasets. While ESA manages to keep very close performance in both *iTrust* and *CM-1*, in the *eTour* dataset ESA performance is significantly worst. In contrast, LSI achieves the worst performance in all three datasets. This means that LSI tends to scatter the correct links all over the list with higher numbers of false positives separating them.

Finally, while identifying a winning technique is not a main goal of our analysis, in terms of the primary and secondary performance measures, our overall analysis results lean toward announcing VSM-T-TD as the most reliable semantically-enabled IR method for traceability link recovery.

3.3.5 Limitations

In our experiment, there are minimal threats to construct validity as standard IR measures (recall and precision), which are used extensively in requirements traceability research, are used to assess the different methods investigated in this chapter. These measures are also complemented by another set of secondary measures (MAP, DiffAR, and Lag (Section 1.5.2)) that are used to provide more insights into the browsability of the generated lists of candidate traceability links. We believe that these two sets of measures sufficiently capture and quantify the different performance aspects of the various methods evaluated in this study.

In terms of external validity, a major threat comes from the datasets used in our experiment. In particular, two of these datasets were developed by students and may not be representative of a program written by industrial professionals. Also, all three of our datasets are limited in size, which raises some scalability concerns. However, we believe that the use of three datasets, from different application domains, helps to mitigate these threats. Finally, specific design decisions and heuristics used during the experiment can also limit the study's applicability. Such decisions include using *Wikipedia* 2009 in ESA, using TFIDF weights in our baseline, the decision of only considering verbs and nouns in VSM-POS, and the heuristic values of α and β used in LDA.

Finally, regarding the internal validity of our experiment, a potential threat comes from our specific implementation of the different methods investigated in this chapter. However, we believe that using freely available open source tools and libraries in our implementations helps to mitigate this threat. It also makes it possible to independently replicate

our study. In addition, an experimental bias might stem from the fact that some of the procedures in our experiment were carried out manually. For instance, the local domain thesaurus in VSM-T-TD was created manually by our researchers, based on their understanding of the system, which might raise some subjectivity concerns that can affect the internal validity of our study.

3.4 Discussion and Impact

Capturing and maintaining accurate lists of requirements traceability links is vital to managing requirements in the multiple phases of the software development process [97, 121]. In this chapter, we investigated the performance of several semantically-enabled IR methods in bridging the semantic gap that often appears in the system as a direct result of software evolution [152, 153]. In particular, we experimentally assessed the effect of different semantic schemes on the performance of various IR-based traceability methods.

Our results revealed that explicit semantic methods (VSM-T, VSM-POS, ESA, and NGD) tend to do a better job in recovering traceability links than latent methods (LSI and LDA). Latent methods, though able to achieve higher recall levels, they often fail to compete with the precision of other methods. This can be explained based on the fact that lexicons and syntax of NL documents differ from those of software artifacts. Source code is more constrained and less varied than natural language, which makes it more regular and repetitive [116, 159]. This limits the ability of latent methods to extract hidden semantics schemes using statistical and probabilistic models. In fact, latent methods were initially developed to work with contents of large document collections [28, 103, 192, 262].

However, software systems are often much smaller than NL text corpora, depriving such methods of context, and causing them to behave randomly, even when calibrated using settings that usually achieve adequate performance over natural language corpora [116, 223]. For instance, poor parameter calibration or wrong assumptions about the nature of the data often lead a method such as LDA to generate several irrelevant or duplicated topics [202].

Probably the most interesting observation in our analysis is that considering more semantic relations in retrieval does not necessarily lead to a better tracing performance. Instead, a local and a more focused semantic support is expected to serve the automated tracing problem better. This was clearly reflected in the performance of VSM-T and ESA, while both methods achieved a relatively good performance, VSM-T managed to keep a significantly higher precision in all three datasets, also significantly higher recall in both *eTour* and *CM-1*. In particular, our analysis shows that methods which only consider the semantic relation of *synonymy*, tend to be the most reliable for traceability link recovery. This can be explained based on the fact that software artifacts are not as semantically rich or complex as free text. In fact, it has been observed that developers tend to shy away from using fancy language when writing specifications. Instead, software artifacts are usually expressed in a simplified form of the natural language, with a smaller vocabulary set and simplified grammars [77]. In addition, source code developers tend to abbreviate names; causing concepts to be denoted through their full name as well as multiple abbreviations [9, 61], thus increasing the density of synonymy relations in software systems.

Our results also show that external sources of knowledge such as *Wikipedia* or WORDNET tend to increase the level of noise in retrieval. This can be explained based on the fact

that often a coherent vocabulary structure, derived from the system application domain, is used through out the project's life-cycle. Therefore, as shown in our analysis, using external and general-purpose sources of knowledge to enrich the system's vocabulary will most likely corrupt that coherent structure with unrelated terms, thus causing a significant decline in the precision of IR methods. In addition, as observed earlier, synonyms generated by abbreviating domain names, seem to be the most occurring semantic relation during software evolution. Being domain-specific, such synonyms are often not included in general purpose dictionaries or knowledge sources.

Finally, in terms of tool support, our results reveal how different methods, at different levels of semantic support, might be useful in certain contexts of requirements management. For example, in tools where accuracy is the main concern, methods that achieve significant precision levels might be useful (e.g., VSM-POS). However, in safety-critical systems, which imposes special demands on ensuring quality and reliability of the system, methods that achieve higher recall levels might be more appropriate [45]. In addition, if practicality is a major concern, then methods that utilize external knowledge sources such as *Wikipedia* or WORDNET should be avoided. Such methods require relatively higher time and space requirements to function properly (e.g., initiating multiple Web search requests or long search queries in NGD, or downloading and indexing *Wikipedia* in ESA).

3.5 Related Work

Several IR-based traceability link recovery methods have been proposed in the literature. Next we selectively review some of the seminal work in this domain over the last

decade. Initial work on IR-based traceability was conducted by Antoniol et al. [12]. The authors used Probabilistic Network Model (PN) and basic Vector Space Model (VSM) to recover traceability links between source code and free text documents. This work provided an initial evidence of the practicality of IR methods as a potential solution for the automated tracing problem. Marcus and Maletic repeated the same case study using Latent Semantic Indexing (LSI) [178]. They compared the performances of LSI with VSM and PN. Results showed that LSI can achieve a comparable performances without the need for stemming.

Huffman-Hayes et al. [120] used two different variants of VSM including retrieval with key phrases and VSM with thesaurus support, to recover traceability links between requirements. The former approach was found to improve recall, however it affected precision negatively. On the other hand, VSM with thesaurus support resulted in improved recall and precision. A more recent work by the same authors addressed issues related to improving the overall quality of the automated tracing process [121]. In particular, the authors analyzed the performance of several IR methods including VSM, VSM with thesaurus support, and LSI, and incorporating relevance feedback from human analysts in the retrieval process. Results showed that using analysts' feedback to tune the weights in the term-by-document matrix of the vector space model improved the final tracing results.

Settimi et al. [226] compared the performance of several IR techniques in tracing UML models. In particular, the authors applied VSM and one of its variants that uses pivot normalization to trace requirements to UML artifacts, code, and test cases. The results raised some concerns about the adequacy of the IR-based paradigm in solving the trace-

ability problem. However, they found that such methods can be used to alleviate some of the manual effort in requirements tracing tasks. Similarly, Oliveto et al. [199] compared the performance of several IR-based traceability recovery methods including the Jensen-Shannon (JS) method, VSM, LSI, and LDA. Results showed that JS, VSM, and LSI were almost equivalent in that they captured almost the same information. However, LDA achieved lower accuracy.

Cleland-Huang et al. [46] introduced three enhancement strategies (hierarchical modeling, logical clustering of artifacts, and semi-automated pruning) to improve the performance of the probabilistic network model. Results showed that such strategies can be used effectively to improve trace retrieval results and increase the practicality of tracing tools. Similar to this work, in our previous work [195], we proposed an approach based on the cluster hypothesis to improve the quality of candidate link generation for requirements tracing. The main assumption was that correct and incorrect links can be grouped into high-quality and low-quality clusters respectively. Result accuracy can thus be enhanced by identifying and filtering out low-quality clusters. Evaluating this approach over multiple datasets showed that it outperformed a baseline pruning strategy.

Lormans and van Deursen [163] used a new strategy, based on LSI, to trace requirements to test case specifications and design artifacts. Their experimental analysis on three case studies provided an evidence that LSI can increase the insight in a system by means of reconstructing the traceability links between the different artifacts. Later, Asuncion et al. [16] employed topic modeling through the use of Latent Dirillect Allocation (LDA)

to recover different types of traceability links. Results showed that the combination of traceability with topic modeling can be useful in practice.

Gibiec et al. [94] used a web-based query expansion algorithm to bridge the vocabulary gap in the system. Evaluating this approach over a group of healthcare datasets showed its ability to improve the traceability of hard-to-trace requirements. Similarly, in our earlier work [172], we introduced semantic relatedness as an approach for traceability link recovery. Results showed that the *Wikipedia*-based ESA achieves a balance between LSI and VSM. It significantly outperforms the recall of VSM and the precision of LSI in most cases, showing more stable performance at different threshold levels. In addition, we conducted a preliminary analysis of VSM with *Part-of-Speech* tagging in recovering traceability links [168]. Results showed that POS could not beat basic VSM in terms of precision and recall. However, a more recent work on POS was conducted by Capobianco et al. [34]. Analysis of this approach over five software artifact repositories indicated that nouns-based indexing of software artifacts significantly improves the accuracy of IR-based traceability recovery methods.

3.6 Conclusions and Future Work

In this chapter, we conducted an experimental analysis to assess the performance of various semantically-enabled IR methods, including semantic-augmented, latent semantic, and semantic relatedness methods, in capturing requirements traceability links in software systems.

The performance of the different methods in terms of results' quality and browsability was compared to the basic VSM as an experimental baseline. Results showed that explicit semantic methods (VSM-T, VSM-POS, ESA, and NGD) tend to outperform latent methods (LSA and LDA). In addition, results revealed that methods that use selective indexing based on the lexical form of terms (VSM-POS), cancel a considerable amount of textual noise, thus achieve the best precision. However, such methods often suffer on the recall as a considerable amount of information is lost when filtering out other parts of speech. Results also showed that considering more semantic relations in retrieval does not necessarily lead to improved performance. Instead, a more focused explicit semantic support, in particular synonyms from a domain-specific thesaurus, is expected to achieve more adequate performance levels.

Research directions to be pursued in our future work include:

- Automated tracing methods: Our work in this chapter is limited to VSM-based methods. In our future work, we are interested in exploring other semantically enabled methods that apply different semantic schemes to the problem. For instance, we are interested in assessing the performance of ontology-based traceability tools which have been shown to achieve satisfactory performance in the domain of automated tracing [107, 263].
- Requirements engineering tasks: In this chapter, we have limited our analysis to the requirements traceability problem. In our future work, we are interested in studying the performance of semantically-enabled methods in supporting other requirements engineering tasks in which the IR paradigm is often employed. Such tasks include for example, reusable requirements retrieval [168], requirements discovery [25], and evolution [21].
- Tool support: Our analysis in this chapter suggested several guidelines for the design and development of practical automated tracing tools. It is in the scope of our future work to implement these findings in a working prototype to support various requirements engineering tasks besides traceability. In addition, a working prototype will allow us to conduct human studies to assess the usability of different methods in practice.

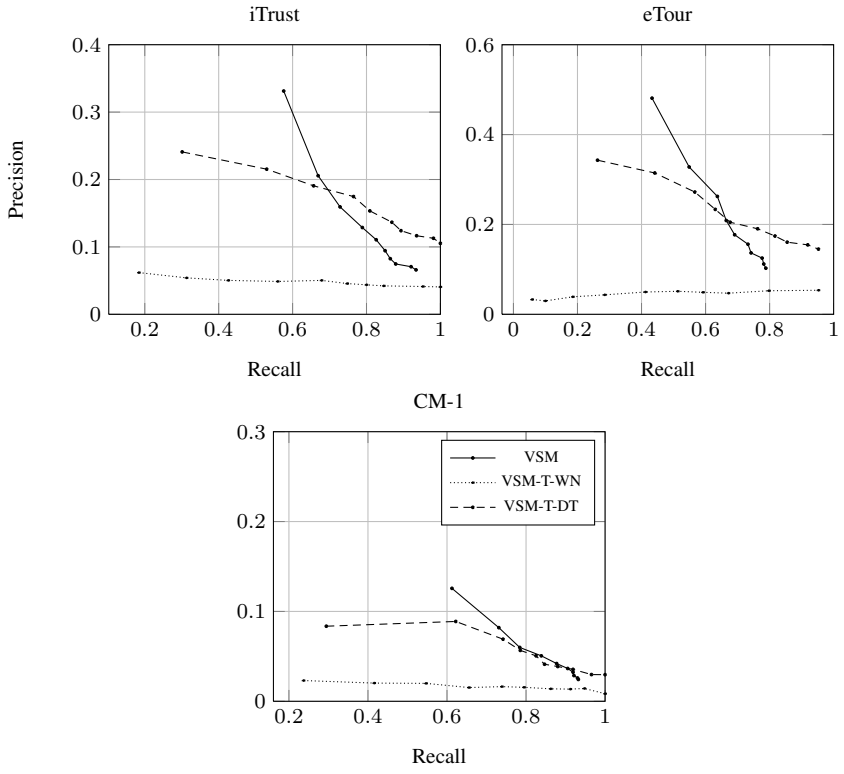


Figure 3.4

VSM-T Methods Performance

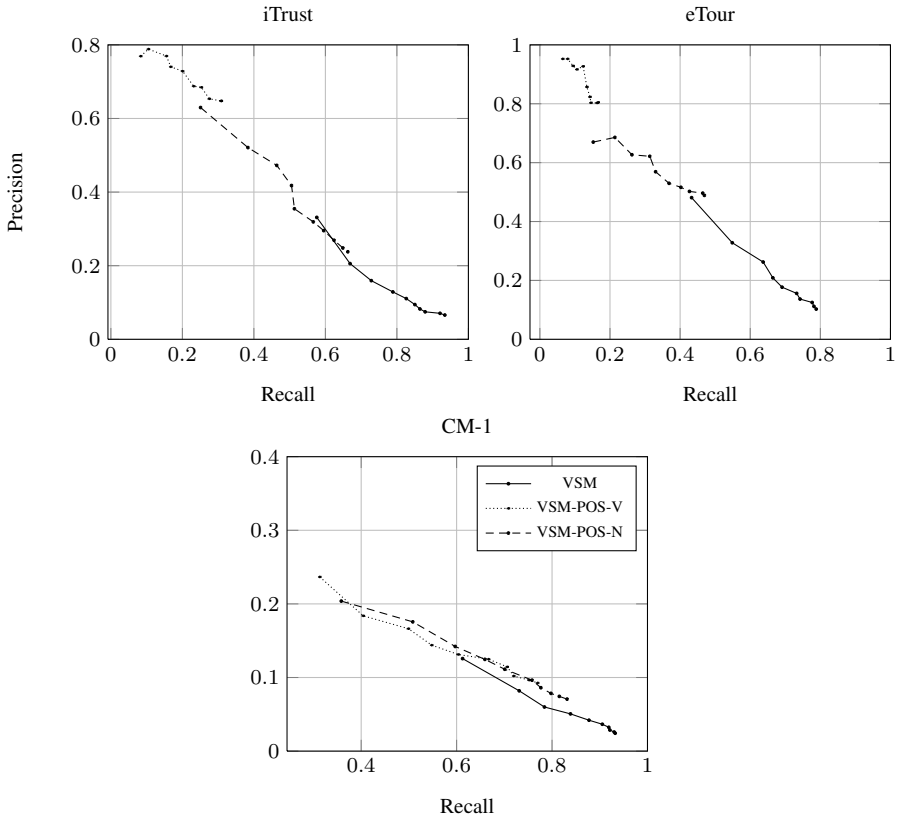


Figure 3.5

VSM-POS Methods Performance

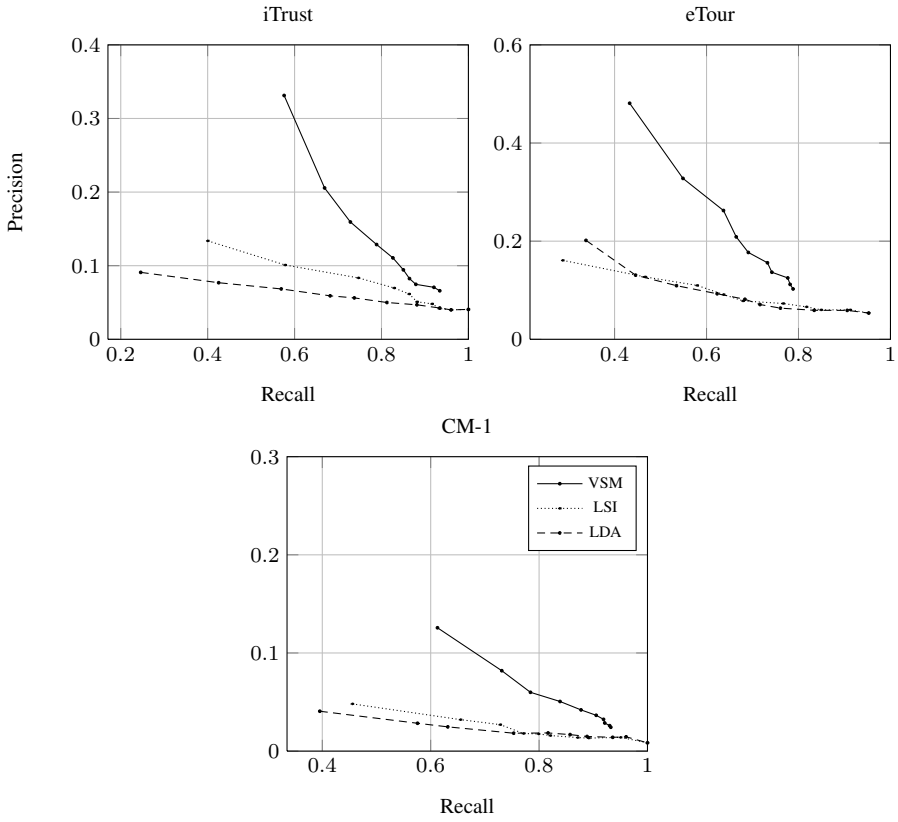


Figure 3.6

Latent Semantic Methods Performance

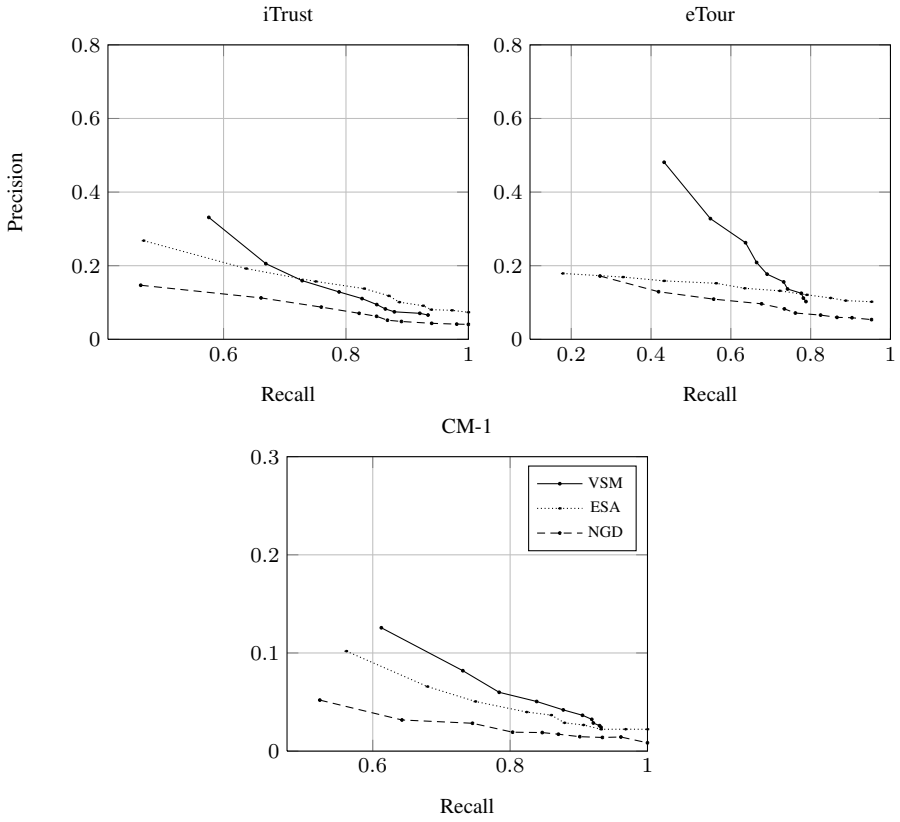


Figure 3.7

Semantic Relatedness Methods Performance

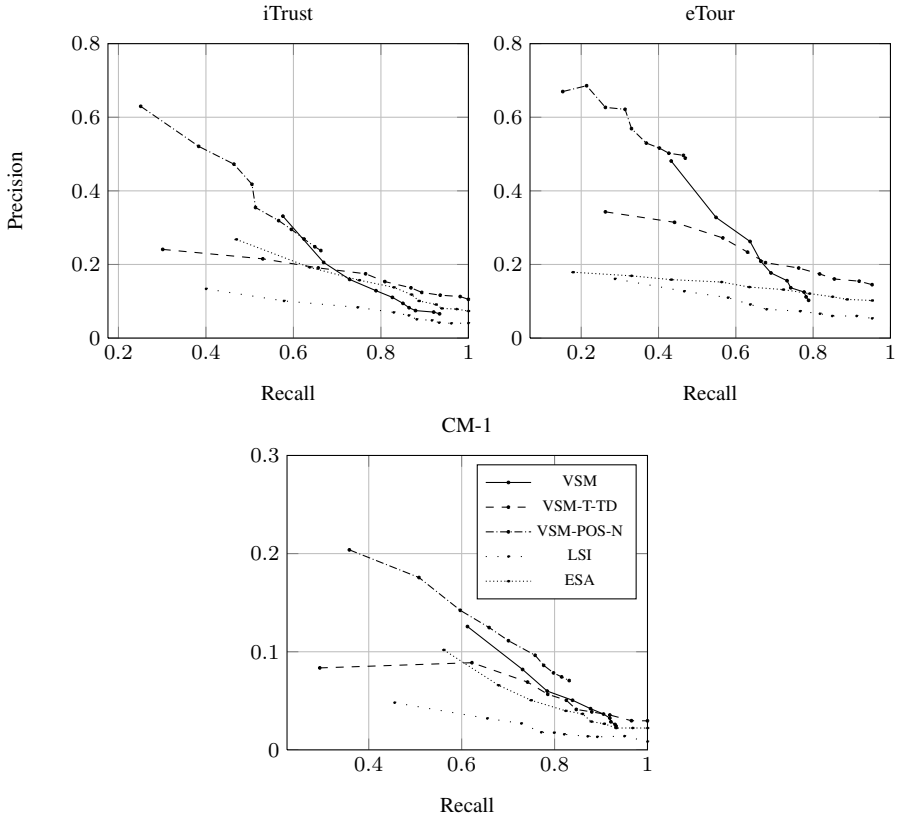


Figure 3.8

Comparing Best Performing Methods Performance

Table 3.4

Wilcoxon Signed Ranks Test results (***p**-values* at $\alpha = .05$)

	iTrust		eTour		CM-1	
	Recall (Z, *p*)	Precision (Z, *p*)	Recall (Z, *p*)	Precision (Z, *p*)	Recall (Z, *p*)	Precision (Z, *p*)
VSM x VSM-T-TD	(-.357, .721)	(-1.785, .074)	(-.255, .799)	(-1.580, .114)	(-1.784, .074)	(-1.785, .074)
VSM x VSM-T-WN	(-2.293 .022)	(-2.803 .005)	(-2.293 .022)	(-2.803 .005)	(-2.191 .028)	(-2.803 .005)
VSM-T-TD x VSM-T-WN	(-2.666 .008)	(-2.803 .005)	(-2.666 .008)	(-2.803 .005)	(-2.666 .008)	(-2.803 .005)
VSM x VSM-POS-N	(-2.803, .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)
VSM x VSM-POS-V	(-2.803, .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)
VSM-Noun x VSM-Verbs	(-2.803, .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)	(-2.803 .005)
VSM x LSI	(-.866, .386)	(-2.803, .005)	(-.459, .646)	(-2.803, .005)	(-2.090, .037)	(-2.803 .005)
VSM x LDA	(-1.478, .139)	(-2.803, .005)	(-.051, .959)	(-2.803, .005)	(-1.784, .074)	(-2.803, .005)
LSI x LDA	(-2.490, .013)	(-2.380, .017)	(-1.599, .110)	(-.652, .515)	(-.652, .515)	(-.178, .859)
VSM x ESA	(-1.580, .114)	(-.765, .444)	(-.866, .386)	(-2.803, .005)	(-.968, .333)	(-2.803 .005)
VSM x NGD	(-1.682, .093)	(-2.803, .005)	(-.663, .508)	(-2.803, .005)	(-1.479, .139)	(-2.803, .005)
ESA x NGD	(-.652, .515)	(-2.803, .005)	(-2.666, .008)	(-2.803, .005)	(-2.547, .011)	(-2.803, .005)
VSM-T-TD x VSM-Nouns	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.701, .007)	(-2.803 .005)
ESA x VSM-T-TD	(-2.429, .015)	(-2.599, .009)	(-2.666, .008)	(-2.803, .005)	(-2.666, .008)	(-2.803 .005)
ESA x VSM-Nouns	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-1.988 .047)
LSI x VSM-T-TD	(-2.192, .028)	(-2.803, .005)	(-1.599, .110)	(-2.803, .005)	(-.652, .515)	(-2.803 .005)
LSI x VSM-Nouns	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803, .005)	(-2.803 .005)
ESA x LSI	(-2.668, .008)	(-2.803, .005)	(-2.547, .011)	(-2.803, .005)	(-2.666, .008)	(-2.803, .005)

CHAPTER 4

CLUSTER BASED RETRIEVAL

Due to the inherent trade-off between recall and precision, IR-based automated tracing methods cannot achieve a high coverage without also retrieving a great number of false positives, causing a significant drop in result accuracy. In this chapter, we propose an approach to improve the quality of candidate link generation for the requirements tracing process. We base our research on the cluster hypothesis which suggests that correct and incorrect links tend to be grouped in high-quality and low-quality clusters respectively. Result accuracy can thus be enhanced by identifying and filtering out low-quality clusters. We describe our approach by investigating three open-source datasets, and further evaluate our work through a case study. The results show that our approach outperforms a baseline pruning strategy and that improvements are still possible.

4.1 Introduction

IR-based tracing tools favor recall over precision. This is mainly because commission errors (false positives) are easier to deal with than omission errors (false negatives) [121]. However, retrieving an excessive number of links can seriously affect the practicality of such tools. A tool that always returns all possible links is guaranteed to have a 100% recall, but is practically useless. Researchers have therefore focused on retrieving and ranking the

correct traceability links in the upper part of the result list, so that the human analyst can save effort by vetting only the top-ranked subset [46, 57]. The links with similarity scores above a certain threshold (cutoff) value are called candidate links [69, 121]. For example, a threshold of 0.3 was used in [37]; other approaches (e.g., [46, 57, 121]) have evaluated different cutoff values.

Determining threshold in practice can be challenging: Using a low threshold value retrieves a larger number of correct links than using a high value, but more incorrect links are captured at the same time [37]. Among the many performance enhancement techniques (e.g., [178, 252]), no approaches to date can largely decrease false positives at low cutoff points and significantly increase correct links at high cutoff points [37]. In this chapter, we aim to enhance IR-based candidate link generation by examining the cluster hypothesis [175], which states that relevant documents tend to be more similar to each other than to irrelevant documents. When adapted to traceability, the hypothesis suggests that correct and incorrect links can be grouped in high-quality and low-quality clusters respectively. Thus, the performance of IR-based tracing can be enhanced by selecting candidate links from high-quality clusters. It is our conjecture that discarding the links from low-quality clusters helps tackle the threshold determination problem.

Prior work in this area has employed clustering as an alternative to the ranked-list presentation to increase the understandability of the candidate links [69]; however, recall and precision are unchanged before and after clustering. In contrast, we leverage clustering to directly improve precision by reordering the retrieved traceability links. Some efforts have also combined clustering with other enhancement techniques [37, 46, 252]; however,

the clustering effect is often implicit in these integrated approaches. In contrast we perform thorough analysis of the underlying clustering algorithms, adopt novel metrics to explore the interplay of key clustering parameters, and conduct rigorous evaluations on both open-source and proprietary projects. The contributions of our work lie in the development of a set of detailed procedures to examine the cluster hypothesis in the context of IR-based requirements tracing. Our work not only advances the fundamental understanding about the role clustering plays in traceability, but also enables principled ways to increase the practicality of automated tracing tools. In what follows, we discuss how clustering has been employed in the related literature.

4.2 Clustering in Information Retrieval

Clustering is an unsupervised learning method which automatically divides data into natural groups based on similarity [132]. There has been extensive research on using clustering to improve IR systems. The basic tenet is known as the cluster hypothesis, stating that relevant documents tend to cluster near other relevant documents and farther away from irrelevant ones [175]. We discuss related work in IR from three perspectives: document, query, and search results. On the document side, clustering is mainly used to increase retrieval efficiency. As the number of documents increases, matching the query to all documents can degrade the system performance. A solution is to build a clustering of the entire collection and then match the query to the cluster centroids [257]. In this way, clustering is done in advance so as to enhance the performance at search time. This is referred to as static clustering as only one fixed clustering is made to accommodate all user queries. Hearst and

Pedersen [115], in designing the Scatter/Gather system, performed dynamic clustering, in which document clusters are formed dynamically depending on the user query. The evaluations demonstrated that Scatter/Gather consistently improved retrieval results and that users were able to distinguish high-quality clusters from low-quality ones [115].

On the query side, clustering is particularly useful for overcoming the well-known short query problem where the user input provides insufficient information. In this context, previously encountered queries are collected and placed into groups. For example, Yi and Maghoul [259] presented an algorithm to compute equivalent classes of queries (i.e., query clusters) and tested the algorithm on a Web search dataset consisting of over 16M unique queries. For a new incoming query, its equivalent class helps to expand the query with terms that reflect similar information needs [259]. In cases where the new query is not similar to any of the document cluster centroids, it may instead be similar to one of the query groups, which in turn can be used to match document clusters [257].

On the result side, clustering is widely applied to support users interactive browsing [115, 156, 261]. In a ranked list presentation, the search results are isolated from their context. Organizing and displaying the retrieved artifacts in topic-coherent clusters can facilitate the comprehension and evaluation of the search results. However, in order to effectively provide the contextual information (e.g., generating cluster labels and determining the ordering among the result clusters), human intervention is still required. In fact, a series of experiments showed that result clustering could be as effective as the interactive relevance feedback based on query expansion [156]. Clusterings primary benefit, then, arises from the sense of control it offers over the relevance feedback process [156].

4.3 Clustering in Traceability

Despite the broad role clustering plays in supporting IR systems, the application of clustering in traceability has emerged only recently. We discuss here first the seminal work by Duan and Cleland-Huang [69] on using clustering to improve the comprehension and evaluation of the retrieved traceability links, followed by the hybrid approaches [37, 46, 252] that synthesize clustering and other trace retrieval enhancement strategies.

The work in [69] is among the first to systematically investigate the application of clustering in traceability. Duan and Cleland-Huang [69] focused on the result-side clustering. The main goal was to increase understandability and reduce the human effort needed to evaluate a set of candidate links. They compared a set of representative clustering algorithms on three traceability datasets, proposed heuristics to determine optimal clustering granularity, and developed metrics to rank the trace clusters. The evaluation on tracing five business requirements showed that clustering traceability links led to fewer number of decision points than presenting links in an ordered list, thereby saving the effort needed to evaluate the candidate links [69].

While result-side clustering improves comprehensibility, it is the same set of candidate links that is retrieved by non-clustering techniques and the clustering-enabled method described in [69]. In other words, recall and precision are unaffected before and after clustering. In attempts to retrieve higher quality links than the baseline IR method, clustering has also been studied. For instance, Cleland-Huang et al. [46] introduced three enhancement strategies (hierarchical modeling, logical clustering of artifacts, and semi-automated pruning) to improve the probabilistic network model, Chen and Grundy [37] described

three supporting techniques (regular expression, key phrases, and clustering) to improve VSM, and Wang et al. [252] presented four strategies (source code clustering, identifier classifying, similarity thesaurus, and hierarchical structure enhancement) to improve LSI. Although the integrated strategies are promising, the studies showed mixed results, e.g., the approach in [252] resulted in higher precision but lower recall than LSI. However, perhaps the more important question unanswered in these studies is the enhancement effect resulted from clustering alone rather than from the combined approaches. Lack of such knowledge limits our understanding of clusterings strengths and weaknesses, which in turn impedes the selection of appropriate clustering techniques to be complemented with other types of strategies. Understanding the role clustering plays in enhancing candidate link generation requires rigorous empirical inquires. This is precisely the focus of our research.

4.4 Research Methodology

This section presents our research methodology including or central research hypothesis and associated research questions. For readability purposes, all the analysis figures are placed at the end of this chapter.

4.4.1 Central Hypothesis

The cluster hypothesis has been shown true on multiple occasions [37, 46, 69, 115, 156, 252, 257, 261]. When applied in requirements tracing, the cluster hypothesis suggests that correct links tend to be more similar to each other than to incorrect links. This motivates us to formulate our central hypothesis clustering algorithms and optimal clustering granular-

ity exist in grouping correct and incorrect links into high-quality and low-quality clusters respectively.

The overarching goal of our research, which is to capture all potential correct traceability links and few incorrect ones, can be accomplished by discovering the appropriate clustering mechanisms, distinguishing between high- and low-quality clusters, and filtering out the links in low-quality clusters.

Figure 4.1 shows our clustering-based enhancement approach in the context of a baseline tracing process. The baseline process is commonly adopted in state-of-the-art tracing tools like RETRO [121] and Poirot [47]. The human analyst selects some requirements artifact to trace, and the IR algorithm retrieves the traceability links by computing the similarity between the query and the software artifacts in the repository. Pruning aims to semiautomatically discard the portion of the retrieved results with a low density of correct links. If this portion were presented to the analyst, then the effort required to discard false positives would become much higher than the effort to validate correct links. For this reason, most approaches use some method to cut (or filter) the result list, thus presenting the human analyst only the subset of retrieved links (or candidate links). Typical pruning strategies include: 1) an absolute threshold on the similarity value, e.g., links whose similarity scores are greater than or equal to 0.3 are chosen as candidate links in [37]; and 2) a relative cutoff point for the proportion of retrieved results, e.g., 70% of the most similar links are selected in [169]. The clustering-based enhancement that we propose is highlighted using white boxes in Figure 4.1. We discuss the key aspects of each step as follows.

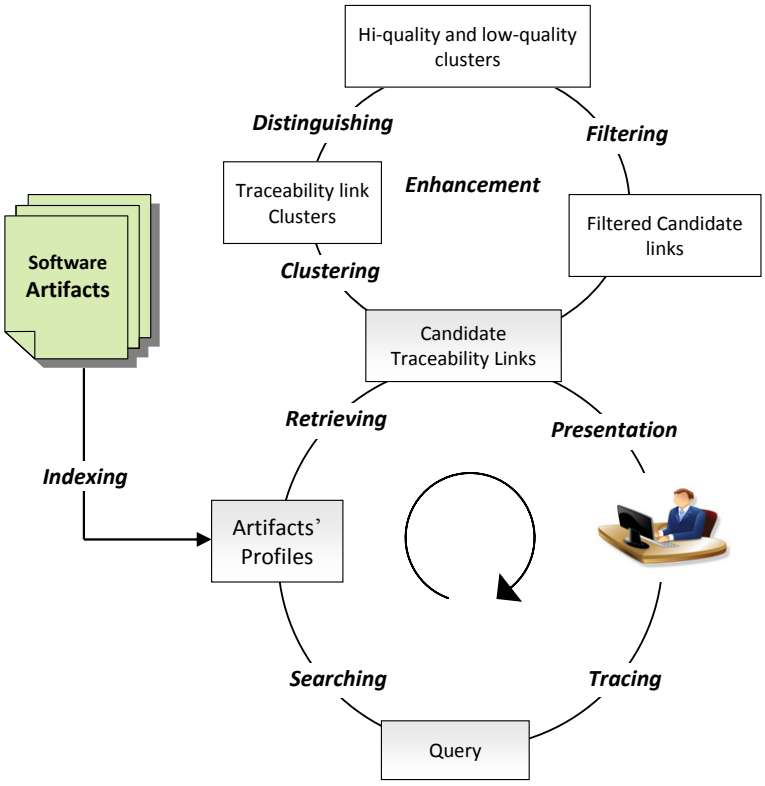


Figure 4.1

A Clustering-based Approach to Enhancing Link Generation for Requirements Tracing

4.4.1.1 Clustering

We perform clustering after the initial search is completed. This makes our clustering dynamic rather than static, i.e., we do not assume that if two artifacts L_1 and L_2 are both correct or incorrect links for requirement RA, they must both be correct or incorrect links for RB. The link clusters produced in our approach are query-dependent, and therefore have the potential to be closely tailored to the characteristics of the specific requirement being traced.

4.4.1.2 Distinguishing

Upon the identification of a clustering that divides correct and incorrect links into separate groups, it is crucial to research automated ways to differentiate between high- and low-quality clusters. We propose several heuristics and test their performance empirically.

4.4.1.3 Filtering

Filtering in our approach does not rely solely on a links similarity to the query, but takes into account the cluster the link belongs to as well. In other words, a links neighbors also define its relevance. Our filtering is thus performed on a cluster basis, which is fundamentally different from the baseline pruning strategy of acting on individual links according to their similarity scores or rankings.

4.4.2 Research Questions

The overall goal of our research is to assess the extent to which the cluster hypothesis can be leveraged to enhance automated traceability. We refine our general goal with a set of specific research questions (Figure 4.1).

- What is the clustering algorithm that best separates correct and incorrect links into different groups? What is the optimal granularity (e.g., the optimal number of resulting clusters) to do so?
- What is the most adequate heuristic in uncovering the quality of traceability link clusters?
- As the links in low-quality clusters are removed, what is the most appropriate way to arrange the remaining candidate links?

The answers to the research questions will enable the discovery of the underlying clustering mechanisms, along with the supporting strategies, for improving candidate link generation. Next, we present an experimental study that seeks to answer these research questions.

4.5 Experimental Investigation

In most cases, identifying the best settings for effectively integrating clustering in a working application is viewed as an NP-complete optimization problem [145]. For this reason, we are compelled to make some assumptions about the general clustering features in order to achieve an acceptable approximation. In particular, we restrict our discussion to mutually exclusive and collectively exhaustive clustering, which allows each of the retrieved traceability links to be assigned to one and only one cluster. Three datasets are used in our investigation including *iTrust*, *eTour*, and *CM-1*. We use these datasets to investigate

the 3 enhancement procedures in our approach. For each procedure, we state the experimental setting, describe the evaluation method, and present the result analysis. We discuss the threats to validity in Section 4.5.4.

4.5.1 Clustering Traceability Links

In one of the most systematic comparisons of traceability link clustering [69], three algorithms, namely Hierarchical Agglomerative Clustering (HAC), k-means, and bisecting divisive clustering (bisecting for short), are evaluated. These algorithms cover a wide spectrum of clustering methods available in the literature. They are also known to perform well in documents clustering [132]. We test a similar set of algorithms in our study, including k-means, bisecting, and 3 variants of HAC including:

$$SL(SingleLinkage) : M(A, B) = min\{d(a, b) : a \in A, b \in B\}. \tag{4.1}$$

$$CL(CompleteLinkage) : M(A, B) = max\{d(a, b) : a \in A, b \in B\}. \tag{4.2}$$

$$SL(AverageLinkage) : M(A, B) = \frac{1}{|A| \cdot |B|} \sum_{a \in A} \sum_{a \in A} d(a, b). \tag{4.3}$$

where $d(a, b)$ is the distance between data objects *a* and *b*, and $M(A, B)$ defines the linkage (merging) criteria for clusters *A* and *B*. For example, SL merges the two clusters with the smallest minimum pairwise distance. All the five clustering algorithms in our study rely on the terms contained in the software artifacts to compute similarity. For our experiments, the similarity scores are computed using TFIDF [175], a popular scheme in VSM which has been validated through numerous traceability studies [118, 121, 199]. Such a choice, at the same time, defines the baseline tracing mechanism (Figure 4.1) in our study to be TFIDF

in VSM. We use our indexer, described in Chapter 1, to pre-process both source code and (requirements & design) documents. Three steps are involved: tokenizing, filtering, and stemming [169].

When exploring the optimal clustering granularity, two interdependent parameters are usually calibrated: the number of clusters *k* and the cluster size $|C|$. In our case, the primary driving factor shared by all the five algorithms is *k*, which we use to gauge the clustering granularity. In terms of $|C|$, especially for k-means and bisecting, we adopt the rule of thumb to generate balanced clusters, i.e., relatively uniformly sized clusters [69].

4.5.1.1 Evaluation Method

In general, clustering quality can be evaluated either internally or externally [132]. Internal evaluation does not refer to external knowledge (e.g., an authoritative decomposition prepared by human experts) and therefore cannot assess the goodness of a clustering method. Rather, internal evaluation compares how close the clusterings are to each other. In [69], for instance, a pairwise correlation analysis was performed between the $\{HAC, k-means, bisecting\}$ algorithms. The result showed that at reasonable clustering granularity (of 5-6 clusters or higher) no significant difference was observed between the clusters produced by the three algorithms. Internal evaluation is particularly suited for exploratory studies where clustering is conducted to discover patterns in the input data [132].

When evaluated externally, clustering results are compared with a gold standard or classification labels [132]. These ground-truth clusters, also known as authoritative figures, are usually produced by experts or provided based on a natural decomposition of the data.

A clustering algorithm can then be evaluated based on how much of the known structure it can recover [132]. External evaluation fits our purpose since our objective is to assess the "goodness" of the clusterings, as well as to determine the optimal clustering granularity.

We devise a gold standard (not the only gold standard) in our study by directly applying the cluster hypothesis to the automated tracing problem. In an ideal situation, only two clusters should be present: one cluster has a 100% recall and precision (i.e., it contains only and all the correct traceability links), and the other cluster has a 0% recall and precision (i.e., it contains only and all the false positives). Figure 4.2-a illustrates this ideal situation.

In order to calibrate the clustering granularity k while externally evaluating the clustering results, we adopt the MoJo distance measure [254], the *de facto* metric embraced by the software clustering community. MoJo measures the distance between two decompositions of the same software system by computing the number of Move and Join operations to transform one to the other. Intuitively, the smaller the MoJo distance, the closer the two clustering results. Take Figure 4.2 as an example, the MoJo value of transforming Figure 4.2-b to Figure 4.2-a is 2: moving the correct link from **E** to **C** and then joining **D** and the revised **E** together. We integrate MoJos optimal implementation described in [254] (release 2.04) into our current analysis.

4.5.1.2 Result Analysis

Before discussing the results, it is important to comment on the intrinsic intractability of clustering with constraints [52] so as to better understand our analysis procedure shown next. The problem of separating n data points into k disjoint sets such that pairwise dis-

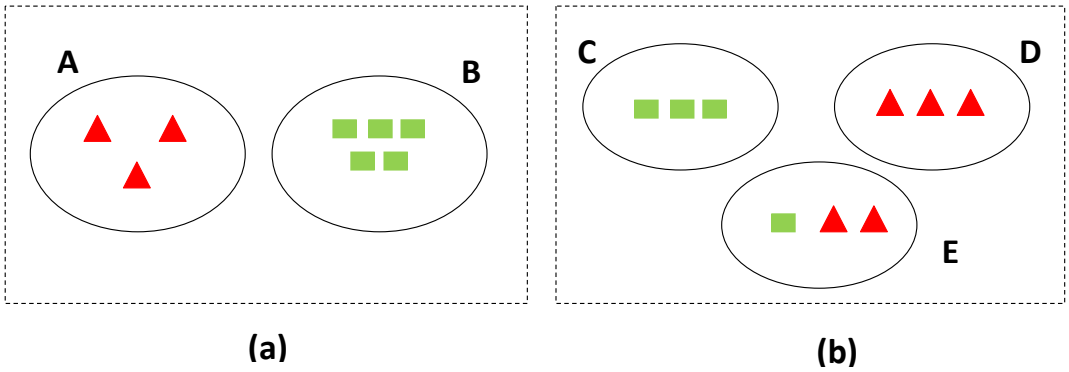


Figure 4.2

MoJo Analysis

tances within sets are bounded by a constant is known to be NP-complete [145]. Therefore, most formulations of the clustering problem are NP-hard, which means that there cannot be an efficient clustering algorithm that satisfies all constraints for all datasets [52]. However, acceptable approximations can be achieved by using procedures that optimize one or more fitness measures. The main goal is to choose from a large set of possible solutions the one that gives the best value for the selected measures. Our procedure for finding optimal clustering settings can be described as follows:

1. **For** each $c_i \in$ Clustering_Algorithms
2. **Loop** (initialize k=2) **AND** (increment k by 1)
3. **For** each $d_j \in$ Datasets
4. **For** each $q_m \in$ Trace_Query(d_j)
5. $RTL_m \leftarrow$ Retrieve_Traceability_Links(q_m)
6. $GS_m \leftarrow$ Produce_Gold_Standard(q_m)
7. $CR_{m,i,k} \leftarrow$ Cluster(RTL_m, c_i, k)
8. $MoJo_{m,i,k} \leftarrow$ Compute_MoJo($CR_{m,i,k,}$, GS_m)
9. $Average\_MoJo_{i,k} \leftarrow \frac{1}{m} \sum_m MoJo_{m,i,k}$
10. **Break** if *local_minimum*($Average\_MoJo_{i,m}$)
11. **Return** *min*($Average\_MoJo_{i,k}$)

The measure that we choose to optimize is the MoJo distance averaged over all the trace queries in a given dataset (lines 9-10). That is, if a local minimum of $Average\_MoJo_{i,k}$ is reached for the clustering algorithm ci at granularity level *k*, then the optimal clustering setting is found and $min(Average\_MoJo_{i,k})$ is returned. The procedure will repeat for the next evaluation cycle with a new clustering algorithm c_{i+1}. It is necessary to point out that *k* is looped from 2. This is because the gold standard that we use consists of 2 clusters (e.g., Figure 4.2-a). Thus, a local minimum near the lower bound of *k* is considered to be an acceptable solution to our particular optimization problem. Figure 4.3 plots the average MoJo values for all the three traceability datasets. For *iTrust*, Average MoJo hits a minimum when using SL to produce 7 clusters. For *eTour*, SL achieves the best performance at 7-8 clusters. For *CM-1*, a local minimum of Average MoJo is obtained when 9 clusters are generated by either CL or SL. From our analysis, SL turns out to be the best candidate for realizing the cluster hypothesis in traceability. The optimal clustering granularity of SL is found to be $k \in [7, 9]$ across the three datasets. These findings differ from the previous observations that *k*-means, bisecting, and HAC result in similar clusterings when *k* equals to 5-6 or higher [69]. We speculate this difference is attributed to the different evaluation methods employed: while we externally assess how close the clusterings are to a gold standard, the study in [69] internally evaluates how close the clusterings are to each other.

To further validate our procedure, we apply the optimal clustering settings (i.e., SL with *k*=8) to trace all the requirements from each dataset. For every trace query, the resulting 8 SL clusters are arranged in a descending order based on their recall values (the percentage

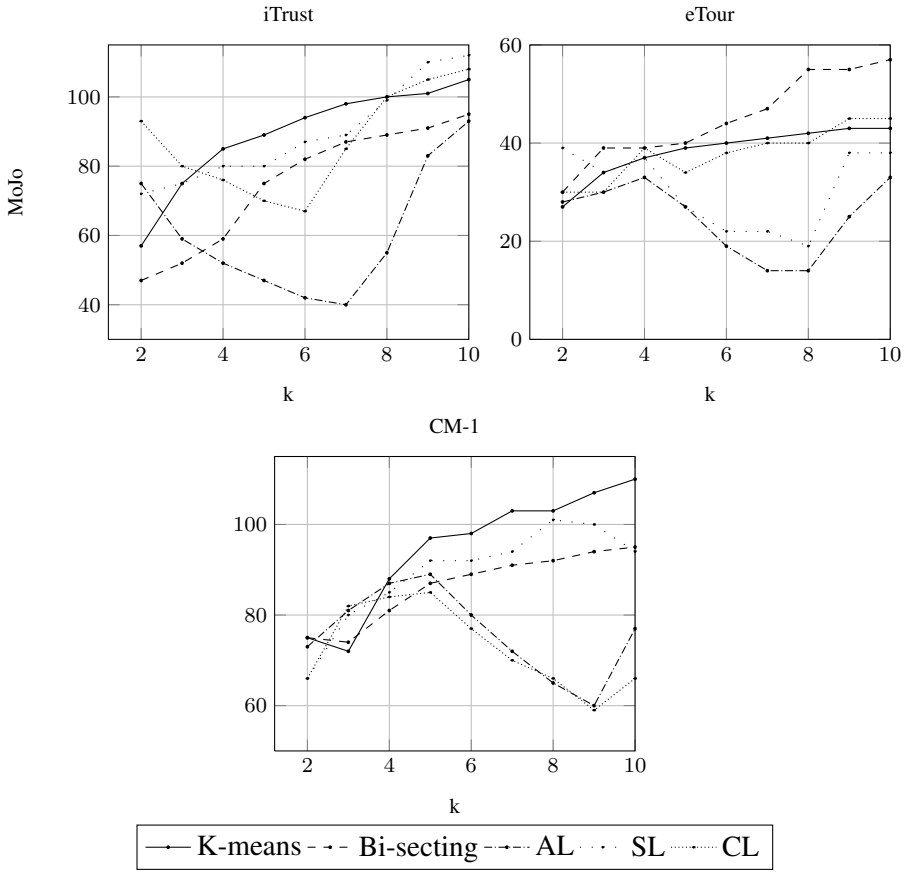


Figure 4.3

MoJo Analysis for Comparing the Clustering Algorithms

of correct links they contain). The recall values of all the clusters at the same rank (1st, 2nd, etc.) are then averaged over all the trace queries. Figure 4.4 shows the results. It is apparent that, when clustered using the optimal settings, the retrieved traceability links are effectively separated in high-quality (high-recall) and low-quality (low-recall) groups. For example, the top-3 clusters in CM-1 contain almost all the correct links. These results provide evidence supporting the findings previously reported in [115, 132], stating that there existed an optimal clustering mechanism such that if the IR system were able to optimize the clustering settings, the mechanism would always perform better than baseline document retrieval.

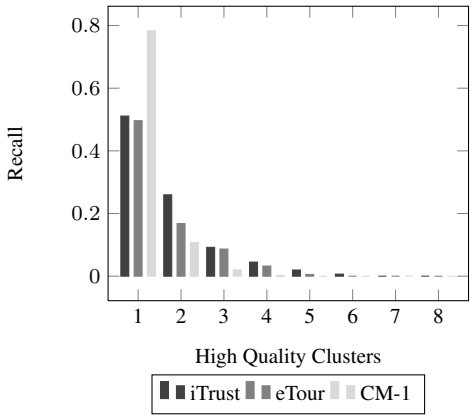


Figure 4.4

Traceability Link Clusters Arranged Based on Recall

4.5.2 Determining the Quality of Link Clusters

In Figure 4.4, the cluster quality is judged by using the answer set available for each dataset. However, under non-experimental settings where no answer set is available, a

new strategy for determining cluster quality is needed. Automated support for this determination is challenging, e.g., the quality of clusters were determined manually in [115]. We investigate three heuristics to distinguish between high- and low-quality clusters, given the cluster hypothesis holds. These heuristics are motivated by dynamic clustering [115], which emphasizes that the clusterings should be query-specific. Since traceability link clustering in our approach is dynamic, each cluster can be characterized via certain link based on the similarity to the trace query.

- MAX (maximum similarity): The link with the maximum TFIDF similarity to the trace query is selected as a representative for the cluster. Clusters are then arranged based on their MAX representatives.
- AVE (average similarity): Clusters are sorted based on the average TFIDF similarity of their links to the query.
- MED (median similarity): Clusters are sorted based on the median TFIDF similarity of the links to the query.

4.5.2.1 Evaluation Method

The link clusters quality hinges on their recall values since automated traceability strives to achieve the highest possible recall. Similar to the iterative procedure decribed earlier, we evaluate the heuristics { MAX, AVE, MED } by closely monitoring the drop rate in recall each time the links from a low-quality cluster are discarded. The heuristic that maintains a consistently high recall during the removal of lowest-quality clusters is then considered to be an effective strategy.

4.5.2.2 Result Analysis

Figure 4.5 compares how recall drops when different heuristics are applied. In all three datasets, MAX is the most successful strategy for determining the quality of link clusters.

In other words, it is adequate to use each clusters link that is the most similar to the query to distinguish high- and low-quality clusters. The results in Figure 4.5 also show that the number of lowest-quality clusters to remove varies from dataset to dataset. When the clusters are arranged using MAX, discarding the links from the bottom 3 clusters has little effect on recall in *iTrust* and *eTour*. *CM-1*, a considerable drop in recall is observed if the 6^{th} bottom-ranked cluster is removed. Although *CM-1*s MAX curve represents a best case scenario, we believe removing 3 lowest-quality clusters is a satisfactory answer to our research question. The links in these clusters are mostly false positives, so discarding them could significantly improve candidate link generation.

4.5.3 Generating Candidate Links

Two tests are performed at this stage. First, it is crucial to compare the candidate links generated via our clustering-based approach with those produced by the baseline pruning strategy. This test evaluates the enhancement effect (Figure 4.1). Second, two styles of presenting the newly generated candidate links are assessed:

- ABS (using absolute similarity scores): The candidate links are ordered based on the similarity to the trace query regardless of their clusters.
- MAX (preserving cluster boundaries): The candidate links are displayed in their respective clusters, which in turn are arranged by their MAX representatives. Inside each cluster, the links are ranked according to theTFIDF similarity to the trace query.

4.5.3.1 Evaluation Method

For the first test on enhancement effect, standard recall and precision metrics are used. For the second test, these primary metrics are inappropriate because the two styles are applied to display the same set of candidate links (i.e., the links contained in the 5 highest-

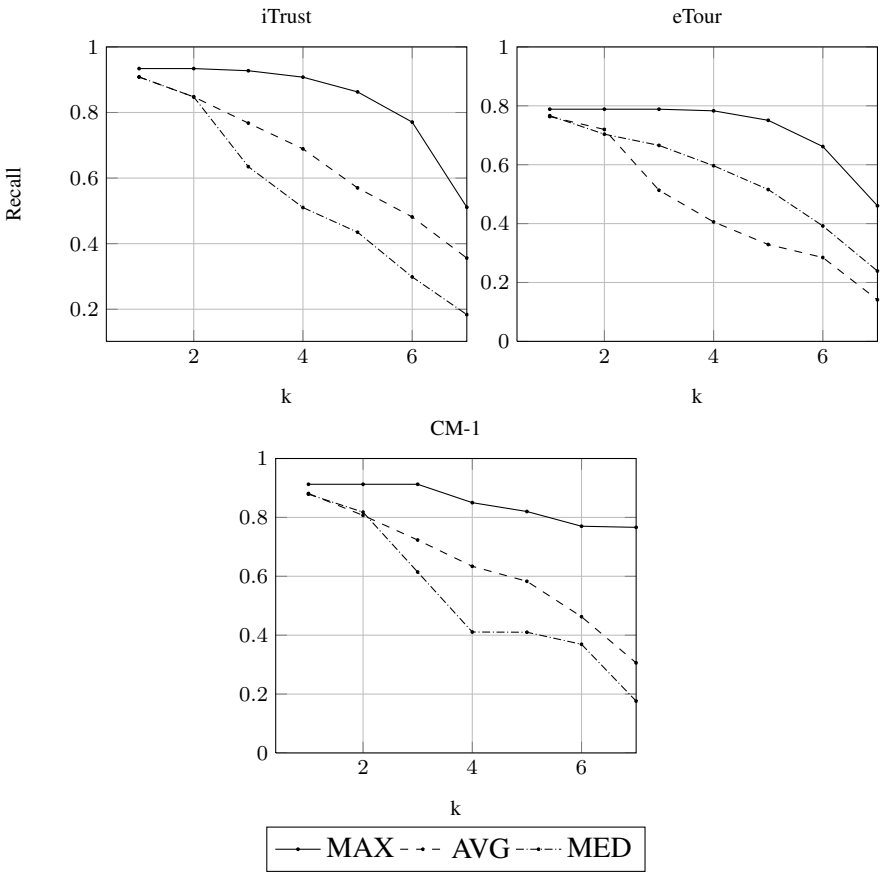


Figure 4.5

Using different representatives to determine the quality of clusters.

quality SL clusters judged by the MAX heuristic). We therefore resort to MAP (mean average precision), a secondary measure useful in evaluating the performance of tracing methods [237]. Intuitively, the higher the MAP, the closer the correct links are to the top of the candidate link presentation. The presentation with a higher MAP is superior as the links are easier to browse [237].

4.5.3.2 Result Analysis

Figure 4.6 and Figure 4.7 show the recall and precision comparisons respectively. Note that the baseline and our clustering-based enhancement use different units in filtering out false positives. In order to reconcile this difference, the thresholds in Figure 4.6 and Figure 4.7 are specified by following the baseline pruning strategy, namely, the top $(x \times 100)\%$ of the most similar links in the ranking defined by the links TFIDF scores, where $x = \{1.0, 0.9, 0.8, \cdots, 0.4\}$. To accommodate the analysis of cluster-based filtering proposed in our approach, the size of each cluster is also depicted through the dotted vertical cluster boundaries line in Figure 4.6 and Figure 4.7. The bold dotted line shows the cutoff point for selecting candidate links in our approach, i.e., discarding the links from the 3 lowest-quality clusters and keeping the remaining as the candidate links. The recall and precision comparisons clearly show that our approach outperforms the baseline. To examine whether the difference is statistically significant, we use Mann-Whitney test ($\alpha = .05$). Mann-Whitney is a non-parametric test. It does not make any assumption about the distribution of the data [48]. The test results show that, in all three datasets, our clustering-based approach performs significantly better than

the baseline pruning method, with $< recall, precision >$ p-values of $< 0.043, 0.018 >$, $< 0.036, 0.046 >$, and $< 0.018, 0.034 >$ for iTrust, eTour, and CM-1 respectively. This leads us to conclude that our approach significantly enhances candidate link generation for automated requirements traceability.

Figure 4.8 shows the MAP values when the two presentation styles, ABS and MAX, are compared. The results imply that ABS slightly outperforms MAX in all three datasets; however, the improvement is significant only in eTour (p=0.046). This finding raises some interesting issues in how to best present the candidate links to the human analyst. While our results seem to suggest that the traditional ranked-list display (ABS) contains more correct links on the top, preserving cluster boundaries (MAX) has shown to be valuable in improving the understandability and usability of candidate traceability links [69]. We are currently carrying out pilot studies with our TraCter tool [170] to further investigate the presentation factor and its impact on assisted requirements tracing [62].

The MAP analysis completes our experimental inquiry of seeking answers to the set of our research questions. The results can be summarized as follows.

- The cluster hypothesis holds in traceability.
- Single-link (SL), at the k=8 clustering granularity, represents a good candidate mechanism for fulfilling the potential suggested by the cluster hypothesis.
- The quality of clusters can be adequately inferred by their maximum similarity (MAX) to the trace query, and the 3 lowest-quality clusters contain such a high density of false positives that discarding them significantly improves the overall quality of the candidate link generation.
- Displaying the candidate links in an absolute similarity (ABS) manner facilitates the browsability aspect of the automated tracing results.

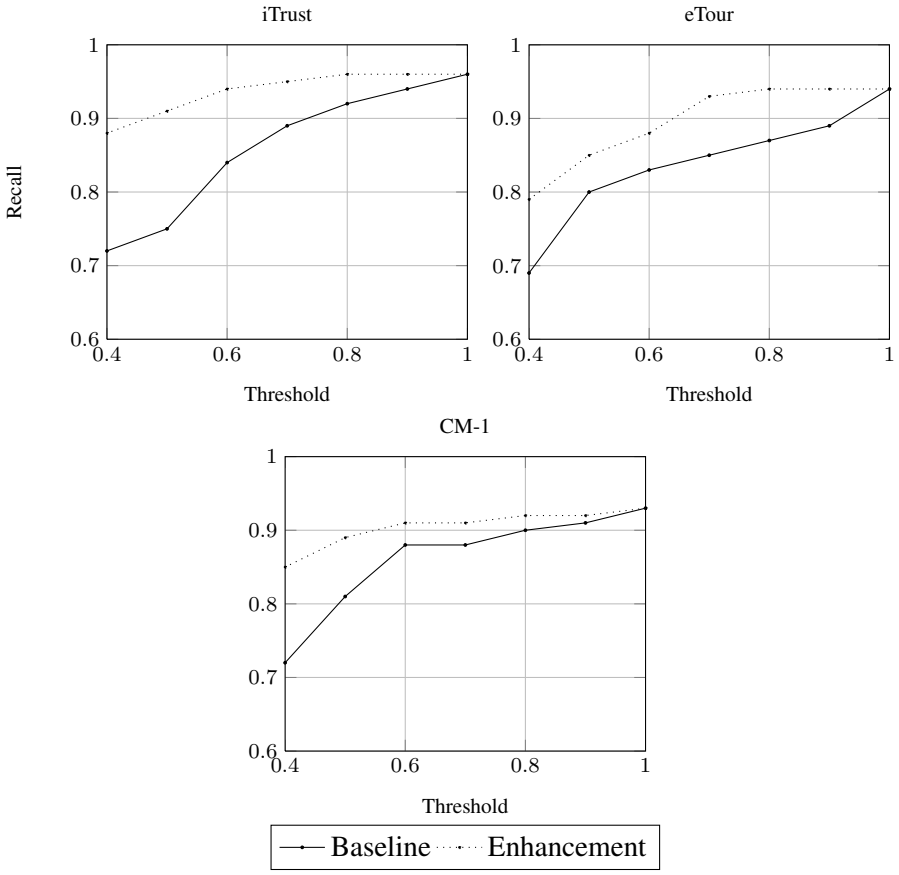


Figure 4.6

Comparing Recall

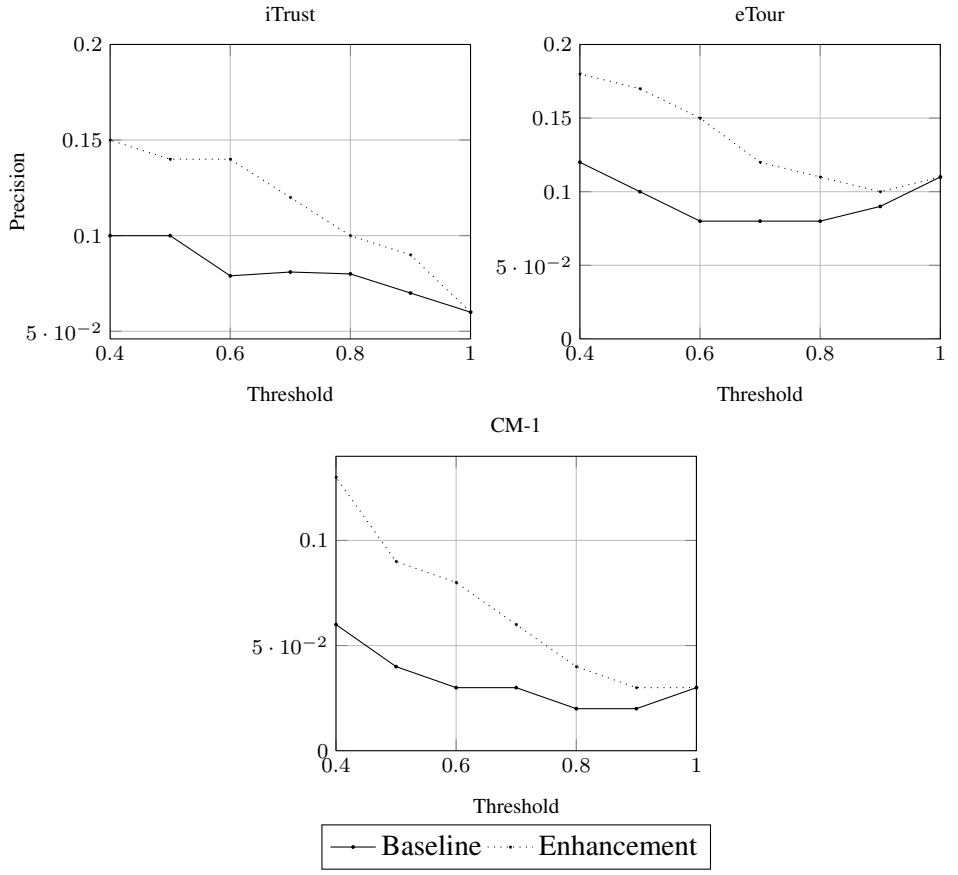


Figure 4.7

Comparing Precision

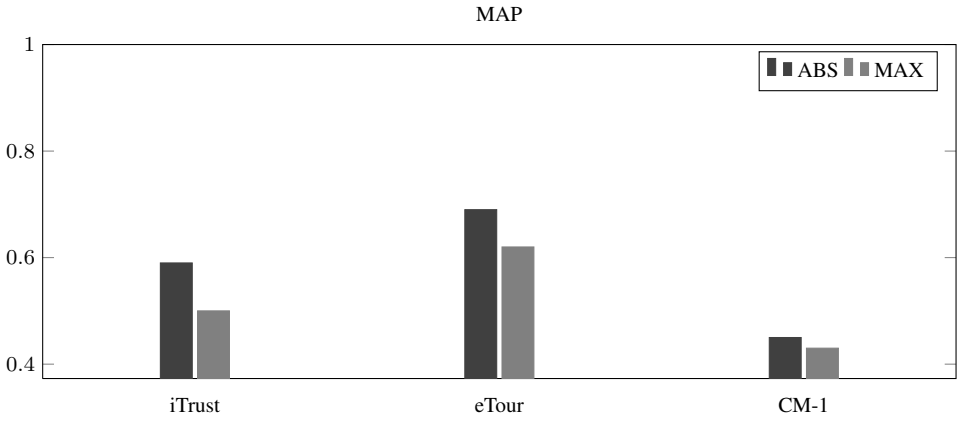


Figure 4.8

Assessing the Browsability of Different Presentation Styles

4.5.4 Threats to Validity

As is the case for most controlled experiments, our investigation into the cluster hypothesis in traceability is performed in a restricted and synthetic context. We discuss here some of the most important factors that must be considered when interpreting the results.

The construct validity [260] of our study can be affected by the use of a gold standard to operationally measure the *goodness* of clustering results. The gold standard that we devise assumes that an optimal decomposition of the traceability link space should have only two clusters, thus effectively splitting the correct links and incorrect ones into perfect-quality and zero-quality groups respectively. This assumption is based on an extreme leverage of the cluster hypothesis to the automated tracing problem. At the other extreme, every retrieved link can be its own cluster or only a couple of correct links can form a cluster which is separated from other links; however, such decompositions offer little value to automated tracing solutions. Our operationalization of *goodness* represents only one of the gauges to search for acceptable approximations to the clustering with constraints problem, but a practically useful one. In cases that other clustering solutions are desired (e.g., the cluster size should be adjusted to $7 - 2$ in order to take human usability into consideration [69]), we argue that the procedure in Section 4.4.1 may be applied incrementally to help find other optimal clustering settings. One of the internal validity [260] threats relates to the sequential examination of the three enhancement steps in our approach (Figure 4.1). The test of the cluster-quality determination heuristics, for example, is executed by applying the optimal clustering mechanisms discovered in the previous step. While we feel that our

execution configurations are logical, caution must be taken in interpreting our findings as a whole rather than separately.

In terms of external validity, two of the projects are developed by students and are not necessarily representative of all systems. In particular, proprietary software products are likely to exhibit different characteristics. We also note that our chosen traceability datasets are of medium size, which may raise some scalability concerns. Nevertheless, we believe the use of three datasets from different domains while incorporating both requirements-to-source-code and requirements-to-design traces helps mitigate related threats. We present an industrial case study in the next section in order to triangulate our findings. Other threats might stem from specific design decisions, such as the preprocessing indexer used, the TFIDF similarity measure computed, and so on.

Finally, in terms of reliability [260], we conduct all the experiments on open-source projects using procedures, algorithms, and measures completely described in this chapter. Moreover, our implementations are available upon request. We therefore believe it is possible to independently replicate our results.

4.6 Evaluation Study

The experimental study presented in this chapter uses open-source projects to answer our research questions in a quantitative and precise way. To further investigate the benefits of our approach, we conduct an exploratory case study [260] by applying the results to WDS, our industrial proprietary software system deployed in the workforce development

domain. The objective is to assess the usefulness and the scope of applicability of our approach, and more importantly, to identify areas for improvement.

4.6.1 Background

The development team of WDS describes its requirements practice as a goal-oriented and agile process. While the high level business goals shall be fulfilled, the changes to low level design and implementation are also embraced. Currently, WDS employs a commercial state-of-the practice issue tracking system to manage the traceability information. Although there is no immediate need to invest in new techniques, the WDS team is very interested in exploring how automated tracing methods like ours can help improve their practice, especially for handling volatile requirements.

4.6.2 Results

Table 4.1 shows the results obtained by applying our approach to the 6 WDS requirements. For each requirement, the table compares the performance of three tracing methods: term-based retrieval, baseline pruning, and our clustering based enhancement. Note that these 6 requirements are processed by using the same procedures, algorithms, and configurations as described earlier. It is encouraging to realize that our approach greatly outperforms the baseline in generating the candidate traceability links for three requirements: Req3, Req4, and Req6. Not only is recall maintained at a high level (93%) by using our approach, but precision is markedly increased. This provides further evidence confirming the validity of the cluster hypothesis in traceability. Meanwhile, the result increases our confidence in the generalizability of our optimal clustering findings. For Req1, even the

initial IR-based tracing results in a relatively low recall (84%). Although the clustering-based enhancement completely matches the recall level, this situation does exploit one of the limitations of our approach. Being query-specific, our approach does not provide much support on the recall side. Requirements like Req1 contributes to the *hard-to-retrieve traces* problem [94], which needs solutions beyond the basic $term - matching$ mechanism. For example, replacing the original query with a new set of query terms is proposed in [94]. Even though this shows that clustering on the query side can be useful, the scope of our approach is currently limited on the retrieval and filtering side.

Among the WDS requirements under study, Req2 has the greatest number of correct traceability links, meaning that its traces are spread all over the link space. It is therefore difficult for any pruning or filtering mechanisms not to throw out some of the correct links. However, this does illuminate the value of our choosing a 2-cluster extreme (Figure 4.2-a) as a gold standard. In practice, when Req2 is traced, a successful dividing of the traceability links into perfect-quality and zero-quality clusters can be of great help. A closer look at Req2 reveals that it is a non-functional requirement (NFR) describing security-related coerns which are addressed by a large number of artifacts in WDS's code base. Tracing NFRs has received growing research attention in recent years [42]. It is therefore interesting to exploit the special nature of NFRs to further improve our approach. As far as Req5 is concerned, clustering has no effect

As far as Req5 is concerned, clustering has no effect on candidate link generation. This is clearly an exception to the general trend in Table 4.1, which shows that our approach improves both recall and precision over the baseline. It turns out that the two Java classes that

Table 4.1

Results of Applying our Approach to Business Requirements

		Recall			Precision		
Requirements	Correct links	Retrieved	Baseline	Enhancement	Retrieved	Baseline	Enhancement
Req1	25	0.84	0.68	0.84	0.06	0.08	0.10
Req2	330	0.92	0.80	0.85	0.77	0.84	0.89
Req3	14	0.93	0.50	0.93	0.08	0.07	0.13
Req4	22	0.00	0.73	0.95	0.06	0.08	0.11
Req5	17	0.94	0.82	0.82	0.11	0.20	0.20
Req6	20	0.95	0.80	0.95	0.08	0.10	0.12

are incorrectly filtered out implement general utility functions DateUtil.java and EmailUtil.java. In the software clustering literature, these are known as omnipresent objects [255] and need to be handled separately from regular data objects. Thus, a potential improvement to our approach is to take into account other types of information, such as structural [255] and runtime [75] information, to produce more reliable and robust clusterings.

4.7 Conclusions

In this chapter, we have proposed an approach to improving the performance of IR-based automated tracing by examining the cluster hypothesis. The approach is presented through a set of detailed procedures. Three open-source datasets from different application domains are investigated to discover optimal settings for these procedures. We further evaluate our approach through a case study that helps identify the limitations of our approach and the avenues for future research. The study results show that our approach outperforms the baseline, but still has more room for improvement. It is imperative to mention that as long as the challenge of achieving a 100% recall and precision is still standing, the

problem of automated tracing remains unsolved. More research is required on different aspects of the problem to achieve the desired quality levels for automated tracing tools to be successfully deployed in industrial settings.

CHAPTER 5

REFACTORING SUPPORT

In this chapter, we hypothesize that the distorted traceability tracks of a software system can be systematically re-established through refactoring, a set of behavior-preserving transformations for keeping the system quality under control during evolution. To test our hypothesis, we conduct an experimental analysis using three requirements-to-code datasets from various application domains. Our objective is to assess the impact of various refactoring methods on the performance of IR-based automated tracing tools.

5.1 Introduction

As projects evolve, new and inconsistent terminology gradually finds its way into the system's taxonomy [153], causing topically related system artifacts to exhibit a large degree of variance in their lexical contents [9, 81]. This phenomena is known as the *vocabulary mismatch* problem and is regarded as one of the principal causes of poor accuracy in retrieval engines [56].

A suggested solution for the vocabulary mismatch problem is to systematically recover the decaying vocabulary structure of the system through refactoring. Refactoring refers to a set of behavior-preserving transformations that improve the quality of a software system without changing its external behavior [200]. These transformations act on the internal

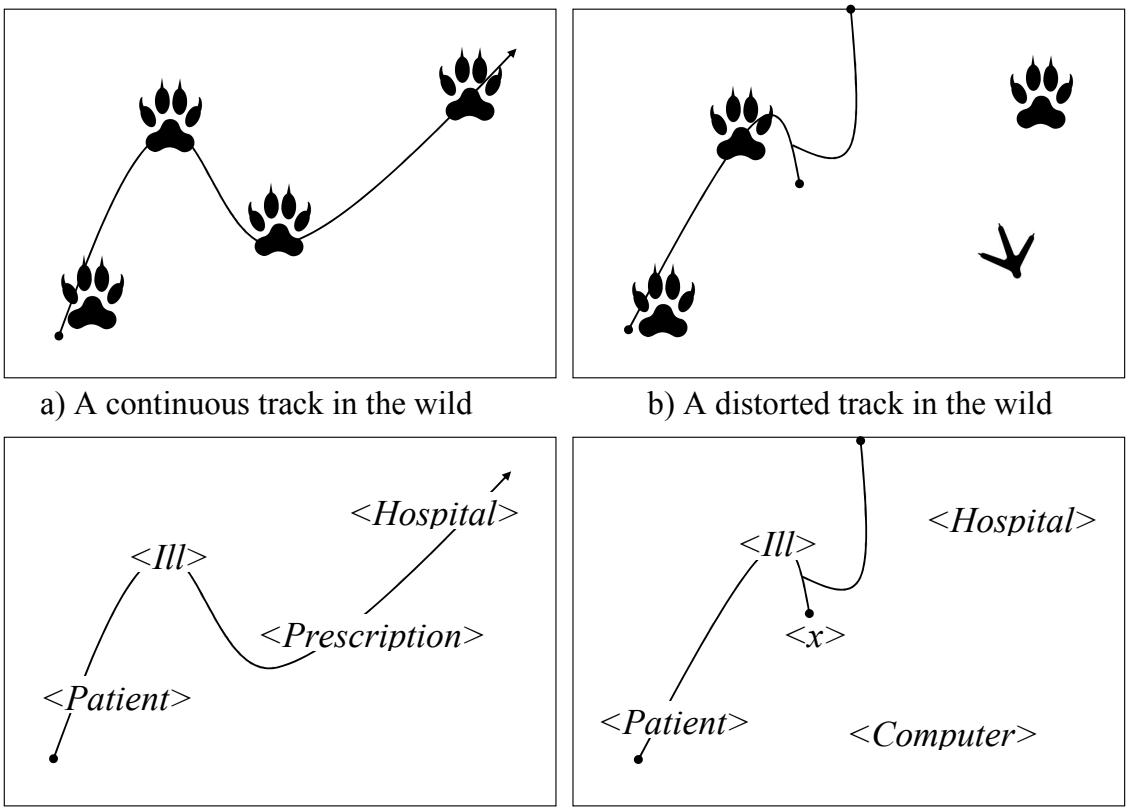
structure of software artifacts, including the nonformal and organizational features in the system, leaving the system's functionality intact [86]. Refactoring is now being advocated as an essential step in software development. For example, in agile methods, refactoring has already been integrated as a regular practice in the software's life cycle [184]. In addition, refactoring tools, which support a large variety of programming languages, have been integrated into most popular integrated development environments (IDEs), targeting various quality aspects of software systems (e.g., increase maintainability, reusability, and understandability) [29, 86, 131, 137, 171, 186, 187]. Motivated by these observations, in this chapter, we hypothesize that certain refactoring methods will help to re-establish the system's vocabulary structure that often gets corrupted during evolution [153], thus improving the retrieval capabilities of IR methods operating on that structure.

Refactoring can take different forms affecting different types of artifacts. Therefore, testing our research hypothesis entails addressing several sub-research questions such as: What refactoring methods improve trace retrieval quality? What refactoring methods have more influence on the system's traceability? How to evaluate such influence? How does refactoring compare to other performance enhancement strategies in automated tracing? And how to reverse any potential negative impact certain refactoring methods might have on traceability? To answer these questions, we conduct an experimental analysis using three datasets from various application domains. Our main objective is to explore systematic ways for enhancing the performance of IR-based automated tracing tools.

5.2 IR-Based Automated Tracing

To understand the mechanism of IR-based automated tracing tools, we refer to the main theory underlying IR-based trace link retrieval. In their vision paper, Gotel and Morris [99] established an analogy between animal tracking in the wild and requirements tracing in software systems. This analogy is based on reformulating the concepts of *sign*, *track* and *trace*. A sign in the wild is a physical impact of some kind left by the animal in its surroundings, e.g., a footprint. Figure 5.1-a shows a continuous track of footprints left by a certain mammal. The task of the hunter is to trace animals' tracks by following these signs. In other words, to trace means basically to follow a track made up of a continuous line of signs. Similarly, in requirements tracing, a sign could be a term related to a certain domain concept, left by a software developer or a system engineer in a certain artifact. Figure 5.1-c shows a continuous track of related words from the health care domain *<Patient, Ill, Prescription, Hospital>*. The task of IR methods is to trace these terms to establish tracks in system. These continuous tracks are known as *links*.

The availability of uniquely identifying marks, or signs, is vital for the success of the tracing process. However, just as in the wild, tracks in software systems can get discontinued or distorted due to several practices related to software evolution [76, 153]. In what follows, we identify three symptoms related to code decay that might lead to such a problem. These symptoms include:



a) A continuous track in the wild

b) A distorted track in the wild

c) A continuous track in the system

d) A distorted track in the system

Figure 5.1

Illustration of Sign Tracking.

5.2.1 Missing Signs

A track can get discontinued when a concept-related term in a certain artifact is lost. Figure 5.1-d shows how the trace link becomes discontinued when the word <*Prescription*> is changed to <*x*>. This can be equivalent to a footprint being washed off by rain in the wild (Figure 5.1-b).

5.2.2 Misplaced Signs

A track can also be distorted by a misplaced sign. For example, the word <*Computer*>, which supposedly belongs to another track, is positioned in the track of Figure 5.1-d. In the wild this is equivalent to a footprint implanted by another animal on the track of unique footprints left by the animal being traced (e.g., Figure 5.1-b shows a bird's footprint left on the mammal's track in Figure 5.1-a).

5.2.3 Duplicated Signs

This phenomenon is caused by the fact that some identical or similar code fragments are replicated across the code. These fragments are known as code clones [31]. In our example, this can be equivalent to a track branching into some other module that contains a word similar to one of the signs of the trace link identified in Figure 5.1-c. Some animals adopt this strategy in the wild to confuse their predators by duplicating their footprints in different directions at different periods of time.

5.3 Refactoring

Refactoring was initially introduced by Opdyke and Johnson as a systematic means for aiding evolution and reuse in legacy software systems [201]. While it can be applied to various types of artifacts, such as design and requirements, refactoring is mostly known for affecting source code [184]. Program refactoring starts by identifying *bad smells* in source code. Bad smells are *"structures in the code that suggest the possibility of refactoring"* [86]. Once refactoring has been applied, special metrics can be used to determine the effect of changes on the quality attributes of the system, such as maintainability and understandability [244].

Refactoring can be manual, semi, or fully automated. Manual refactoring requires software engineers to synthesize and analyze code, identify inappropriate or undesirable features (code smells), suggest proper refactorings for these issues, and perform potentially complex transformations on a large number of entities manually. Due to the high effort associated with such a process, the manual approach is often described as repetitive, time-consuming, and error-prone [182]. The semi-automated approach is what most contemporary IDEs implement. Under this approach, refactoring activities are initiated by the developer. The automated support helps to carry out the refactoring process, such as locating entities for refactoring and reviewing refactored results. In contrast, the fully automated approach tries to initiate refactoring by automatically identifying bad smells in source code and carrying out necessary transformations automatically. However, even in fully automated tools, the final decision whether to accept or reject the outcome of the refactoring process is left to the human [131].

Deciding on which particular refactoring to apply to a certain code smell can be a challenge. In fact, applying arbitrary transformations to a program is more likely to corrupt the design rather than improving it [200]. However, there is no agreement on what transformations are most beneficial and when they are best applied. In general, such decisions should stem from the context of use, such as the characteristics of the problem, the cost-benefit analysis, or the goal of refactoring (e.g., improving robustness, extensibility, reusability, understandability, or performance of the system) [182, 184]. In automated tracing, the main goal of adopting refactoring is to improve the system's vocabulary structure in such a way that helps IR-based tracing methods to recover more accurate lists of candidate links. Based on that, we define the following requirements for integrating refactoring in the IR-based automated tracing process:

- Altering nonformal information of the system: As mentioned earlier, IR-based tracing methods exploit nonformal information embedded in the textual content of software artifacts [8]. Therefore, for any refactoring to have an impact on IR-based automated tracing methods, it should directly affect the system's textual content.
- Coverage: Traceability links are often spread all over the system, linking a large number of the system's artifacts through various types of traceability relations [231]. Therefore, statistically speaking, to have a noticeable impact on the performance, adopted refactorings shall affect as many software entities as possible.
- Automation: Since the main goal of automated tracing tools is to reduce the manual effort, any integrated refactoring should allow automation to a large extent. For any refactoring process to be considered effort-effective, it should provide automated solutions for code smell detection and applying code changes [85]. Automating these two steps will help to alleviate a large portion of effort usually associated with manual refactoring.
- Granularity level: In all of our experimental datasets, traceability links are established at class granularity level (i.e., requirements-to-class) [121]. This limits our analysis in this chapter to refactorings that work within the class scope (e.g., MOVE METHOD and EXTRACT METHOD), rather than refactorings that affect the class structure of the system (e.g., REMOVE CLASS or EXTRACT CLASS). Enforcing this requirement ensures that our gold-standard remains unchanged after applying various refactorings.

Based on these requirements, we identify three categories of refactoring that might have an impact on the performance of IR-based tracing methods. These categories include: refactorings that restore, remove, and move textual information in the system, represented by RENAME IDENTIFIER, EXTRACT METHOD, and MOVE METHOD refactorings respectively. These particular refactoring methods have been reported to be among the most understood and commonly used refactorings in practice [3, 66, 187, 190]. In addition, the research on the automation of these particular refactorings have noticeably excelled in the past few years, producing a wide selection of tools that support a large number of programming environments in a scalable manner [131, 189, 219, 245]. Therefore, we select these particular refactorings as a target of our investigation in this chapter. In what follows we describe each of these refactorings in greater detail.

5.3.1 Restoring Information

Refactoring methods under this category target the degrading vocabulary structure of source code [9, 153]. The main goal is to restore the domain knowledge that often gets lost over iterations of system evolution. In general, any refactoring that results in adding new words to the set of the system's vocabulary can be classified under this category. For example, refactorings such as ADD PARAMETER or INTRODUCE EXPLAINING VARIABLE introduce new variables or parameters, thus potentially new domain-related knowledge. However, the most popular refactoring in this category is RENAME IDENTIFIER (RI) [3,66, 187,190]. As the name implies, this transformation refers to simply renaming an identifier (e.g., a variable, class, structure, method, or field) to give it a more relevant name [86].

RENAME IDENTIFIER is expected to target the *Missing Sign* problem affecting traceability methods.

As mentioned earlier, to be considered in our analysis, refactoring methods should provide support for automatic detection of code smells they target. In our analysis, we refer to the literature of source code abbreviations and acronyms expansion to identify procedures for capturing opportunities for RENAME IDENTIFIER refactoring [35,150,242]. In particular we apply the following procedure:

1. Identifiers are first divided into their constituent parts for analysis [150].
2. Identifiers with less than 4-character length. These are usually acronyms or abbreviations. In that case, the long form is used. For example, the parameter *HCP* in our health care system is expanded to *HealthCarePersonnel*. If the identifier is less than 4 characters but it is not an acronym nor an abbreviation, then it is renamed based on the context.
3. Identifiers which have a special word as part of their names. For example, the variable *PnString* is expanded to *PatientNameString*.
4. Identifiers with generic names. For example, in our health care system, the method's name *import* is expanded to *importPatientRecords*.

The main objective of this procedure is to achieve consistency. During multiple iterations of software evolution, slightly different abbreviations might be used to refer to the same domain concept, causing a mismatch between the vocabulary used in source code and that used in other software artifacts [152, 153]. This phenomenon is often described as a very common problem in software maintenance [61, 144, 150]. The proposed procedure for renaming identifiers tries to eliminate this inconsistency in the system by using one consistent form, whether an abbreviation or an extended form, to refer to the same domain concept. In our analysis, we use the extended full-word form. Our decision is based on

the converging evidence from related literature which indicates that, in the long run, abbreviations impact comprehension negatively [105, 239]. In contrast, full-word identifiers often lead to the best comprehension [51, 152]. In addition, using the long form keeps identifiers' names in sync with their functionality, which results in an overall improvement in code quality, and thus, the accuracy of IR methods working with these identifiers.

We implement our procedure to find candidates for renaming in our datasets. Once the candidate identifiers for renaming have been identified, the refactoring tool available in ECLIPSE 4.2.1 IDE is used to carry out the renaming process. This will ensure that all corresponding references in the code are updated automatically. Finally, the code is compiled to make sure no bugs were introduced during the process. It is important to point out that at the current stage of the research, choosing new identifiers' names is still a manual task, carried out by our researchers, using keywords available in the system's documentation, based on their understanding of the system's application domain, and the particular functionality of the identifier being renamed.

Table 5.1

Refactoring Methods Used in our Analysis

Refactoring	**Code Smell**	**Tracing Problem**	**Tool Support**	**Manual Effort**
RENAME IDENTIFIER	Decaying vocabulary	Missing signs	ECLIPSE	Verifying candidates for renaming Selecting identifiers' names
MOVE METHOD	Feature envy	Misplaced sign	Jdeodorant ECLIPSE	Verifying move-method candidates Verifying the results
EXTRACT METHOD	Code clones	Duplicated signs	SDD ECLIPSE	Verifying code clone candidates Selecting extracted method's name Selecting host class Verifying the results

5.3.2 Moving Information

This category of refactoring methods is concerned with moving code entities between system modules. The goal is to reduce coupling and increase cohesion in the system, which is a desired quality attribute of Object-Oriented design [93]. Refactorings under this category provide a remedy against the *Feature Envy* code smell. An entity has *Feature Envy* when it uses, or being used by, the features of a class other than its own (different from where it is declared). This may indicate that the entity is misplaced [86].

In our experiment, we adopt MOVE METHOD (MM) refactoring as a representative of this category. By moving entities to their correct place, this particular refactoring is expected to target the *Misplaced Sign* problem mentioned earlier. To identify potentially misplaced entities, we adopt the strategy proposed by Tsantalis and Chatzigeorgiou [245], in which they introduced a novel entity placement metric to quantify how well entities have been placed in code. This semi-automatic strategy starts by identifying the set of the entities each method accesses (parameters or other methods). *Feature Envy* code smell instances are then detected by measuring the strength of coupling that a method has to methods belonging to all foreign classes. The method is then moved to the target foreign class in such a way that ensures that the behavior of the code will be preserved. This procedure has been implemented as an ECLIPSE plug-in (*Jdeodorant* [1]) that identifies *Feature Envy* instances and allows the user to apply the refactorings that resolve them. However, despite of the high degree of automation, this process can still be regarded as semi-automatic [245]. In particular, verifying or rejecting the MOVE METHOD candidates suggested by the tool,

[1] http://www.jdeodorant.org/

and making sure that moving a method does not introduce any bugs in the system, are still manual tasks. In our analysis, we only consider misplaced methods. *Move Attribute* refactoring is excluded based on the assumption that attributes have stronger conceptual binding to the classes in which they are initially defined, thus they are less likely than methods to be misplaced [245].

5.3.3 Removing Information

These refactorings remove redundant or unnecessary code in the system. A popular code smell such refactorings often handle is *Duplicated Code*. This code smell is usually produced by Copy-and-Paste programming [134], and indicates that the same code structure appears in more than one place. These duplicated structures are known as code clones and are regarded as one of the main factors for complicating code maintenance tasks [181]. Exact duplicated code structures can be detected by comparing text [86]. However, other duplicates, where entities have been renamed or the code is only functionally identical, need more sophisticated techniques that work on the code semantics rather than its lexical structure [68].

The most frequent way to handle code duplicates is EXTRACT METHOD (XM) refactoring [176, 256]. In particular, for each of the duplicated blocks of code, a method is created for that code, and then all the duplicates are replaced with calls to the newly extracted method. When the duplicates are scattered in multiple classes, the new extracted method is assigned to the class that calls it the most. By removing potentially ambiguous duplicates, EXTRACT METHOD is expected to target the *Duplicated Sign* problem of

software artifacts. In our analysis, we use the duplicated code detection (SDD) [2] ECLIPSE plug-in to detect code clones. EXTRACT METHOD refactoring available in the ECLIPSE 4.2.1 IDE is then used to refactor candidate clones. In particular, the user selects the code fragment to be extracted from the list of candidate clones returned by the tool, and ECLIPSE will ask for a method name and a class to host the newly extracted method. A method name is selected based on the context of the code. Once the method is created, the user is responsible for replacing all clone instances with a call to the new method and making sure that no bugs are introduced by this process.

Table 5.1 summarizes the categories of refactoring methods introduced in this section, including the code smells they target, traceability problems they impact, the tool support available, and a summary of the manual effort required to carry out each refactoring.

5.4 Methodology and Research Hypothesis

Three datasets were used to conduct the experiment in this chapter including: *iTrust*, *eTour*, and *WDS*. Our experimental procedure can be described as a multi-step process as follows:

5.4.1 Refactoring

Initially the system is refactored using various refactoring methods mentioned earlier. The goal is to improve the system lexical structure before tracing. The results of applying different refactorings over our three experimental datasets. The table shows the number of entities affected by refactoring in each dataset (e.g number of moved or extracted methods

[2] http://wiki.eclipse.org/Duplicated_code_detection_tool_(SDD)

and number of renamed identifiers), the number of affected classes in each system, and the number of affected classes in the gold-standard, or classes that are part of a trace link in our answersets (C').

5.4.2 Retrieval

IR methods are used to identify a set of traceability links by matching the traceability query's profile with the artifacts' profiles in the software repository. In our experiment, we use Vector Space Model with TFIDF weights as our experimental baseline. TFIDF is a popular term-weight scheme in VSM which has been validated through numerous traceability studies as an experimental baseline (e.g., [121, 172]).

5.4.3 Evaluation

At this step, different evaluation measures (Sec. 1.5), are used to assess the different aspects of the performance. Performance of each dataset after applying a certain refactoring, in comparison to the baseline (VSM), is presented as a precision/recall curve over various threshold levels ($< .1, .2, ..., 1 >$) [121]. Wilcoxon Signed Ranks test is used to measure the statistical significance of the results. We use $\alpha = 0.05$ to test the significance of the results. Note that different refactorings are applied independently, so there is no interaction effect between them.

5.5 Results and Discussion

This section starts by describing our analysis results. In particular, the effect of different refactoring methods on the performance in terms of preliminary measures (precision and

recall) and browsability measures (MAP (Eq. 1.4) and DiffAR (Eq 1.5)) is described. The section then proceeds by further exploring the effect of each refactoring method in greater detail. In particular, we compare the performance of methods that have positive impact on traceability with other related techniques in automated tracing, and explore strategies for mitigating any potential negative impact certain refactorings methods might have on the performance.

Table 5.2

Wilcoxon Signed Ranks Test results (***p**-values* at $\alpha = .05$) for Primary Performance Measures

	iTrust		eTour		WDS	
	Recall (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)
Refactorings						
RI	(-2.395, <.010)	(-2.803, <.005)	(-2.701, <.007)	(-2.803, <.005)	(-2.090, ¡.05)	(-2.803, <.005)
MM	(-.405, .686)	(-1.599, .110)	(-1.753, .080)	(-.663, .508)	(-1.572, .116)	(.000, 1.000)
XM	(-2.803, <.005)	(-2.803, <.005)	(-2.701, <.007)	(-2.803, <.005)	(-2.803, <.005)	(2.701, <.007)
	MAP (Z, *p-value*)	**DiffAR** (Z, *p-value*)	**MAP** (Z, *p-value*)	**DiffAR** (Z, *p-value*)	**MAP** (Z, *p-value*)	**DiffAR** (Z, *p-value*)
Refactorings						
RI	(-.357, .721)	(-1.732, .083)	(2.380, <.010)	(-1.414, .157)	(-2.803, <.005)	(-1.000, .317)
MM	(-.653, .514)	(.000, 1.000)	(1.478, .139)	(.000, 1.000)	(-1.680 .093)	(.000, 1.000)
XM	(-2.803, <.005)	(-2.842, <.005)	(-2.809, <.005)	(-2.803, <.005)	(-2.803, <.005)	(-3.051, <.005)

5.5.1 Analysis Results

Figure 5.10 shows the recall and precision curves of our three datasets after applying RENAME IDENTIFIER (RI), MOVE METHOD (MM), and EXTRACT METHOD (XM), in comparison to the VSM baseline. Statistical analysis over the results is shown in Table 5.2.

In general, the results show that different refactorings vary in their impact on the performance. In details, RENAME IDENTIFIER refactoring has the most obvious positive impact on the results, affecting the recall significantly in all three datasets. In the *iTrust* dataset, both precision and recall have improved significantly, achieving optimal recall levels at higher thresholds. The same performance is detected in the *eTour* dataset, in which the improvement in the recall and the precision over the baseline is statistically significant. In the *WDS* dataset, the precision has dropped significantly with the significant increase in the recall. This can be explained based on the inherent trade-off between precision and recall. In this particular dataset, even though renaming identifiers has helped to retrieve more true positives, it also retrieved a high number of false positives.

The results also show that MOVE METHOD refactoring has the least influence on the performance. In all datasets no significant improvement in the recall or the precision is detected. In fact, the performance after applying this particular refactoring is almost equivalent to the baseline. In contrast, statistical analysis shows that EXTRACT METHOD has resulted in a significant increase in the precision. However, when applied, it was no longer possible to achieve high recall, hence the performance lines in Figure 5.10 stopped at recall of 66%, 61%, and 93% in *iTrust*, *eTour*, and *WDS* respectively. In general, In terms of recall, the results show that removing redundant textual knowledge from the system has caused a significant drop in the number of true links, taking the recall down to significantly lower levels in all three datasets. The spike in the precision can be simply explained based on the inherent trade-off between precision and recall.

In terms of browsability, statistical analysis in Table 5.2 shows that both RENAME IDENTIFIER and MOVE METHOD have no significant impact on the average DiffAR. However, EXTRACT METHOD seems to be achieving significantly better performance over the baseline. In terms of MAP, Figure 5.2 shows the superior performance of EXTRACT METHOD over other refactorings in comparison to the baseline. This behavior can be explained based on the fact that VSM retrieves the smallest number of links after applying EXTRACT METHOD. However, even with lower recall, only a few false positives were separating true positives, with most of these true positives located toward the top the of list, thus taking the precision of these links to higher levels, which in turn resulted in higher MAP values.

MAP results also show the inconstant performance of RENAME IDENTIFIER across the different datasets. In the *iTrust*, no significant difference in the performance is detected. In contrast, in the *eTour* dataset, RENAME IDENTIFIER achieves significantly better performance than the baseline and significantly worse performance in *WDS*. In addition, analysis results show that MOVE METHOD does not have any significant impact on the MAP values, which is actually expected based on the fact that it does not have a significant impact on the primary performance measures.

In general, our results suggest that RENAME IDENTIFIER refactoring has the most significant positive effect on the results, improving the recall to significantly higher levels in all three datasets. In contrast, EXTRACT METHOD has a significantly negative impact, taking the recall down to significantly lower levels in all three datasets, and MOVE METHOD has no clear impact on the performance. Automated tracing methods emphasize recall

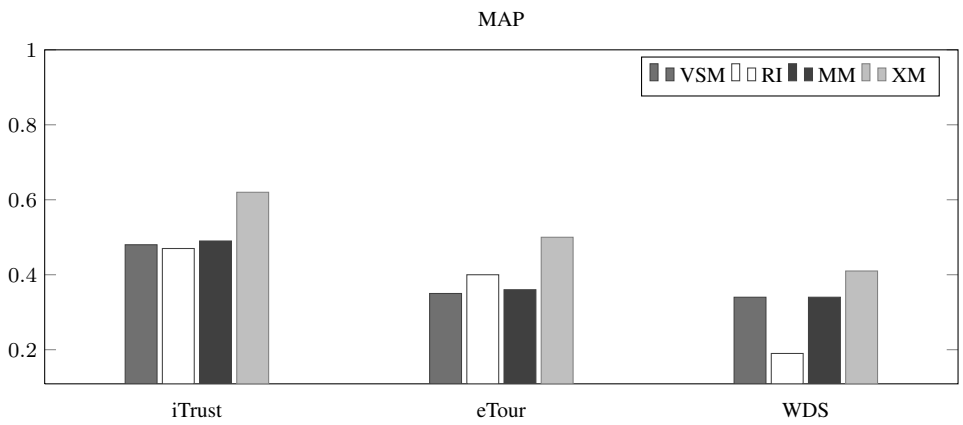


Figure 5.2

MAP Values in iTrust, eTour and WDS after Applying Different Refactorings

over precision [121]. This argument is based on the observation that an error of commission (false positive) is easier to deal with than an error of omission (false negative). Based on that, we conclude that RENAME IDENTIFIER refactoring has the most potential as a performance enhancement technique for IR-based requirements-to-code automated tracing. In what follows, the operation of each refactoring is discussed in greater detail.

5.5.2 Rename Identifier Effect

Our results suggest that restoring textual information has the most positive impact on the system's traceability. In particular, RENAME IDENTIFIER refactoring targets the vocabulary mismatch problem in software artifacts, which seems to be the most dominant problem affecting IR-based traceability tools [90]. In the automated tracing literature, the vocabulary mismatch problem is often handled by using semantics to fill the textual gap caused by poor coding habits [94, 121, 172]. In order to gain better insights into the oper-

ation of RENAME IDENTIFIER; we compare its performance with the performance of two semantic enhancement strategies including ESA (Sec. 3.2.4.1) and VSM-T (Sec. 3.2.2.1).

To compare the performance of these two techniques with RENAME IDENTIFIER, we trace our datasets using both VSM with thesaurus support (VSM-T) and ESA, before applying RENAME IDENTIFIER refactoring, and compare their performance to the VSM baseline after applying RENAME IDENTIFIER (VSM-RI). The results are shown in Figure 5.11 and Table 5.3. Results show that query expansion technique (ESA) was able to hit almost a 100% recall at higher threshold levels in all datasets; however, the precision was affected negatively due to the high number of false positives. In general, textual enrichment of artifacts might have a positive influence on the recall, especially retrieving some of the hard-to-trace requirements [94]; however, it has a significant negative impact on the accuracy, which was reflected in the fast drop in the precision values at higher threshold levels. In contrast, the results show that VSM with a domain-specific thesaurus support was able to achieve a comparable performance to our refactoring-based approach; no statistically significant difference in terms of precision and recall was detected in all of our three datasets.

To demonstrate the operation of these three different techniques we refer to the example in Figure 5.3. This figure shows a true trace link between requirement 6.2.3, which describes a basic forgotten password recovery functionality, and the method FP_OnClick, which implements this particular requirement. Figure 5.4 shows the refactored method after applying our RENAME IDENTIFIER procedure described in Section 5.3.1. Figure 5.5 shows a snapshot of our domain-specific thesaurus. Basically, entries were added to handle

abbreviations and basic domain-specific *synonymy* relations. Figure 5.6 shows the query expansion terms that have been added after applying ESA to both ends of the trace link. In particular, the link has been expanded with several unrelated terms, extracted from the general purpose knowledge source. For example, the list of semantically related terms for the term <*forget*> from requirement 6.2.6 includes many domain irrelevant terms such as <*bury*, *leave*>. This explains the high noise-to-signal ratio returned by this method, causing the retrieval of a large number of incorrect links.

Table 5.3

Wilcoxon Signed Ranks Test results (***p-values*** at $\alpha = .05$) for Query Expansion and Sign Preserving Techniques

	iTrust		eTour		WDS	
	Recall (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)
Technique						
VSM-T	(-.051, .959)	(-1.599, .110)	(-.652, .515)	(-.178, .859)	(-1.478, .139)	(-.866, .386)
ESA	(-2.490, <.05)	(-2.380, <0.05)	(-2.803, <.005)	(-2.701, <.05)	(-2.842, <.005)	(-3.051, <.005)
	Recall (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)	**Recall** (Z, *p-value*)	**Precision** (Z, *p-value*)
Technique						
Summarization	(-.357, <.005)	(-2.842, <.005)	(-3.051, <.005)	(-2.803, <.005)	(-2.803, <.005)	(-2.090, <.01)
Labeling	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)

5.5.3 Handling Code Clones

A surprising observation in our analysis is that removing redundant information from software artifacts has a negative impact on the performance of IR-based automated tracing tools. This suggests that code clones actually serve a positive purpose for traceability link

6.2.3 Forget Password
The system should provide a functionality for recovering user's password. New password should be sent to the user's registered email.

(a) Requirement 6.2.3

```
1.  public bool FP_OnClick(string uName)
2.      {
3.          If(validteUsrNm(uName));
4.              stEmail = getUsrEml(uName);
5.              stPwd = genRndPWD();
6.              sendPwd(stEmail, stPwd);
7.              return true;
8.          else Error =”Invalid Credentials”;
9.              return false;
10.     }
```

(b) Method FP_OnClick

Figure 5.3

A Trace Link between Requirement 6.2.3 and Method FP_OnClick

```
1.  public bool ForgetPassword_OnClick(string userName)
2.      {
3.          If(validteUserName(userName));
4.              stEmail = getUserEmail(userName);
5.              stPassword = generateRandomPassword();
6.              sendPassword(stEmail, stPassword);
7.              return true;
8.          else Error =”Invalid Credentials”;
9.              return false;
10.     }
```

Figure 5.4

Applying RENAME IDENTIFIER on Method FP_OnClick

- ***pwd***: password
- ***credential***: email, password
- ***usr***: user

Figure 5.5

Domain Thesaurus Support

- ***credential***: *certificate, password, email*
- ***validate***: *formalize, formalize, corroborate*
- ***password***: *watchword, word, parole, countersign*
- ***recover***: *retrieve, find, regain, recuperate, reclaim*
- ***forget***: *bury, block, blank out, leave*

Figure 5.6

ESA Query Expansion

recovery. However, there is a conventional wisdom that code cloning is generally a bad development practice. From a refactoring perspective, code clones are considered a code smell [17]. They increase the maintenance cost and the error proneness of the code as inconsistent changes to code duplicates can lead to unexpected behavior. Therefore, code clones have to be refactored whenever detected [139, 219].

To mitigate the impact of removing code clones on the system traceability, we suggest a sign-preserving treatment to reverse the negative effect of EXTRACT METHOD refactoring. Applying this treatment, whenever a redundant code (a code clone) is removed, appropriate comments that describe that code can be automatically inserted to fill the textual gap left by refactoring that particular code. This can be achieved by utilizing automatic techniques to generate descriptions for source code [126]. Several techniques for code labeling

and summarization have been introduced in the literature [53, 109, 126]. Next we experiment with two commonly used techniques, at different levels of complexity, for preserving traceability signs.

5.5.3.1 Code Summarization

Refers to the creation of a shortened version of a computer program by capturing and preserving the subject matter of the code [126]. Summarization is often performed with the objective of producing meaningful summaries of source code to aid program comprehension [109]. To generate code summaries of code clones we use Latent Semantic Indexing (LSI) [60], a technique that is used very often for automatically extracting summaries of natural language. In particular, we adopt the approach proposed by Haiduc et al. [109] to extract the most important terms in a certain code. This approach has been shown to achieve high correlation with human-generated summaries [109]. The process starts by indexing the source code corpus at the method level. The cosine similarity is then computed between the code clone's profile and each of the terms in the corpus in the LSI-reduced space. The corpus terms are then ranked in decreasing order based on their similarity with the code clone. The summary is then constructed by considering the top N terms in the ranked list.

5.5.3.2 Code Labeling

Refers to the extraction of a set of representative words for a certain code element. A simple code labeling can be achieved by indexing the redundant code (removing stopwords, splitting code identifiers into their constituent words, and performing stemming [53]).

The resulting words are then added as comments to replace the removed code. The term *labeling* stems from the fact that no human-like meaningful descriptions are generated; instead, just discrete words (labels) are extracted to facilitate information retrieval. Figure 5.8-a, b show the outcome of applying the code summarization procedure and the code labeling procedure respectively over the code clone shown in Figure 5.7.

To evaluate the effectiveness of these two techniques in preserving traceability signs we perform missing traceability sign analysis. In particular, we calculate the percentage of lost effective signs when removing code clones. We define an effective sign as a term or a keyword that contributes to a trace link (appears in both ends of the traceability link). Figure 5.9 shows the percentage of lost traceability signs in *iTrust*, *eTour* and *WDS* after applying EXTRACT METHOD (XM), and after applying code labeling and code summarization techniques(N=7). The figure shows that the indexing-based code labeling was actually more successful than code summarization in preserving a large number of the original signs lost after removing code clones. The best performance of the code summarization technique was achieved at (N=7). The poor performance of code summarization can be explained based on the fact that size of a redundant code is not sufficient to produce meaningful summaries. For example, often the redundant code is just a part of a method. Such code fragments usually lack valuable information that can be useful to the summarization process. For instance, in our example in Figure 5.8-a, the code clone does not include the method's signatures, which has been found to add a significant information value to the generated summaries [53, 109, 234]. These findings come actually aligned

```
1.  try {
2.      conn = factory.getConnection();
3.      ps = conn.prepareStatement("SELECT * FROM NDCodes, OVMedication, OfficeVisits "
4.          + "WHERE NDCodes.Code=OVMedication.NDCode AND OVMedication.VisitID=OfficeVisits.ID "
5.          + "AND PatientID=? AND ((DATE(?) < OVMed.EndDate AND DATE(?) > OVMed.StartDate)"
6.          + "OR (DATE(?) > OVMedication.StartDate AND DATE(?) < OVMedication.EndDate ) OR "
7.          + "(DATE(?) <= OVMedication.StartDate AND DATE(?) >= OVMedication.StartDate)) "
8.          + "ORDER BY VisitDate DESC");
9.      ps.setLong(1, patientID);
10.     ps.setString(2, startDate);
11.     ps.setString(3, startDate);
12.     ps.setString(4, endDate);
13.     ps.setString(5, endDate);
14.     ps.setString(6, startDate);
15.     ps.setString(7, endDate);
16.     ResultSet rs = ps.executeQuery();
17.     return loader.loadList(rs);
18. } catch (SQLException e) {
19.     e.printStackTrace();
20.     throw new DBException(e);
21. }
```

Figure 5.7

A Code Clone Detected in the *iTrust* Dataset

```
1. /* factory ND Code ID End */
2. byDate(patientID, startDate, endDate);
```

(a) Comments Generated by the LSI-based Code Summarization Technique

```
1. /** factory
2. * ND Code
3. * OV Medication
4. * Office Visit
5. * Visit ID
6. * Office Visit
7. * Patient ID
8. * End Date
9. * Star tDate
10. **/
11. byDate(patientID, startDate, endDate);
```

(b) Comments Generated by the Indexing-based Code Labeling Technique

Figure 5.8

Applying Traceability Sign Preservation on a Code Clone

with previous observations that simple techniques were shown to better reflect the subject matter of the code than other more complicated techniques such as LSI or LDA [53].

To further compare the effectiveness of these techniques, we integrate them into our experimental procedure after applying EXTRACT METHOD. We then re-trace all of our experimental datasets. Results are shown in Figure 5.12 and Table 5.3. The results show that, when the indexing-based code labeling procedure is used, no statistically significant drop in terms of precision or recall is detected in any of our three datasets i.e., the performance is unaffected by removing the clones. In contrast, while applying the LSI-based code summarization technique helps to preserve some of the effective traceability signs, it still could not improve the results significantly.

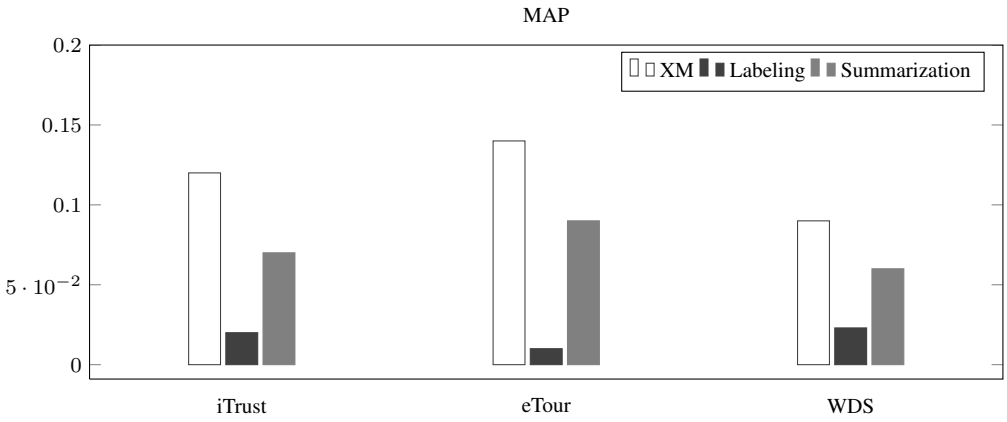


Figure 5.9

Percentage of Lost Traceability Signs in iTrust, eTour and WDS

5.5.4 Moving Information

Our results show that moving misplaced information among the system's modules has no significant impact on the performance. This suggests that misplaced signs might not be as problematic for IR-based traceability as missing or duplicated signs. This phenomenon can be explained based on the fact that *Feature Envy* code smell tends to be less dominant and more complicated to detect in software systems than other code smells such as code clones [19, 220, 245]. In fact, further analysis shows that even when a method is moved to another class, it is often still highly referenced (called) in its original class, thus the track is unlikely to get discontinued, causing MOVE METHOD refactoring to have no obvious impact on the performance. However, it is important to point out that in some cases, where a high density of misleading signs were detected, MOVE METHOD was able to reverse that effect, thus resulting in a slight increase in the recall, especially in *WDS* and *eTour*, however, that improvement was statistically insignificant.

5.5.5 Discussion

Our analysis has revealed that, in terms of precision and recall, maintaining a domain-specific thesaurus can be equivalent to applying RENAME IDENTIFIER refactoring. Therefore, this particular refactoring can be considered as an alternative strategy to handle vocabulary mismatch in software artifacts. However, refactoring provides a more systematic way to handle this problem. In other words, instead of separately maintaining an external *ad-hoc* dictionary of the system's vocabulary and their synonyms, this process can be handled internally through refactoring, as an integral part of the evolution process. As

mentioned earlier, RENAME IDENTIFIER refactoring is in fact the most applied refactoring in practice, and it has already been integrated in most contemporary IDEs [3, 66, 187, 190].

In terms of effort, as Table 5.1 shows, the amount of effort required to rename identifiers can be comparable to the external thesaurus technique. While in RENAME IDENTIFIER human analysts still have to select new identifiers' names, when building an external thesaurus, synonym relations have to be identified manually. In addition, both methods require a sufficient knowledge of the system's application domain. However, thesaurus-based methods often require calibrating a certain parameter (α in Eq.7) to achieve the desired performance levels [121], while no calibration or optimization is required when applying RENAME IDENTIFIER. In addition, the procedure we propose to identify renaming opportunities helps to alleviate a considerable amount of the effort required to identify misleading signs. In fact, the research on fully automating this process has noticeably advanced in recent years, especially in the domain of acronyms and abbreviations expansion, opening the door for this process to be fully automated [35, 150, 242].

Our results also show that a simple code labeling technique can fill the vocabulary gap that might result from removing code clones in the system. In terms of effort, code labeling techniques are fully automated, so the human effort is minimized. However, it is important to point out that these implanted labels (signs) are also subject to become outdated as code evolves, thus generating misleading tracks. Therefore, it is important to keep such labels up-to-date and in sync with any changes affecting the code segments they describe.

Finally, even though moving misplaced signs in the system did not result in a statistically significant improvement in the performance, such refactoring can still have an in-

fluence on traceability, especially in safety critical systems, where losing even one critical link could be detrimental [45]. However, unlike the renaming process, MOVE METHOD is a nontrivial process, and often results in introducing bugs in the system [190]. Therefore, a careful cost-benefit analysis might be required to determine if performing such transformation is worthwhile.

Our findings in this chapter helped in exploring several issues related to applying refactoring as a performance enhancement strategy in IR-based automated tracing. In particular, our study provides insights into developers' actions that might have an impact on the system's traceability during evolution, and reinforced past proposals advocating the use of consistent, and regular vocabulary in identifiers' names [27]. In addition, our analysis revealed how potentially negative effects of removing code clones could be reversed through code labeling, an option that might be important to have in code clone refactoring tools [20, 129].

5.6 Limitations

The experiment presented in this chapter has several limitations that might affect the validity of the results. Threats to external validity impact the generalizability of results [59]. In particular, the results of this study might not generalize beyond the underlying experimental settings. A major threat to our external validity comes from the datasets used in this experiment. In particular, two of the projects were developed by students and are likely to exhibit different characteristics from industrial systems. We also note that our traceability datasets are of medium size, which may raise some scalability concerns. Nevertheless, we

believe that using three datasets from different domains, including a proprietary software product, helps to mitigate these threats.

Another threat to the external validity might stem from the fact that we only experimented with three refactorings. However, the decision of using these particular refactorings was based on careful analysis of the IR-based automated tracing problem. In addition, these refactorings have been reported to be the most frequently used in practice [176, 256]. Another concern is the fact that only requirements-to-code-class traceability datasets were used. Therefore, our findings might not necessarily apply to other types of traceability such as requirements-to-requirements, requirements-to-design or even different granularity levels such as requirements-to-method. However, our decision to experiment only with requirements-to-class datasets can be justified based on the fact that refactoring has accelerated in source code, especially Object-Oriented code, more than any other types of artifacts, thus we find it appropriate at the current stage of research to consider this particular traceability type at this granularity level.

Other threats to the external validity might stem from specific design decisions such as using VSM with TFIDF weights as our experimental baseline. Refactoring might have a different impact on other IR methods such as LSI and ESA, thus, different results might be obtained. Also, a threat might come from the selection of procedures and tools used to conduct refactoring. However, we believe that using these heavily used and freely available open source tools helps to mitigate this threat. It also makes it possible to independently replicate our results.

Internal validity refers to factors that might affect the causal relations established in the experiment. A major threat to our study's internal validity is the level of automation used when applying different refactorings. In particular, an experimental bias might stem from the fact that the renaming process is a subjective task carried out by the researchers. In addition, human approval of the outcome of the refactoring process was also required. However, as mentioned earlier, in the current state-of-the-art in refactoring research and practice, human intervention is a must [86, 184]. In fact, it can be doubtful whether refactoring can be fully automated without any human intervention [131]. Therefore, these threats are inevitable. However, they can be partially mitigated by automation.

In our experiment, there were minimal threats to construct validity as standard IR measures, which have been used extensively in requirements traceability research, were used to assess the performance of different treatments (recall, precision, MAP and DiffAR). We believe that these two sets of measures sufficiently capture and quantify the different aspects of methods evaluated in this study.

5.7 Related Work

Our work in this chapter can be classified under the research category of and managing traceability in evolving software systems. In particular, we investigate on strategies to mitigate the risks of software evolution on traceability. In what follows, we review seminal work in this domain, and briefly describe how such work relates to, or can be distinguished from, our work.

Cleland-Huang et al. [44] presented an event-based approach that establishes traceability links through the use of publish-subscribe relationships between dependent objects in the system. When a change to a certain requirement occurs, an event notification message is published to all the subscribed dependent objects. Therefore, ensuring that all these publish-subscribe relations (trace links) are up-to-date or consistent during system evolution. Our work can be distinguished from this work based on the fact that this approach handles the change from the requirement side of the link, while the proposed approach in this chapter handles evolution from the source code side. Our approach is based on the observation that code is more prone to change than requirements [21, 154]. Therefore, working on that side of the link is expected to have more immediate effect on traceability.

Egyed [72] proposed an approach that uses observations about the runtime behavior of the system to detect associations among functional scenarios and their executing code. In particular, traces are defined based on the data flow in the form of a footprint graph. A footprint is defined as the lines of code used while executing a scenario. Using our approach, the code does not have to execute or even compile, thus avoiding complications related to the runtime behavior of the system. In addition, no test cases or usage scenarios are needed.

Antoniol et al. [13] proposed an automatic approach to identify class evolution discontinuities due to possible refactorings. The approach identifies links between classes obtained from refactoring, and cases where traceability links were broken due to refactoring. Our approach can be related to this work in the sense that we propose mitigation strategies to overcome problems that may result from certain refactorings (EXTRACT METHOD).

Moreover, we propose the use of refactoring as a pre-processing step to enhance the performance, rather than only dealing with the implications of already applied refactorings.

Mäder et al. proposed a rule-based traceability approach for maintaining traceability relations during evolutionary change [167]. This approach revolves around the monitoring of elementary changes that take place to UML model elements, and updating a pre-existing set of traceability relations associated with such changes. This insures that changes in the system's structure will be reflected on traceability, thus keeping such relations up-to-date. However, this approach is restricted to the scope of UML-based, Object-Oriented (OO), software engineering. In contrast, refactoring is not limited to structural and OO code, and no UML models have to be generated for the system.

The approach proposed by Charrada et al. [21] tackles the problem that we tackle in this chapter from a different perspective. In particular, the authors proposed an approach to automatically identifying outdated requirements by analyzing source code changes during evolution to identify the requirements that are likely to be impacted by the change. This approach can be complementary to our approach. While our approach works on the decaying vocabulary structure from the code side, their approach works on the same problem but from the opposite side of the traceability link (the trace query). This will accelerate the process of bridging the textual gap in the system.

Finally, since this chapter is based on Gotel and Morris's theoretical approach of IR-based automated tracing [99], we find it appropriate here to end our discussion with Gotel's latest views on the field. In their most recent roadmap paper, Gotel et al. [96] identified a number of challenges for implementing effective software and systems traceability. In

the set of short-term goals that they specified, they emphasized the need for researchers to focus on mechanisms to mix and match approaches to achieve different cost and quality profiles. The work we presented in this chapter is aligned with that goal. In particular, our objective is to add to the current incremental effort of this domain in a way that helps to move forward on the automated tracing roadmap.

5.8 Conclusions and Future Work

In this chapter, we explored the effect of applying various refactoring methods on the different performance aspects of IR-based tracing methods. Our main hypothesis is that certain refactorings will help to reestablish the decaying traceability tracks of evolving software systems, thus helping IR methods to recover more accurate lists of candidate links. To test our research hypothesis, we examined the impact of three refactorings on the performance of three requirements-to-code datasets from different application domains. In particular, we identified three main problems associated with IR-based automated tracing including: missing, misplaced, and duplicated signs, and we suggested three refactorings to mitigate these problems. Results showed that restoring textual information in the system's artifacts (RENAME IDENTIFIER) had a significantly positive impact on the performance. In contrast, refactorings that remove redundant information (EXTRACT METHOD) affected the performance negatively. The results also showed that moving information between the system's modules (MOVE METHOD) had no noticeable impact on the performance.

Furthermore, in our analysis, we compared the performance of RENAME IDENTIFIER with two other commonly used techniques for handling the vocabulary mismatch problem

in software systems. These methods include retrieval enhancement with thesaurus support and query expansion techniques. We analyzed the performance of these techniques, exploited their limitations, and demonstrated how refactoring could address these limitations. In addition, we suggested a sign-preserving technique to mitigate the negative impact of refactoring code clones on traceability. In particular, we proposed two methods for generating source code descriptions including code labeling and summarization, and we analyzed and evaluated their effectiveness in preserving traceability information. Results showed that simple code labeling was more successful than code summarization in preserving traceability signs from getting lost when code clones were refactored. Furthermore, an effective traceability sign analysis was conducted to quantify the effect of different investigated refactorings on the traceability tracks in our experimental systems.

The line of work in this chapter has opened several research directions to be pursued in our future work. These directions can be described as follows:

- Refactoring: In our future analysis, we are interested in investigating the effect of other refactorings on traceability. In particular, refactorings that work on the structural information of the system (e.g., EXTRACT CLASS [84]), and target different granularity levels, will be investigated.
- IR methods: In our future work, we are interested in exploring the effect of refactoring on other IR methods that are often used in automated tracing, such as (LSI) [174] and (LDA) [16, 28]. These methods work by exploiting hidden (latent) structures in software systems, rather than directly matching keywords in software artifacts, thus they might have a different response to different refactoring methods.
- Tool support: In terms of tool support, a working prototype that implements our findings in this chapter is in order. A working prototype will allow us to conduct long term studies that will give us a better understanding of the role of human analysts in the process [197]. In particular, quantifying the potential effort-saving of our approach, its usability, and scope of applicability.
- Automation: As observed in our experiment, there is still a major effort concern when it comes to refactoring. Humans still play a major role in controlling the

refactoring process, starting from approving refactoring candidates captured by code smell detection tools, to applying required refactorings, and verifying the outcome of the process. Therefore, in our future work we will be exploring refactoring automation strategies that can help to alleviate some of the manual effort in the process.

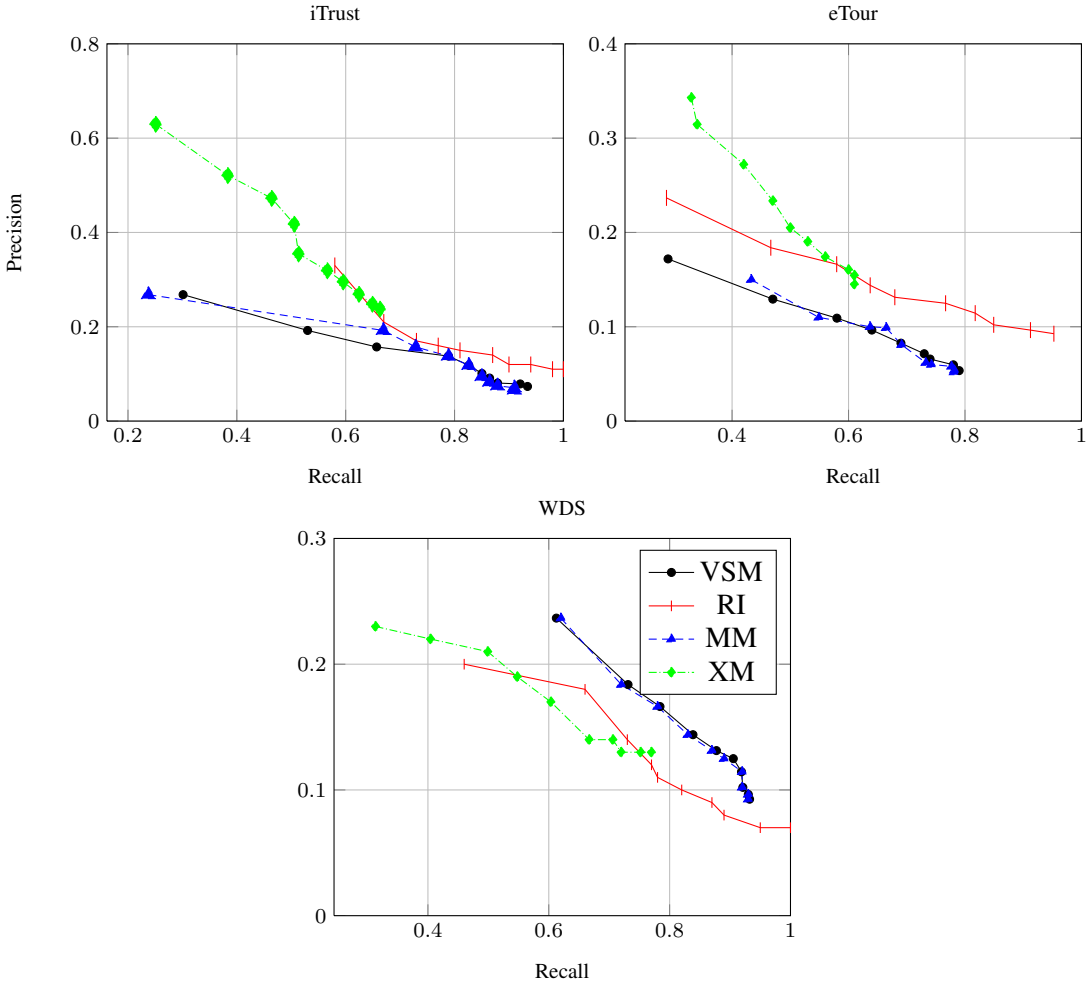


Figure 5.10

Performance after Applying Different Refactorings

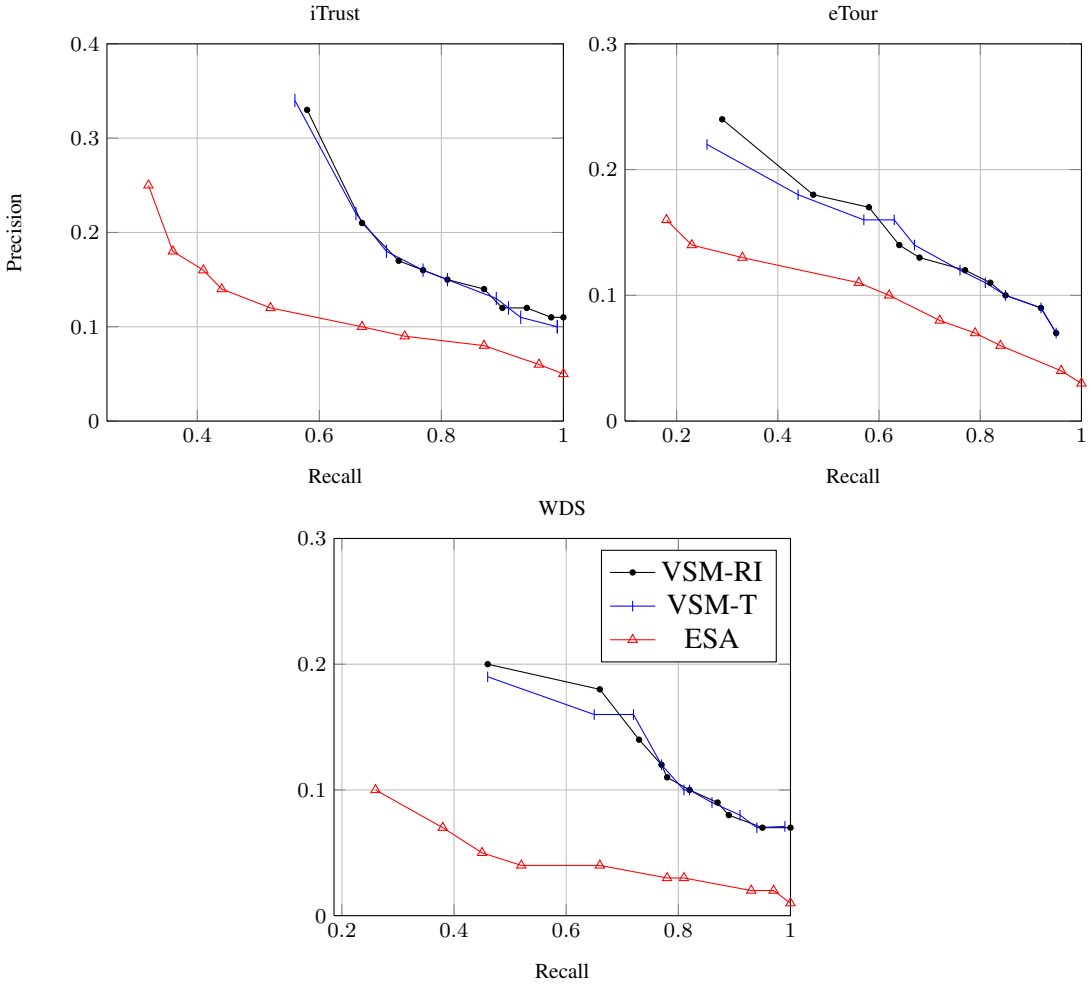


Figure 5.11

Comparing the Performance of (VSM-RI), with (VSM-T) and (ESA)

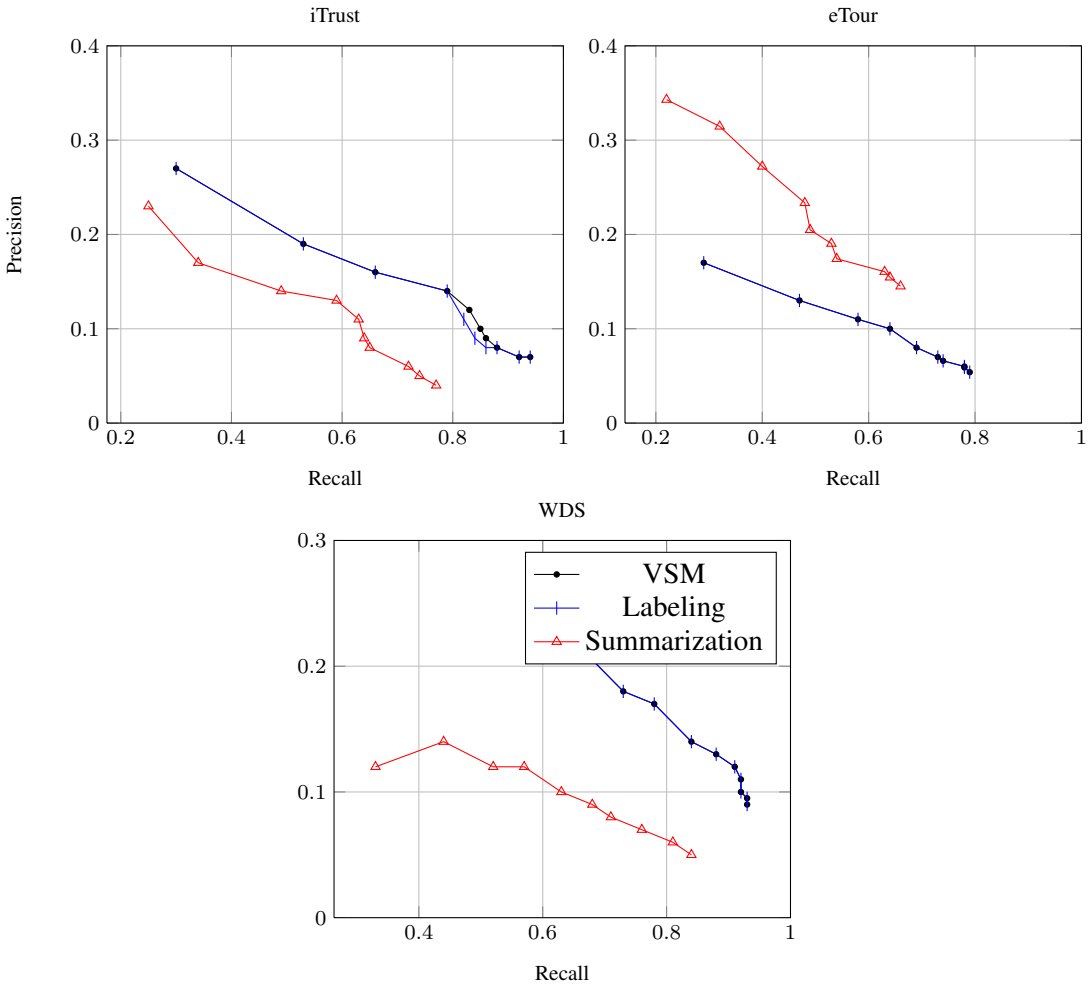


Figure 5.12

Preserving Traceability Signs Code Summarization and Code Labeling

CHAPTER 6

HUMAN ASPECTS AND TOOL SUPPORT

Studying human analyst's behavior in automated tracing is a new research thrust. Building on a growing body of work in this area, we offer a novel approach to understanding requirements analyst's information seeking and gathering. We model analysts as predators in pursuit of prey — the relevant traceability information, and leverage the optimality models to characterize a rational decision process. The behavior of real analysts with that of the optimal information forager is then compared and contrasted. The results show that the analysts' information diets are much wider than the theory's predictions, and their residing in low-profitability information patches is much longer than the optimal residence time. These uncovered discrepancies not only offer concrete insights into the obstacles faced by analysts, but also lead to principled ways to increase practical tool support for overcoming the obstacles.

6.1 Introduction

One area that traceability is indispensable is the engineering of mission- or safety-critical software systems. Under these circumstances, a human analyst must vet (e.g., browse and validate) the candidate RTM offered by the tool [49]. Vetting is thus a cen-

tral activity in *assisted requirements tracing*, in which a human analyst engages with an automated tracing tool to perform the assigned tracing task [62].

Studies of human behavior [49, 62, 119, 138] showed that, in interacting with the tracing tool to prepare their final RTM, the analysts made both errors of omission (threw out correct links) and errors of commission (added incorrect links). A thoroughly conducted experiment by Dekhtyar et al. [62] tested 11 vetting variables. The results revealed that only the accuracy of the initial TM and the analyst effort expended in validating offered links had statistically significant effects on the final TM, while the other 9 factors (e.g., tool used, tracing experience, effort on searching for omitted links, etc.) did not make a difference [62]. A qualitative study by Kong et al. [138] provided additional insights into analysts' behavior. For example, all the analysts were observed to make multiple correct decisions in a row, and such correct-decision bouts were interleaved with streaks of incorrect decisions [138].

Currently, empirically observing analysts' behavior is adopted as the main methodology for researching the human factors in assisted requirements tracing. Observational studies are particularly valuable in answering "*what*" questions by uncovering behavioral patterns. However, little is known about "*why*" analysts behave in a certain way and "*how*" to improve the analysts' tracing performance in a principled manner. Addressing these knowledge gaps is of vital importance because, with a deeper theoretical understanding about the fundamental mechanisms underlying the analysts' behavior, the empirical observations can be related more coherently and the key factors can be tested more completely.

In this work, we explore the theoretical underpinning of analyst's requirements tracing behavior based on Pirolli's *information foraging theory* [208]. The theory uses our animal ancestors' "built-in" food-foraging mechanisms [235] to understand human information seeking and gathering in the vastness of the Web. Lawrance and colleagues [146–149] have recently pioneered the application of information foraging theory to the debugging domain, and presented encouraging results that matched the theory's predictions with the developers' actual code navigations. Building on their influential work, we aim to investigate human analysts' requirements tracing strategies in light of foraging theory's constructs and principles.

This chapter reports an exploratory study that examines two of foraging theory's foundational models [208] in the context of tracing: 1) the *diet model* optimizes the decision related to what kinds of information to consume and what to ignore; and 2) the *patch model* determines the optimal time to spend in an information patch. These optimality models allow us to define the behavioral problems that are posed by the requirements tracing environments, and therefore allow us to determine how well an optimal forager (analyst) performs on those problems. We then compare the behavior of real analysts with that of the optimal forager. In particular, the theory's predictions are confronted with the tracing behavior of 6 analysts who validated the links between requirements and code of a software system in the healthcare domain. The departures from optimality revealed by the comparison not only offer concrete insights into the obstacles faced by the human analysts, but also lead to principled ways to overcome the obstacles.

The contributions of our work lie in the analysis of optimality within the shaping limits placed by the task and the information environments. Our work advances the fundamental understanding about analysts' information seeking in light of the adaptiveness of human behavior. This improved understanding, in turn, enables principled ways to increase practical support for software traceability. In what follows, we present background information on foraging theory and assisted requirements tracing in Section 6.2. We then detail our research methodology in Section 6.3. Sections 6.4 and 6.5 describe the empirical study's design and results respectively. The implications of our work are discussed in Section 6.6, and finally, Section 6.7 concludes the chapter.

6.2 Background and Related Work

6.2.1 Optimal Foraging Theory

Animals adapt, among other reasons, to increase their rate of energy intake. To do this they evolve different methods: a wolf hunts for prey, but a spider builds a web and allows the prey to come to it. *Optimal foraging theory* is developed in biology for analyzing the adaptive value of food-foraging strategies [235]. Optimality here refers to the strategy that maximizes the gain per unit cost. Central to optimal food foraging are the *diet model* and the *patch model*.

The *diet model* deals with the tradeoffs when a predator forages in an environment in which food is distributed in a patchy manner. Figure 6.1-a illustrates the patchy environment by presenting a hypothetical bird foraging in berry clusters. The forager must expend some amount of between-patch time (t_B) arriving at a prey-patch, and t_W denotes

the within-patch foraging time. Figure 6.1-c, depicts Charnov's Marginal Value Theorem [235] which states that the rate-maximizing time to spend in patch, t^*, occurs when the slope of the within-patch gain function $g(t_W)$ is equal to the average rate of gain, which is the slope of the tangent line R^*. In this figure $g(t_W)$ represents a decelerating expected net gain function. The amount of energy gained per unit time of foraging is therefore $R = g(t_W)/(t_B + t_W)$. Following Stephens and Krebs [235], we use $\pi = g/t_W$ to denote the patch's profitability and use $\lambda = 1/t_B$ to denote the patch's encounter rate.

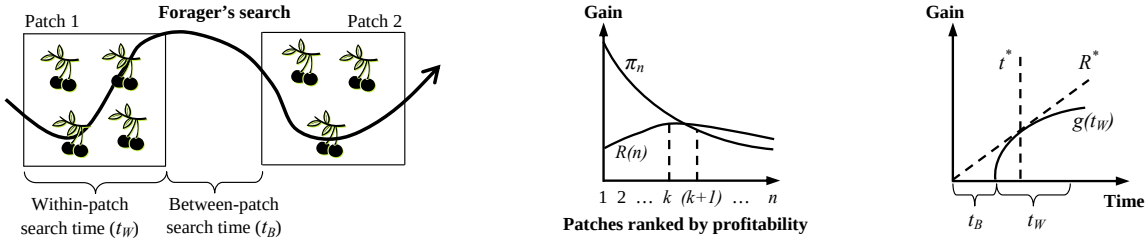


Figure 6.1

(a) Patchy environment. **(b)** Optimal Diet. **(c)** Charnov's Marginal Value Theorem

The optimal diet selection follows the principle of lost opportunity [235]. Intuitively, the principle states that the prey-patch is predicted to be ignored if its profitability is less than the expected rate of gain of continuing search for other prey-patches. Formally, as shown in Figure 6.1-b, let the patches be ranked by profitability, $\pi_i = g_i/t_{Wi}$, such that $\pi_1 > \pi_2 > \ldots > \pi_n$. The optimal diet can be expanded by adding prey-patches in order

of decreasing profitability (i.e., from 1 to n) until the rate of gain for a diet of the top k prey-patches is greater than the $k+1$st prey-patch's profitability,

$$\left(R(k) = \frac{\sum_{i=1}^{k} \lambda_i \cdot g_i}{1 + \sum_{i=1}^{k} \lambda_i \cdot t_{Wi}}\right) > \left(\frac{g_{k+1}}{t_{W\,k+1}} = \pi_{k+1}\right). \tag{6.1}$$

The left side of the inequality concerns the rate of gain obtained by the diet of the k highest profitability prey-patches, whereas the right side concerns the profitability of the $k+1$st prey-patch. In Figure 6.1-b, the optimal diet contains $\{\text{prey-patch}_1, \text{prey-patch}_2, \ldots, \text{prey-patch}_k\}$, and therefore $\{\text{prey-patch}_{k+1}, \ldots, \text{prey-patch}_n\}$ are predicted to be ignored by the forager.

Once a prey-patch is selected to be part of the forager's diet, the *patch model* deals with predictions of the amount of time to spend within the patch. The basic idea is illustrated in Figure 6.1-c. As the forager gains energy, the amount of food diminishes or depletes. Consequently, there will be a point at which the expected gains from foraging within the current prey-patch become less than the expected gains that could be made by leaving for a new one. Figure 6.1-c shows that the rate-maximizing time, t^*, occurs when the derivative of $g(t_W)$ is equal to the slope of the tangent line R^*.

In a nutshell, the simple rule in optimal foraging theory is: "do not expend more energy finding the food than the food provides." Animals (including humans) have evolved some very sophisticated and fascinating food-seeking mechanisms. Optimal foraging theory has been proven to be productive and resilient in addressing food-searching behavior in the field and the lab, whereby the adequacy of the tenets (e.g., the patch model and the diet model) is tested to account for the evolution of given structures or behavioral traits [235].

6.2.2 Foraging Theory Applied to Web and Code Navigation

Humans seeking information adopt various strategies, sometimes with striking parallels to those of animal foragers. The wolf-prey strategy bears some resemblance to classic information retrieval [175], and the spider-web strategy is like information filtering [227]. Pirolli [208] developed *information foraging theory* by laying out the basic analogies between food foraging and information seeking: predator (human in need of information) forages for prey (the useful information) along patches of resources and decides rationally on a diet (what information to consume and what to ignore). By adopting the optimality models and adding details where necessary, Pirolli raised foraging theory from the physical and biological levels to the knowledge and rational levels.

The main application area of information foraging theory is the study of users' information seeking on the Web. During Web navigation, users operate in two environments [208]. The *task environment* embodies a goal, problem, or task that drives human behavior, whereas the *information environment* structures users' interactions with the content. An optimal Web user's navigation is then calculated according to the notion of *information scent* [207], a subjective sense of value and cost of accessing an information source based on perceptual cues. The WUFIS (Web User Flow by Information Scent) algorithm [38] represents one of the most rational and effective computational models of information scent by considering both environments in its computation: 1) a spreading activation network [6] that represents user's goal memory in the task environment; and 2) an inter-word correlation representation used to approximate user's conception of word synonymy [225] in the information environment. Computing the Web user's "informa-

tion diet" provides remarkable insights into issues like link selection and decision to leave a webpage. As a result, information foraging theory has become extremely useful as a practical tool for website design and evaluation [233].

Inspired by human's adaptive interaction with information on the Web, researchers began to apply foraging theory in software engineering. Ko et al. [136] were among the first to relate information foraging to developers' seeking relevant code in software maintenance. In recent years, Lawrance et al. [146–149] have made tremendous strides in understanding programmer navigation during debugging by viewing programmer as predator and bug-fix as prey. Building on Pirolli's work, Lawrance et al. [147] developed the PFIS (Programmer Flow by Information Scent) model by combining: 1) a spreading activation over the source code's topology (analogous to links on webpages); and 2) a word similarity measure between the bug report and the source code (computed as vector space model's cosine similarity using the TF-IDF weighting schema [175]). Extending beyond Pirolli's work, Lawrance et al. [148] presented the PFIS2 model which incorporated the incremental changes in programmers' conception of the navigation goals during debugging. More recent work [206] focused on empirically assessing programmer navigation models' predictive accuracy and optimally composing single factors (e.g., recency, spatial proximity, etc.) into a family of PFIS3 models.

In summary, information foraging theory [208] provides an evolutionary-ecological approach to understanding human information-seeking on the Web. Applying the theory in software engineering has also been fruitful as the foraging-theoretic approach provides a foundation for studying developers' navigation around the code base [196]. Building

and expanding upon Lawrance and colleagues' seminal work on debugging [146–149], we investigate an important but different information-intensive software engineering task — assisted requirements tracing.

6.2.3 Assisted Requirements Tracing

Although IR-based tools automate the RTM generation to a large extent, in coping with mission- or safety-critical software systems, the human analyst must vet the candidate RTM produced by the tool and add and remove links as necessary to arrive at the final RTM [49]. It is important to emphasize that traceability is not an end in itself but a means towards some other end. The analyst who vets the candidate RTM may be involved in risk assessment, criticality analysis, regulatory compliance, or some other software engineering activities. As a result, the analyst can always override any tool's output and has the final say on whether or not the traceability is correct [119].

Assisted requirements tracing, thus, refers to the process in which the human analyst becomes actively involved and makes decisions concerning the automated tool's output. The foundational work in this area was laid by Hayes and Dekhtyar [119] where they elucidated the need to study human interaction (reaction) with the tracing tool's results. Since then, a series of studies [49, 62, 138] investigated analyst behavior and revealed that human tended to degrade the accuracy of the RTM provided by the tool. Among the important findings, a rather surprising one was that incorrect decisions were often made if the analyst spent much time on the links [138]. In order to understand the findings like this, it was suggested that we might need to experimentally study one variable at a time [50]. Eleven

variables were then examined by Dekhtyar et al. [62]. The work represents one of the most significant empirical discoveries to date.

In essence, assisted requirements tracing is aimed at providing the best of both worlds, allowing human and automated tool to do what they do best [62]. While recent empirical contributions have enlightened the vital role of human factors, we believe gaining a theoretical understanding can further advance the field. A foraging-theoretic exploration can shed light on the mechanisms underlying the human's adaptive interaction with the information presented in the tracing tool. This is precisely the focus of our research.

6.3 Research Methodology

Assisted requirements tracing shares many characteristics with 1) IR-based Web search and 2) navigation along the software entities. As both are domains to which information foraging theory applies (cf. Section 6.2.2), we contend that the analyst's seeking and gathering traceability information can be mathematically modeled in light of the "built-in" foraging mechanisms. In this way, the human analyst can be viewed as a predator in pursuit of prey — the relevant traceability information.

The overall goal of our research is to explore the differences between the analyst's actual information foraging behavior and that defined by the optimality models. Our comparison concentrates on the information diet selection (the diet model) and the residence time within a selected information patch (the patch model). Before formulating specific research questions in Section 6.3.2, we detail how an optimal analyst's decisions are made based on the rational analysis of information foraging.

6.3.1 Rational Analysis

Anderson's rational analysis [5] is built upon the principle of *optimization under constraints*. The basic idea is that the constraints of the environment place important shaping limits on the optimization that is possible [208]. Applied to debugging, the principle implies that optimal programmers will make the best possible navigational choices, given the information the integrated development environment (IDE) like Eclipse makes available to them at each moment [149]. Similarly, the human analyst's optimal behavior in tracing must be rationalized by scrutinizing the information that the automated tool makes available at each moment.

Figure 6.2 shows a screenshot of the automated requirements tracing tool used in our study. We call the tool "ART-Assist" to emphasize the integral yet supportive role it plays in assisted requirements tracing, in which the human analyst must be actively involved [119]. The foremost aim in developing ART-Assist was to obtain functional adequacy of state-of-the-art tracing tools. To that end, we surveyed the basic features of RETRO [121], ADAMS [55], and Poirot [157], and also reviewed our own experience from building the TraCter tool [170]. As different IR-based traceability recovery methods show comparable performance [199], the back-end of ART-Assist adopts the vector space model with TF-IDF weighting [10, 121]. The front-end uses the ordered list to display the retrieved traceability links according to the similarity score computed by the IR algorithm. ART-Assist ranks traceability links much like search engines like Google rank search results in response to a user's query. To support the best practice of *in-place traceability* [47] which advocates tracing-related artifacts being managed within their native environments, each

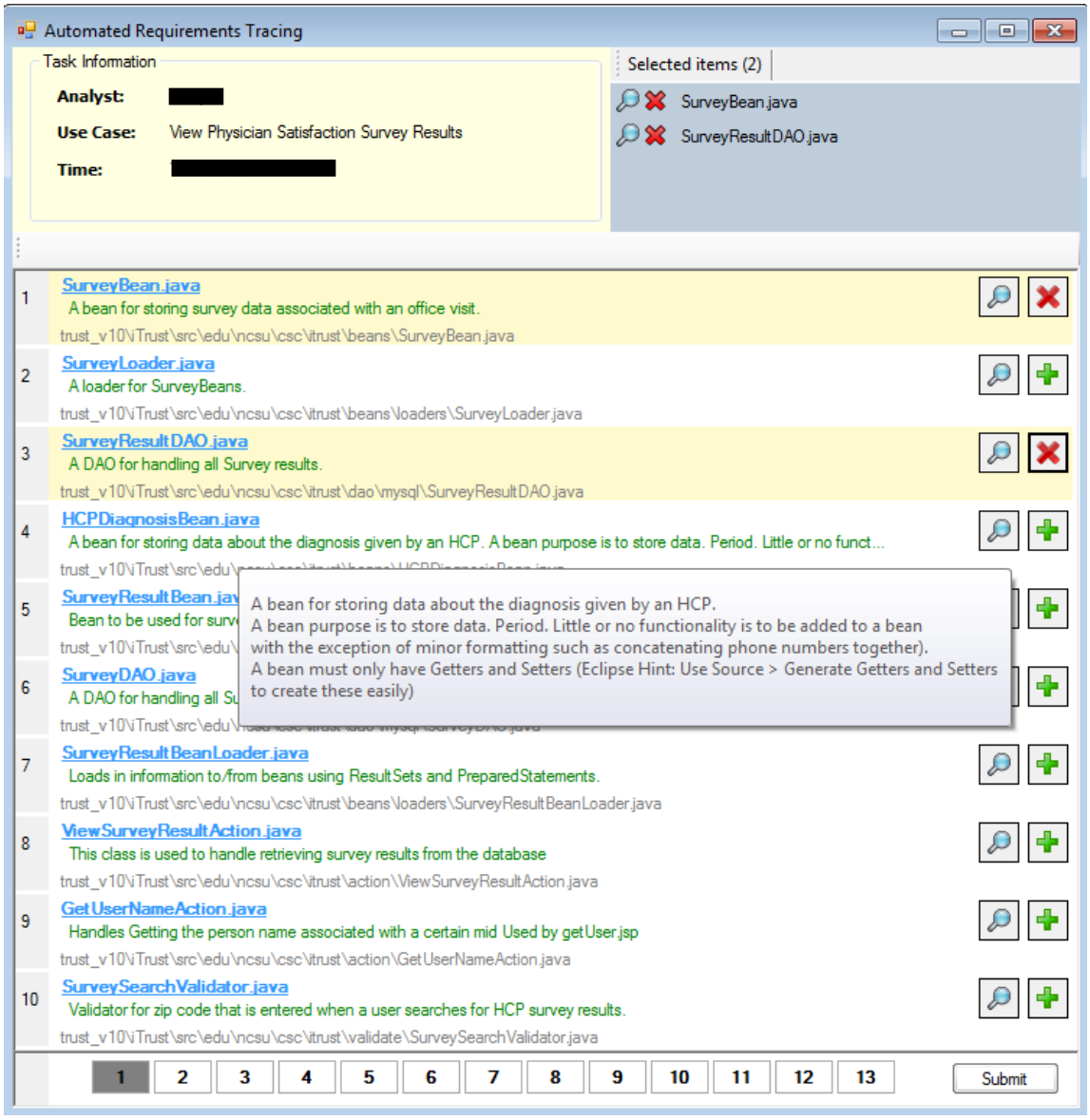


Figure 6.2

A Screenshot of ART-Assist

page of ART-Assist presents 10 retrieved links (or more accurately, the links' snippets) as if they reside in Google's search result page (SRP).

Several design decisions about ART-Assist's information handling are worth discussing. We restrict the discussion to the RTM between requirements-level use cases and implementation-level classes, as this is the granularity level at which our empirical study is conducted.

What information is used for retrieval? ART-Assist extends our indexer [169] to process both source code and use case descriptions. Three steps are carried out: tokenizing, filtering (i.e., removing stop words like `the` and `int`), and stemming (i.e., reducing a word to its inflectional root: "patients" $\rightarrow$ "patient"). The resulting indices, which contain the artifacts' partial and important information, are then used for retrieval.

How much retrieved traceability information to keep? A basic tenet of IR-based methods is that the list of retrieved links contains a higher density of correct traceability links in the upper part of the list and a much lower density of such links in the bottom part of the list [57]. ART-Assist therefore presents the human analyst with 70% of the most similar links, a threshold used in prior work [169].

What information is included in the snippet? The snippet design is informed by our recent study on *topical locality* [195] where we found that class name along with header comments conveyed class body's topic. In addition, the class path offers the file hierarchy information and can act as the URL for the retrieved link. Thus, ART-Assist displays a 3-line snippet in the SRP: the class name, the class header trimmed in 1 line, and the class

path. An innovative design in ART-Assist is that a mouse-hover over a link's snippet pops up the full class header comments, as shown in Figure 6.2.

How to interact with the traceability information? ART-Assist's interaction design philosophy is to fulfill analyst's tracing goal while keeping the operations straightforward, accessible, and responsive. Direct navigation to a certain SRP is enabled by clicking the corresponding page number. The highlighted page number shows which SRP is currently displayed. Clicking the class name in the snippet or the "magnifying glass" icon allows the entire class file to be viewed in a new window. A link can be selected or deselected via "+" (add) or "×" (remove). A shopping-cart-like area in ART-Assist's upper-right corner enables the explicit management of selected links. Once a link is selected, its snippet is yellow highlighted. Once the final RTM is approved, the "submit" button shall be pressed.

Understanding how ART-Assist works is necessary for modeling the way the information environment structures an optimal analyst's foraging. Due to the information needs of the human analyst, navigation in ART-Assist has two fundamental differences from both navigation on the Web and navigation in an IDE such as Eclipse. First, what counts as a hyperlink is well-defined in a website [208] and can be readily modeled by the one-click link built in Eclipse [147, 148]. In contrast, only a single hyperlink type is defined in ART-Assist, namely, the click that enables the viewing of the complete source code file. Second, Web and program navigations can be of great depth because many information items can be reached via clickable hyperlinks. In contrast, the depth of ART-Assist navigations is rather limited because the analyst is primarily interested in viewing and (de-)selecting a traceabil-

ity link. Such differences suggest that the hyperlink topology in a website or a modern IDE is much richer and denser than the one defined in ART-Assist. For this reason, we apply foraging theory's optimality models directly instead of adopting the spreading activation technique used in WUFIS [38] and PFIS [147].

We model each page retrieved by ART-Assist as an *information patch* in which the prey might hide [148]. A patch then contains the SRP and the enclosed *information items* (traceability links) that can be viewed and collected. The key for instantiating "patch", one of foraging theory's core constructs, is to preserve the *locality* such that there are more transitions within a patch than between patches. In Web navigation [208], for instance, a website is treated as an information patch since it is easier to navigate information within the same patch (website) than to navigate information across patches (websites). To assess our patch selection, Fig.6.3 uses a problem behavior graph to visualize an analyst's interaction with ART-Assist when tracing the use case shown in Figure 6.2 ('View Physician Satisfaction Survey Results').

The Problem behavior graph in Figure 6.3 demonstrates the proper mapping of each ART-Assist's page to an information patch. Time in the graph proceeds left to right and top to bottom. Boxes are states. Arrows are moves (transitions). Double vertical lines are returns to a previous state. Dotted enclosing box shows the patch's boundary. The states S_0, SRP_i, and F_j represent the initial state, the i^{th} search result page (SRP), and the information item (traceability link) respectively. Every F_j state is annotated with an ART-Assist icon that indicates the specific operation performed by the human analyst: "magnifying glass" means "view", "+" means "select", and "×" means "deselect". F_j: link (source code file)

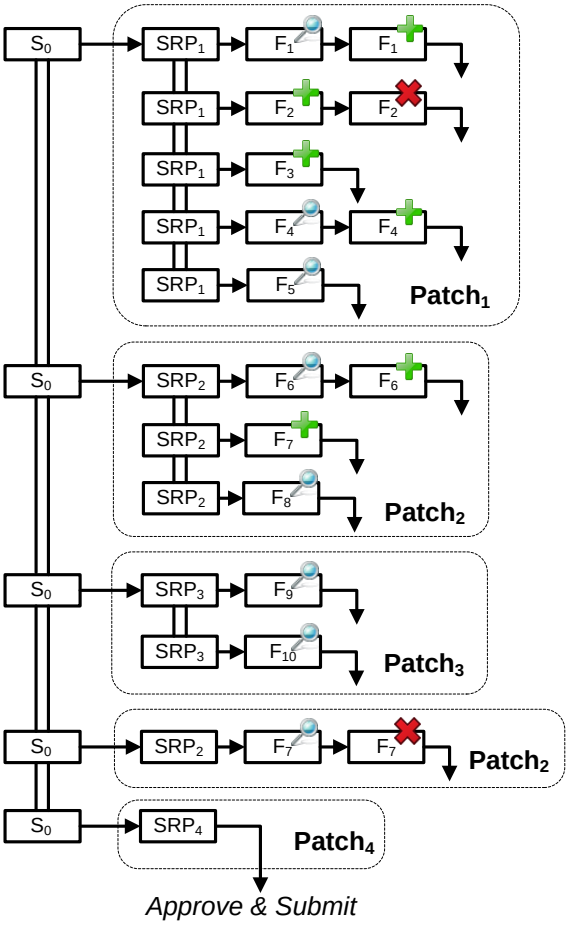


Figure 6.3

Problem Behavior Graph

correspondences are as follows: F_1: SurveyBean.java, F_2: SurveyResultBean.java, F_3: SurveyResultDAO.java, F_4: HCPDiagnosisBean.java, F_5: GetUserNameAction.java, F_6: PersonnelBean.java, F_7: Role.java, F_8: ViewVisitedHCPsAction.java, F_9: PrescriptionReportBean.java, and F_{10}: ViewOfficeVisitAction.java.

A problem behavior graph, which is the foundation for Web behavior graph [208] and code navigation graph [196], is particularly good at showing the *structure* of human's problem solving. Figure 6.3 not only illustrates ART-Assist navigation's limited depth, but also supports patch selection's locality property. Among the total of 36 sessions in our empirical study presented later, the average ratio of within- to between-patch operations (transitions) is 4.2. This ratio is comparable to the values found in Web navigation (ratio=3.7 [208]) and in code navigation (ratio=4.4 [196]).

Figure 6.4 illustrates how to apply the principle of lost opportunity [208] to determine an optimal forager's information diet. For each of the 5 patches given in Figure 6.4-a, the information gain (g) is defined by precision. To characterize the within-patch foraging time (t_W), we leverage MAP (mean average precision), a metric widely accepted in the IR community. MAP measures "the quality across the recall levels" [175]. The formal definition of MAP can be found in the standard IR reference [175] and the traceability-specific literature [237]. Intuitively, the higher the MAP, the closer the correct links are to the top of the information patch and therefore the less within-patch foraging time (t_W) an optimal analyst would need. Thus, we define an information patch's profitability as: $\pi = g/t_W = \text{precision} \cdot \text{MAP}$. The optimal diet can then be selected according to the relationship specified in equation (6.1); here, the encounter rate of all patches is assumed

to be a single unit (i.e., λ=1) as ART-Assist allows each patch to be accessed by a single click on the page number. Figure 6.4-b shows how the theoretical diet (D_Th) is iteratively expanded and optimized in order to counteract the lost opportunity [208].

Patch	**Precision** (g)	**MAP** ($1/t_W$)	**Profitability** (π)
Patch_1	0.40	0.67	0.27
Patch_2	0.30	0.81	0.24
Patch_3	0.20	1.00	0.20
Patch_4	0.20	0.83	0.17
Patch_5	0.10	1.00	0.10

(a)

k	$R(k)$	π_{k+1}	D_Th (optimal diet in theory)
0	–	–	$\{\text{Patch}_1\}$
1	0.16	0.24	$\{\text{Patch}_1, \text{Patch}_2\}$
2	0.19	0.20	$\{\text{Patch}_1, \text{Patch}_2, \text{Patch}_3\}$
3	0.19	0.17	$\{\text{Patch}_1, \text{Patch}_2, \text{Patch}_3\}$

// Following equation (6.1), the optimal diet selection
// terminates because $R(3) > \pi_4$.

(b)

Figure 6.4

(a) Optimal Diet Delection. **(b)** The Optimal Diet (D_Th)

To analyze the optimal residence time within a patch, we apply Charnov's Theorem [208, 235] to examine how much information value the forager acquires over time t:

$$g(t) = \frac{N_R \cdot t}{N_T \cdot t_s + N_R \cdot t_h}. \tag{6.2}$$

In this equation, N_R is the number of relevant information items the forager handles, N_T is the total number of information items encountered, t_s is the scanning time, and t_h is the handling time. We instantiate the value of these parameters based on analysts' actual trac-

ing sessions. This allows for theoretically determining the optimal within-patch residence time (cf. Figure 6.1-c) which we denote by t*_Th.

6.3.2 Research Questions

The research questions addressed in our work are the differences and discrepancies between D_Th (optimal diet in theory) and D_Ac (analyst's actual information diet), and those between t*_Th (optimal within-patch residence time in theory) and t*_Ac (analysts' actual within-patch residence time). Specifically, we are interested in understanding real analysts' deviations and departures from optimality from two complementary perspectives.

- *Structural*. To what extent is an information patch's selection affected by its profitability and quality? While profitability ($\pi = \text{precision} \cdot \text{MAP}$) is computed only if the answer set of true links is known, quality of a patch can be measured by its internal cohesion [69].
- *Behavioral*. To what degree is an information patch's selection affected by the residence time and the navigation behavior of the human analysts?

The answers to the research questions will enable not only a systematic assessment of the factors suggested by information foraging theory, but also a coherent account for unifying the observations that would otherwise not be linked in a meaningful way. For example, with the foraging-theoretic foundation we speculate that (i) the interleave of analyst's correct- and incorrect-decision streaks [138] might be due to the inclusion of both optimal and non-optimal information patches in the actual diet, and (ii) the incorrect decisions after a long foraging period [138] might be attributed to the excessive residence time within a patch.

6.4 Empirical Study Setup

To answer the research questions, we conducted an assisted requirements tracing experiment by using the iTrust dataset (http://agile.csc.ncsu.edu/iTrust). iTrust is a Java application aimed at providing patients with a means to keep up with their medical records, as well as to communicate with their doctors. Although originated as a course project, iTrust has exhibited real-world relevance and served as a traceability testbed for understanding the importance of security and privacy requirements in the healthcare domain [183].

Table 6.1

Requirements Traces Used in the Experiment

ID (our study)	Title (iTrust ID)	|true links|	|true links retrieved|
UC_1	Document Office Visit (UC-11)	26	25
UC_2	Maintain a Hospital Listing (UC-18)	4	4
UC_3	Maintain Standards Lists (UC-15)	13	12
UC_4	Safe Drug Prescription (UC-37)	20	17
UC_5	View Physician Satisfaction Survey Results (UC-25)	8	8
UC_6	View Patients (UC-28)	4	4

The iTrust dataset has 46 use cases (UCs) and 226 Java classes. The requirements-to-source-code traceability matrix is of size 10,396. The answer set, prepared by iTrust's developers, contains 314 true links. Since we wanted to observe analyst's navigation behavior, only the UCs with more than 1 true link defined in the answer set were considered. We identified 32 such UCs, among which 6 were randomly selected. Table 6.1 lists these requirements tracing tasks. Note that ART-Assist keeps 70% of the most similar links in

the retrieval results. Table 6.1 shows that, in some cases, not every true link is presented in ART-Assist.

We recruited 6 upper-division students in computing science from Mississippi State University, including 2 seniors and 4 graduate students. None of the participants knew iTrust before the experiment, but all of them reported being familiar with the healthcare domain. The participants had all learned about traceability and reported a median of 0.25 years tracing experience (mainly done manually).

During the experiment, each participant (analyst) worked alone in a lab and began by signing the consent form and by learning how to use the ART-Assist tool. The analyst was then given hard copies of the UC descriptions and was told to use only ART-Assist and not to use internet or any other resources in the experiment. We asked the analyst to trace all 6 UCs and to carry out the tracing tasks in any order they would prefer. A researcher was present to run the ART-Assist tutorial, to encourage the analyst to think aloud during tracing, and to conduct an informal exit interview to elicit the analyst's feedback about their tracing experience. Each experiment session lasted approximately 1 hour.

6.5 Results and Analysis

ART-Assist logs fine-grained, time-stamped user interactions. In order to extract the analyst's actual diet (D_Ac) from the logs, we analyze the task environment by modeling the problem space of tracing. Figure 6.5 shows the state-transition diagram that depicts the lifecycle of an information item (namely, a traceability link). The analyst may view (scan) the item as it is presented in ART-Assist's retrieval page (patch), and further pursue the

prey (link) if she regards it as relevant. The analyst may refine an item's (de-)selection for several times, during which the item can be viewed optionally. Upon analyst's approval, the item becomes part of the finally submitted RTM.

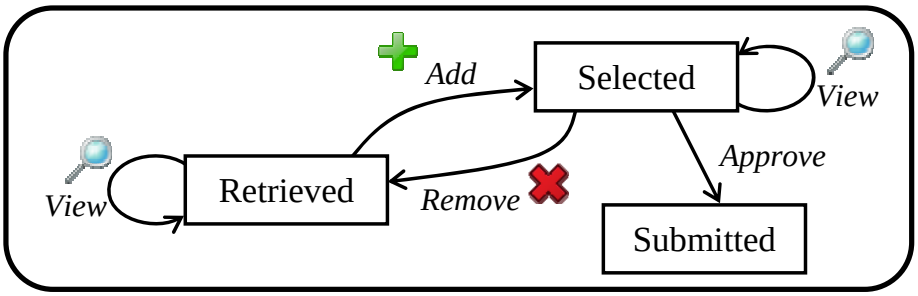


Figure 6.5

State-space of an Information Item (Traceability Link)

Modeling the task environment is critical to defining the variables of rational analysis (cf. Section 6.3.1), especially those appeared in equation (6.2). In our study, t_s (scanning time) denotes the view time for an item that is not selected. In another word, t_s refers to the time the leftmost, self-looped "*View*" transition in Figure 6.5 takes. This figure shows a state-space of an information item (traceability link) as a human analyst with the tracing task interacts with the ART-Assist tool. Boxes show states. Arrows show state transitions annotated with icons representing the ART-Assist operations (cf. Figure 6.2). The other 4 transitions in Figure 6.5 represent operations on what the analyst believes to be a relevant link. Thus, the time transitioned to and from the "Selected" state in Figure 6.5 gives rise to t_h — the handling time. Based on this, we define the analyst's actual diet to be the set of

patches containing handled items, i.e., D_Ac = Patch P | ∃ link $l \in$ P such that 'the analyst handles l as a relevant link at some point during tracing'.

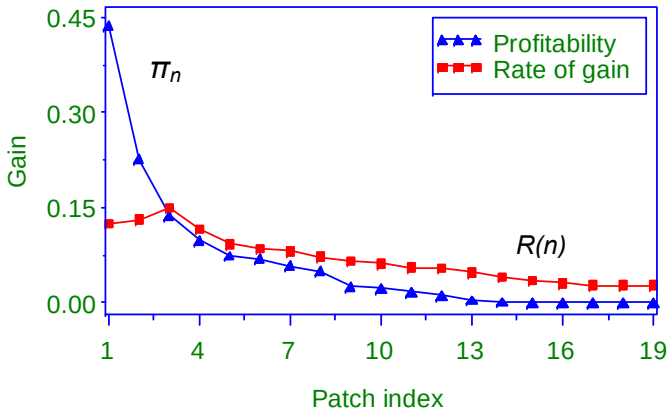


Figure 6.6

Applying the Diet Model (cf. Figure 6.1-b) to the Experimental Tracing Tasks

For the comparison of D_Ac with D_Th, Figure 6.6 shows the average case in which the trace's information environment shapes the optimal diet selection ($R(n)$ and π_n are both averaged over the 6 UCs). Figure 6.6 provides D_Th specific to each UC, along with the total number of patches (pages) retrieved by ART-Assist. On average, only 13.5% of the available patches are included in D_Th. In contrast, D_Ac is much less selective as it is composed of an average of 45.0% available patches. For all 6 tracing tasks, D_Th is a proper subset of D_Ac, which implies that the theory's predictions are highly accurate. To evaluate the matching degree with D_Th, we expand upon the work of Lawrance et al. [146] and use the analysts' consensus to further categorize the patches in D_Ac.

- *Match* refers to the overlap between D_Ac and D_Th. We define the match is *large* if the patch appears in over half of the analysts' actual diets (i.e., $\geq$3 analysts' diets in our study); otherwise, the match is *slight*.

Use Case	**# of patches** (total)	**D_Th** (optimal diet)	**D_Ac** (human analysts' actual information diets)			
			Largely matched with D_Th	Slightly matched with D_Th	Slightly departed from D_Th	Largely departed from D_Th
UC_1	17	$\{P_1, P_2, P_3\}$	$\{P_1\}$	$\{P_2, P_3\}$	$\{P_6, P_8, P_{11}, P_{12}, P_{16}\}$	$\{P_4, P_5\}$
UC_2	19	$\{P_1\}$	$\{P_1\}$	Φ	$\{P_3, P_4, P_{11}, P_{19}\}$	$\{P_2\}$
UC_3	15	$\{P_1, P_2, P_3\}$	$\{P_1, P_2, P_3\}$	Φ	$\{P_7, P_8, P_{13}\}$	$\{P_4, P_6\}$
UC_4	12	$\{P_1, P_2\}$	$\{P_1, P_2\}$	Φ	$\{P_4\}$	$\{P_3\}$
UC_5	13	$\{P_1\}$	$\{P_1\}$	Φ	$\{P_2, P_3, P_4, P_5\}$	Φ
UC_6	14	$\{P_1, P_2\}$	$\{P_1\}$	$\{P_2\}$	$\{P_5, P_6, P_7, P_{13}\}$	$\{P_3, P_4\}$

Figure 6.7

Comparing Optimal Forager's Diet (D_Th) with Real Analysts' Diets (D_Ac)

- *Departure* refers to the difference of D_Ac and D_Th. We say the departure is *large* if the patch is handled by over half of the analysts as they pursue relevant prey in the patch; otherwise, the departure is *slight*.

Table 6.2 shows the structural and behavioral aspects of requirements analyst's information foraging. Descriptive statistics are given in terms of $(\text{mean} \pm \text{standard} - \text{deviation})$. Inferential statistics are performed via the Mann-Whitney test [48], a non-parametric test which was also used by Lawrance et al. [146] for assessing software developer's information foraging. It is evident from Table 6.2 that the matched diets are more profitable than the departed ones. This should not be surprising since the optimal diet (D_Th) is selected based on the descending order of profitability (π). However, the analysts did pursue in a greater number of low-profitable prey-patches than in theoretically high-profitable ones (Mann-Whitney, $U = 32.5$, $p < 0.05$). On one hand, this may account for the quality degradation on the RTM after human vetted the retrieved links [49,62,119,138]. On the other hand, this may suggest that analysts needed to consume the "bad" in order to recognize the "good". In this sense, the correct and incorrect decisions were indeed interdependent.

Table 6.2

Assessing Human Analyst's Information Foraging from Structural and Behavioral Perspectives

D_Ac Categories		**Structural Properties**		**Navigational Behaviors** (all time values are in seconds)				
		π (profitability)	cohesion	# of revisits	t_s	t_h	t*_Ac	Δt* = t*_Ac − t*_Th
Match with D_Th	Large	0.32	0.61	0.45	6.11	4.95	29.50	9.55
	Slight	0.27	0.40	0.33	6.05	6.27	35.29	13.19
Departure from D_Th	Slight	0.09	0.35	1.94	7.97	9.58	90.03	56.07
	Large	0.12	0.53	1.05	7.31	8.08	55.64	48.40

Profitability (π) is computed based on the answer set that defines the true links for each requirement. Under non-experimental settings where no answer set is available, Duan and Cleland-Huang [69] argued that internal metrics, such as coupling and cohesion, could be used to assess the quality of the cluster-patch. We adapt this idea and compute the *patch cohesion* as the average pairwise *TFIDF* differences of all the information items in a given patch. Table 6.2 shows that greater consensus (largely matched and largely departed) was achieved on more cohesive patches. Further comparison reveals that the cohesion of analysts' handled patches (i.e., those in D_Ac) is significantly greater than that of the patches the analysts did not view as relevant (Mann-Whitney, $U = 271.0$, $p < 0.01$). Thus, analysts seem to use cohesion to judge an information patch's relevance; testing this hypothesis requires future research.

As far as the navigation behavior is concerned, the patches matched with D_Th were visited mostly once. However, the D_Th-departed patches received a surprisingly high number of revisits. This shows analysts' struggles with deciding the relevance of certain prey-patches. Even when relevance was determined, the struggles with D_Th-departed patches continued as analysts expended more time handling the links (t_h). Interestingly, the scanning time (t_s) stayed roughly the same across the D_Ac categories.

Our final analysis is concerned with analysts' within-patch residence time (t^*_Ac) and how it differs from the optimal residence time (t^*_Th). As shown in Table 6.2, t*_Ac exhibits considerable variation among D_Ac categories, but in all the cases, the actual residence time is greater than t^*_Th. Such deviations from optimality ($\Delta\ t^*$) were observed to be substantially greater in D_Th-departed diets than in D_Th-matched ones (Mann-Whitney, U=744.5, p<0.01). Figure 6.8 uses the navigation steps to illustrate $\Delta\ t^*$. The value of the $<t_s, t_h>$ pair (cf. equation (6.2)) is instantiated by the average time according to the analysts' actual navigations reported in Table 6.2. To reduce clutter, only one sample $\Delta\ t^*$ (largely matched) is given in Figure 6.8.

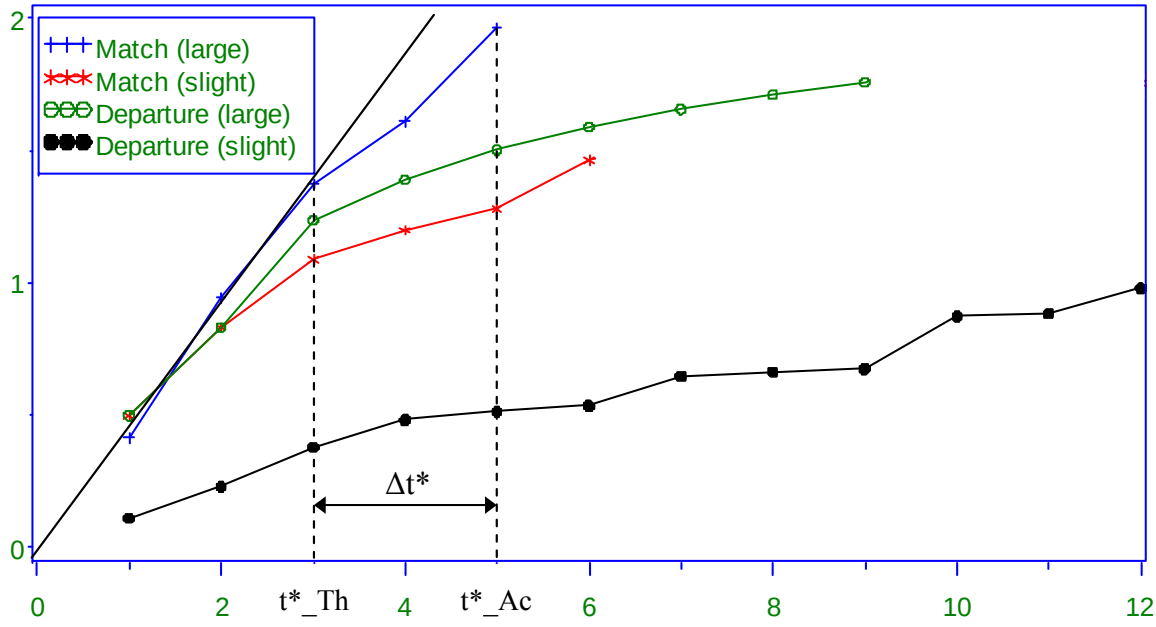


Figure 6.8

Applying the Patch Model to Plot the Average Information Gain Per Navigation Step

6.6 Discussion

As our results indicate that discrepancies exist between human analyst's information seeking and the behavior determined via foraging theory's optimality models, we suggest design guidelines for tools supporting analysts in performing requirements tracing. We then relate our work to other models and discuss the limitations of our study.

6.6.1 Implications for Tool Support

Our objective of uncovering the gaps from optimality is to enable principled ways to reduce the gaps. Our study shows several important discrepancies that provide concrete insights into the behavioral traits of and the obstacles faced by human analysts.

First, although the low-profitable, non-optimal patches (i.e., D_Th-departed patches) turned out to be indispensable for analysts' information diets, the analysts did struggle to determine the relevance of those patches. The primary reason, based on our interviews, was the lack of contextual information when the analysts navigated from one patch (page) to another. In fact, the analysts often unintentionally returned to locations that had already been visited. Most agreed that such revisits were wasted interactions since they had to repeat the relevance judgments. One way to alleviate the struggles is to introduce explicit tagging or rejecting the patch as a whole, as designed in the work of Duan and Cleland-Huang [69]. Another support is to leverage advanced information scent modeling techniques, such as exploiting analyst's navigation recency [148], to generate reactive navigation recommendations.

Second, while the scanning time (t_s) remained approximately constant, the handling time (t_h) was longer for D_Th-departed diets than for D_Th-matched ones. During tracing, more than half of the analysts expressed uncertainty about whether a link should be placed in the shopping-cart for checkout. Some suggested that the gathering of traceability information could be diversified. For example, a black list of irrelevant links and a working area for storing to-be-determined links could be added to the tracing tool.

Third, the analysts invariably overspent the within-patch time (t^*_Ac $> t^*$_Th), especially when foraging in D_Th-departed patches. Enabling analysts to correctly reason about the patch profitability can help them to shorten the time difference (Δt^*). However, it is not possible to expect the analysts to have the perfect knowledge about the information environment. Our study implies that patch cohesion, one of the internal quality indicators, can be of much practical value. In this way, cohesion acts as the *perceived* profitability [146] and its improvement via clustering [69, 195] has already led to remarkable enhancements in tracing.

6.6.2 Relationship to Other Models

In what follows we discuss the relationship between our model and other models.

6.6.2.1 Models of Information Seeking and Gathering

Ko et al. [136] suggested that software maintainers' seeking relevant code follows an iterative 'Search-Relate-Collect' process. While in line with Ko's model, assisted requirements tracing is also related to Web search (e.g., the initiation, selection, and collection stages described by Hearst [114]). Although it is well known that Web users view only the

first few search result pages (e.g., Jansen et al. [124] reported that 80% of the users viewed only the first 2 pages), our results show that human analysts did go as far as the last page to collect the relevant traceability information.

6.6.2.2 Foraging-Theoretic Approaches to Code Navigation

Table 6.3 situates our study within the previous research investigating programmer navigation. Because tracing is a new domain for applying foraging theory, our mapping of an information patch is different from the prior work. The most important difference, in our opinion, is the use of the diet model in our study to determine D_Th, which represents a novel mechanism for assessing D_Ac.

6.6.2.3 Studies of Human Factors in Tracing

Our work complements the studies of analyst's tracing performance based on the final RTM's quality (e.g., [49, 62, 119]). In our study, both the RTM decision and the rational decision-making process are examined. While our work demonstrates the value of optimal foraging models, how to expand the analysis to account for the predictable level of human fallibility [50] and to balance the economics of maintaining and utilizing the requirements traces [73, 74] remains an open question.

6.6.3 Study Limitations

The applicability of this study's results may be limited by ART-Assist's design, operationalization of analyst's actual diet, and participants' unfamilarity with the subject system.

Table 6.3

Comparing Our Work with Other Foraging-theoretic Models

Fundamentals of information foraging theory	**Patches**	**Scent**	**Diet**
Seminal work [146]	Classes	Textual similarity	D_Ac
PFIS [147]	Classes	Textual similarity, Program topology	D_Ac
PFIS2 [148]	Methods	Program topology, Navigation recency	D_Ac
PFIS3 [206]	Methods	Single factors, Optimal composites	D_Ac
Our work	Pages retrieved	Textual similarity	D_Ac, D_Th

ART-Assist provides basic features commonly found in IR-based tracing tools. The 70% threshold filters out certain true links. Adjusting this value, statically or dynamically, may alter the analyst's information foraging behavior.

When defining D_Ac, we adopted a behavioral viewpoint by focusing on *how* to operate a link (cf. Figure 6.5) rather than *what* links were approved in the end. Shifting D_Ac's definition to the final RTM's perspective would modify an important assumption of the decision problem. Once assumptions like this are updated, they can feed back into the rational analysis of the information forager.

Our work with student participants limits how the results could be generalized. Egyed et al. [73] note that in many industrial settings people have no intimate system knowledge during trace recovery. There is also precedence in traceability work: prior studies have used students with low levels of industry experience to represent new people joining a company [49, 62, 73, 138]. Nevertheless, it would be interesting to study how familiarity levels may alter the tracing behavior.

6.7 Conclusions

The main contributions of this chapter are the evolutionary-ecological understanding of the fundamental mechanisms underlying human analysts' requirements tracing behaviors, the theoretical analysis of optimality within the shaping limits placed by tracing's task and information environments, the empirical evaluation of the matches and mismatches between theory's predictions and analysts' actual behaviors, and the concrete insights of the principled ways to increase practical support for software traceability. Building on the extensive research on the IR-based candidate link recovery methods [10, 37, 55, 121, 264], the study of human analysts represents a milestone in the traceability literature, as we now have reached a general consensus regarding the equivalence of the underlying IR methods [199]. The success of requirements tracing, as measured by the final RTM's quality, therefore hinges largely on the analysts' interactions with and decisions about the tool's output. Building on the growing body of work on human factors [49, 62, 119, 138], it is hoped that our work contributes a step towards understanding the ecologically valid ways to "design a fast, accurate and certifiable tracing process" [50].

CHAPTER 7

CONCLUSIONS

This chapter concludes this dissertation, including the main research assumptions, experiments conducted, and final contributions. In addition, we list our main publications in this domain.

7.1 Contributions Summary

We started our analysis by looking at the indexing process. In particular we have tackled the problem of indexing source code for supporting requirements-to-code traceability. We introduced a feature diagram to describe the indexing process, and conducted an experiment using three datasets to examine some of the diagram's features and their dependencies. The results showed that considering comments in the indexing process helped to improve the traceability link quality significantly. Stemming was also found useful when comments were considered. However, if comments were ignored then the overhead of stemming was found to be unnecessary.

In the second phase of our analysis, we investigated the potential benefits of utilizing natural language semantics in automated traceability link retrieval. In particular, we evaluated the performance of a wide spectrum of semantically-enabled IR methods in capturing and presenting requirements traceability links in software systems. Our objectives

were to gain more operational insights into these methods, and to provide practical guidelines for the design and development of effective requirements tracing and management tools. To achieve our research objectives, we conducted an experimental analysis using three datasets from various application domains. Results showed that considering more semantic relations in traceability link retrieval did not necessarily lead to higher quality results. Instead, a more focused semantic support, that targets specific semantic relations, was found to have a greater impact on the overall performance of tracing tools. In addition, our analysis showed that explicit semantic methods, that exploit local or domain-specific sources of knowledge, often achieve a more satisfactory performance than latent methods, or methods that derive semantics from external or general-purpose knowledge sources.

In terms of performance enhancement, we proposed an approach to improving the performance of IR-based automated tracing by examining the cluster hypothesis. The approach was presented through a set of detailed procedures to cluster candidate traceability links, identify low quality clusters, and rearrange links in high quality clusters in such a way that maximizes the browsability. Three open-source datasets from different application domains were investigated to discover optimal settings for these procedures. We further evaluated our approach through a case study to identify the limitations of our approach and the avenues for future research. The study results showed that our approach outperformed the baseline, and had more room for improvement.

In addition, we proposed a novel approach for enhancing the performance of automated tracing tools using refactoring. In particular, we described an experiment for assessing the effect of applying various types of refactorings on the different performance aspects of

IR-based tracing methods. To test our hypothesis, we conducted an experimental analysis using three requirements-to-code datasets from various application domains. Our objective was to assess the impact of various refactoring methods on the performance of automated tracing tools based on information retrieval (IR). Results showed that renaming inconsistently named code identifiers, using RENAME IDENTIFIER refactoring, often leads to improvements in traceability. In contrast, removing code clones, using EXTRACT METHOD refactoring, was found to be detrimental. In addition, results showed that moving misplaced code fragments, using MOVE METHOD refactoring, had no significant impact on trace link retrieval. We further evaluated RENAME IDENTIFIER refactoring by comparing its performance to other strategies often used to overcome the vocabulary mismatch problem in software artifacts. In addition, we proposed and evaluated various techniques to mitigate the negative impact of EXTRACT METHOD refactoring. An effective traceability sign analysis was also conducted to quantify the effect of these refactoring methods on the vocabulary structure of software systems.

Regarding the presentation aspects of the automated tracing process, we leveraged information foraging optimality models to characterize a rational decision process in the domain of automated tracing. Our objective was to offer concrete insights into the obstacles faced by requirements analysts working with IR-based automated tracing tools. our main contributions were the evolutionary-ecological understanding of the fundamental mechanisms underlying human analysts' requirements tracing behaviors, the theoretical analysis of optimality within the shaping limits placed by tracing's task and information environments, the empirical evaluation of the matches and mismatches between theory's

predictions and analysts' actual behaviors, and the concrete insights of the principled ways to increase practical support for software traceability.

Figure 7.1 summarizes our contributions in this dissertation.

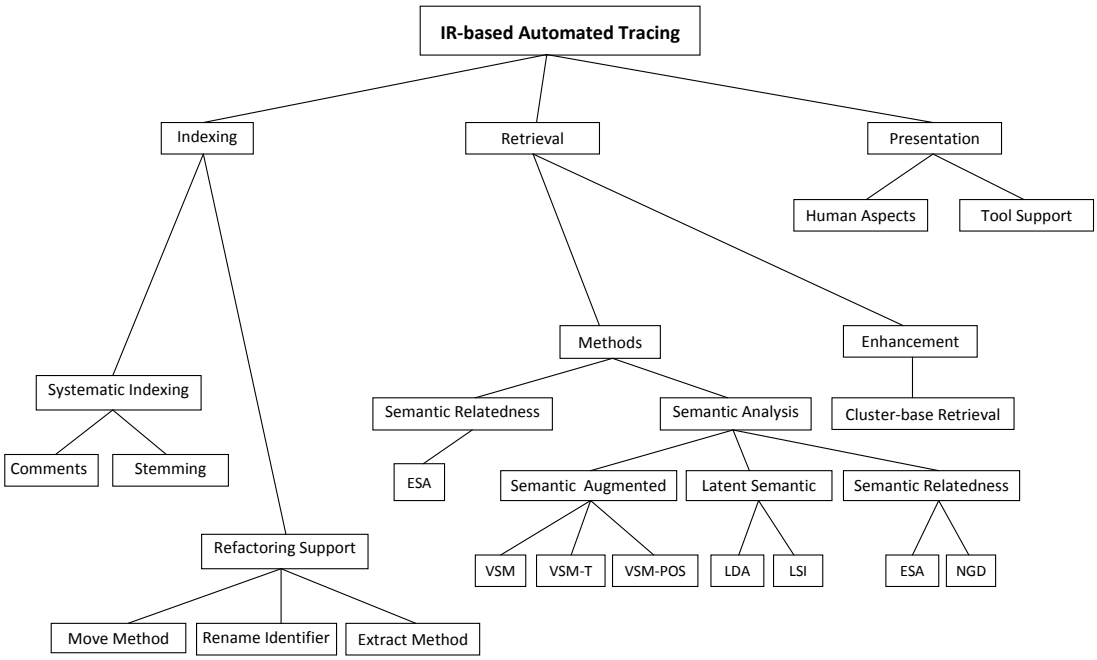


Figure 7.1

Research Contributions

7.2 Publications

The work in this dissertation has resulted in several peer-reviewed journal and conference publications as well as several presentations in various national and international software engineering venues. The following in a list of our main publications in this domain.

1. A. Mahmoud and N. Niu, On the Role of Semantics in Automated Requirements Tracing, Requirements Engineering Journal, (REJ) (accepted).

2. A. Mahmoud and N. Niu, Supporting Requirements to Code Traceability through Refactoring, Requirement Engineering Journal - Special Issue on Best Papers of RE'13 (S.I. REJ) (accepted).

3. A. Mahmoud and N. Niu, Supporting Requirements Traceability Through Refactoring, International Requirements Engineering Conference (RE 2013), pp. 32-41.

4. A. Mahmoud and N. Niu, Source Code Indexing for Automated Tracing, International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE/ICSE 2011), pp. 3-9.

5. A. Mahmoud and N. Niu, Using Semantics-Enabled Information Retrieval in Requirements Tracing: An Ongoing Experimental Investigation, IEEE Computer Software and Applications Conference (COMPSAC 2010), pp. 246-247.

6. A. Mahmoud and N. Niu, A Semantic Relatedness Approach for Traceability Link Recovery, International Conference on Program Comprehension (ICPC 2012), pp.183-192.

7. A. Mahmoud, Toward an Effective Automated Tracing Process: A research Agenda, Student Research Symposium, International Conference on Program Comprehension (ICPC 2012), pp. 269 - 272.

8. N. Niu and A. Mahmoud, Enhancing Candidate Link Generation for Requirements Tracing: The Cluster Hypothesis Revisited, International Requirements Engineering Conference (RE 2012), pp. 81-90.

9. N. Niu, A. Mahmoud, Z. Chen, and G. Bradshaw, Departures from Optimality: Understanding Human Analyst's Information Foraging in Assisted Requirements Tracing, International Conference on Software Engineering (ICSE 2013), pp. 572-581.

10. A. Mahmoud and N. Niu, An Experimental Investigation of Reusable Requirements Retrieval, International Conference on Information Reuse and Integration (IRI 2010), pp. 330 - 335.

11. A. Mahmoud and N. Niu, TraCter: A Tool for Candidate Traceability Link Clustering, International Requirements Engineering Conference, (RE 2011) pp. 335-336.

12. N. Niu, A. Mahmoud, and G. Bradshaw, Information Foraging as a Foundation for Code Navigation, International Conference on Software Engineering (ICSE 2011), pp. 816 - 819.

REFERENCES

[1] A. Abadi, M. Nisenson, and Y. Simionovici, "On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery," *International Conference on Program Comprehension*, 2008, pp. 103–112.

[2] S. Abebe and P. Tonella, "Natural Language Parsing of Program Element Names for Concept Extraction," *International Conference on Program Comprehension*, 2010, pp. 156–159.

[3] D. Advani, Y. Hassoun, and S. Counsell, *Refactoring trends across n versions of n java open source systems: An empirical study*, 2005.

[4] D. Ahn, V. Jijkoun, G. Mishne, K. Mller, M. de Rijke, and S. Schlobach, "Using Wikipedia at the TREC QA Track," *Interactive Poster and Demonstration Sessions*, 2004, pp. 73–76.

[5] J. Anderson, *The Adaptive Character of Thought*, Lawrence Erlbaum Associates, 1990.

[6] J. Anderson and P. Pirolli, "Spread of activation," *Journal of Experimental Psychology*, vol. 10, 1984, pp. 791–798.

[7] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical Debugging Using Latent Topic Models," *European conference on Machine Learning*, 2007, pp. 6–17.

[8] N. Anquetil, C. Fourrier, and T. Lethbridge, "Experiments with Clustering as a Software Remodularization Method," *Working Conference on Reverse Engineering*, 1999, pp. 235–255.

[9] N. Anquetil and T. Lethbridge, "Assessing the Relevance of Identifier Names in a Legacy Software System," *Conference of the Centre for Advanced Studies on Collaborative Research*, 1998, pp. 4–14.

[10] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Transactions on Software Engineering*, vol. 28, 2002, pp. 970–983.

[11] G. Antoniol, G. Canfora, and A. De Lucia, "Maintaining Traceability During Object-Oriented Software Evolution: A Case Study," *International Conference on Software Maintenance*, 1999, pp. 211–219.

[12] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-code traceability for object-oriented systems," *Annals of Software Engineering*, vol. 9, no. 1-4, 2000, pp. 35–58.

[13] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," *International Workshop on Principles of Software Evolution*, 2004, pp. 31–40.

[14] R. Arnold and S. Bohner, "Impact Analysis - Towards a Framework for Comparison," *Conference on Software Maintenance*, 1993, pp. 292–301.

[15] J. Aslam, E. Yilmaz, and V. Pavlu, "A geometric interpretation of r-precision and its correlation with average precision," *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2005, pp. 573–574.

[16] H. Asuncion, A. Asuncion, and R. Taylor, "Software Traceability with Topic Modeling," *International Conference on Software Engineering*, 2010, pp. 95–104.

[17] L. Aversano, L. Cerulo, and M. Di Penta, "How Clones are Maintained: An Empirical Study," *European Conference on Software Maintenance and Reengineering*, 2010, pp. 81–90.

[18] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley, 1999.

[19] B. Baker, "On finding duplication and near-duplication in large software systems," *Working Conference on Reverse Engineering*, 1995, pp. 86–95.

[20] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *International Conference on Software Maintenance*, 1998, pp. 368–377.

[21] E. Ben Charrada, A. Koziolek, and M. Glinz, "Identifying Outdated Requirements Based on Source Code Changes," *International Requirements Engineering Conference*, 2012, pp. 61 –70.

[22] L. Biggers, C. Bocovich, R. Capshaw, B. Eddy, L. Etzkorn, and N. Kraft, "Configuring Latent Dirichlet Allocation based feature location," *Empirical Software Engineering*, 2012, pp. 1–36.

[23] T. Biggerstaff, B. Mitbander, and D. Webster, "Program Understanding and the Concept Assignment Problem," *Communications of ACM*, vol. 37, no. 5, 1994, pp. 72–82.

[24] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," *Working Conference on Mining Software Repositories*, 2011, pp. 203–206.

[25] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Development," *Computer Technologies and Information Sciences*, 2010, chapter 37.

[26] D. Binkley and D. Lawrie, "Maintenance and Evolution: Information Retrieval Applications," *Encyclopedia of software engineering*, P. A. Laplante, ed., 2011, pp. 454–463.

[27] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Science of Computer Programming*, vol. 74, no. 7, 2009, pp. 430–445.

[28] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, 2003, pp. 993–1022.

[29] F. Bourquin and R. Keller, "High-impact Refactoring Based on Architecture Violations," *European Conference on Software Maintenance and Reengineering*, 2007, pp. 149–158.

[30] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, 1983, pp. 543–554.

[31] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwé, "On the Use of Clone Detection for Identifying Crosscutting Concern Coden," *IEEE Transactions on Software Engineering*, vol. 31, 2005, pp. 804–818.

[32] A. Budanitsky and G. Hirst, "Evaluating WordNet-based Measures of Lexical Semantic Relatedness," *Computer Linguistics*, vol. 32, no. 1, 2006, pp. 13–47.

[33] G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, vol. 24, no. 2, 1991, pp. 61–70.

[34] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based Traceability Recovery via Noun-based Indexing of Software Artifacts," *Journal of Software Maintenance and Evolution Research and Practice*, vol. 25, no. 7, 2013, pp. 743–762.

[35] B. Caprile and P. Tonella, "Restructuring Program Identifier Names," *International Conference on Software Maintenance*, 2000, pp. 97–107.

[36] P. Chen and S. Lin, "Automatic keyword prediction using Google similarity distance," *Expert Systems with Applications*, vol. 37, no. 3, 2010, pp. 1928–1938.

[37] X. Chen and J. Grundy, "Improving automated documentation to code traceability by combining retrieval techniques," *Proceedings: 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 223–232.

[38] E. Chi, P. Pirolli, K. Chen, and J. Pitkow, "Using information scent to model user information needs and actions and the Web," *SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 490–497.

[39] A. Chowdhury and M. McCabe, "Improving Information Retrieval Systems using Part of Speech Tagging," *Technical report*, ISR, Institute for Systems Research, 1998.

[40] R. Cilibrasi and P. Vitanyi, "The Google Similarity Distance," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 3, 2007, pp. 370–383.

[41] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering*, vol. 14, no. 1, 2009, pp. 93–130.

[42] J. Cleland-Huang, "Toward Improved Traceability of Non-functional Requirements," *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2005, pp. 14–19.

[43] J. Cleland-Huang, "Traceability research: taking the next steps," *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2011, pp. 1–2.

[44] J. Cleland-Huang, C. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, 2003, pp. 796–810.

[45] J. Cleland-Huang, M. Heimdahl, J. Huffman-Hayes, R. Lutz, and P. Mäder, "Trace queries for safety requirements in high assurance systems," *International Conference on Requirements Engineering: Foundation for Software Quality*, 2012, pp. 179–193.

[46] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability," *International Conference on Requirements Engineering*, 2005, pp. 135–144.

[47] J. Cleland-Huang, R. Settimi, and E. Romanova, "Best Practices for Automated Traceability," *Computer*, vol. 40, no. 6, 2007, pp. 27–35.

[48] Conover, *Practical Nonparametric Statistics*, Wiley, 1999.

[49] D. Cuddeback, A. Dekhtyar, and J. Huffman-Hayes, "Automated Requirements Traceability: The Study of Human Analysts," *International Requirements Engineering Conference*, 2010, pp. 231–240.

[50] D. Cuddeback, A. Dekhtyar, J. Huffman-Hayes, J. Holden, and W. Kong, "Towards overcoming human analyst fallibility in the requirements tracing process (NIER track)," *International Conference on Software Engineering*, 2011, pp. 860–863.

[51] K. David, *Selected papers on computer languages. Center for the Study of Language and Information*, CSLI Lecture Notes, vol 139, 2003.

[52] I. Davidson and S. Ravi, "Intractability and Clustering with Constraints," *International Conference on Machine Learning*, 2007, pp. 201–208.

[53] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichelle, "Using IR methods for labeling source code artifacts: Is it worthwhile?," *International Conference on Program Comprehension*, 2012, pp. 193–202.

[54] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Enhancing an Artefact Management System with Traceability Recovery Features," *International Conference on Software Maintenance*, 2004, pp. 306–315.

[55] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering Methodology*, vol. 16, no. 4, 2007, pp. 13–50.

[56] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental Approach and User Feedbacks: A Silver Bullet for Traceability Recovery," *International Conference on Software Maintenance*, 2006, pp. 299–309.

[57] A. De Lucia, R. Oliveto, and G. Tortora, "IRbased Traceability Recovery Processes: An Empirical Comparison of "One-Shot" and Incremental Processes," *International Conference on Automated Software Engineering*, 2007, pp. 39–48.

[58] A. De Lucia, R. Oliveto, F. Zurolo, and M. Di Penta, "Improving Comprehensibility of Source Code Via Traceability Information: A Controlled Experiment," *International Conference on Program Comprehension*, 2006, pp. 317–326.

[59] A. Dean and D. Voss, *Design and Analysis of Experiments*, Springer, 1999.

[60] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, 1990, pp. 391–407.

[61] F. Deiβenböck and M. Pizka, "Concise and consistent naming," *International Workshop on Program Comprehesnion*, 2005, pp. 97–106.

[62] A. Dekhtyar, O. Dekhtyar, J. Holden, J. Huffman-Hayes, D. Cuddeback, and W. Kong, "On human analyst performance in assisted requirements tracing: Statistical analysis," *IEEE International Requirements Engineering Conference*, 2011, pp. 111–120.

[63] A. Dekhtyar, J. Huffman-Hayes, and G. Antoniol, "Benchmarks for Traceability?," *International Symposium on Grand Challenges in Traceability*, 2007.

[64] J. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices," *Journal on Scientific and Statistical Computing*, vol. 11, no. 5, 1990, pp. 873–912.

[65] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *Journal of Machine Learning Research*, vol. 7, 2006, pp. 1–30.

[66] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," *International Conference on Software Maintenance*, 2005, pp. 389–398.

[67] R. Domges and K. Pohl, "Adapting traceability environments to project-specific needs," *Communications of the ACM*, vol. 41, no. 12, 1998, pp. 54–62.

[68] E. DualaEkoko and M. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Transactions on Software Engineering Methodology*, vol. 20, no. 1, 2010, pp. 1–31.

[69] C. Duan and J. Cleland-Huang, "Clustering Support for Automated Tracing," *International Conference on Automated Software Engineering*, 2007, pp. 244–253.

[70] S. Dumais, "LSI meets TREC: A Status Report," *D. Harman (Ed.), The First Text REtrieval Conference (TREC1), National Institute of Standards and Technology Special Publication*, 1993, pp. 137–152.

[71] M. Edwards and S. Howell, "A Methodology for Systems Requirements Specification and Traceability for Large Real Time Complex Systems," The Institute of Electrical and Electronics Engineers, 1991.

[72] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *IEEE Transactions on Software Engineering*, vol. 9, no. 2, 2003, pp. 116–132.

[73] A. Egyed, F. Graf, and P. Grunbacher, "Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments," *International Requirements Engineering Conference*, 2010, pp. 221–230.

[74] A. Egyed, P. Grnbacher, M. Heindl, and S. Biffl, "Value-Based Requirements Traceability: Lessons Learned," *International Requirements Engineering Conference*, 2007, pp. 115–118.

[75] A. Egyed and P. Grünbacher, "Automating Requirements Traceability: Beyond the Record & Replay Paradigm," *International Conference on Automated Software Engineering*, 2002, pp. 163–171.

[76] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, 1998, pp. 1–12.

[77] L. Etzkorn and C. Davis, "An Approach to Object-oriented Program Understanding," *IEEE Workshop on Program Comprehension*, 1995, pp. 14–15.

[78] D. Evans and C. Zhai, "Noun-phrase Analysis in Unrestricted Text for Information Retrieval," *Annual Meeting on Association for Computational Linguistics*, 1996, pp. 17–24.

[79] J. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic Extraction of a WordNet-Like Identifier Network from Software," *International Conference on Program Comprehension*, 2010, pp. 4–13.

[80] J. Fang and L. Guo, "Calculation of Relatedness by Using Search Results," *International Workshop on Intelligent Systems and Applications*, 2011, pp. 1–4.

[81] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," *International Conference on Program Comprehension*, 2009, pp. 188–197.

[82] C. Fellbaum, *WordNet: An Electronic Lexical Database*, MIT Press, 1998.

[83] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin, "Placing Search in Context: The Concept revisited," *ACM Transactions on Information Systems*, vol. 20, no. 1, 2002, pp. 116–131.

[84] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of Extract Class refactorings in object-oriented systems," *Journal of System and Software*, vol. 85, no. 10, 2012, pp. 2241–2260.

[85] F. Fontanaa, P. Braionea, and M. Zanonia, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, 2011, pp. 1–8.

[86] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, 1999.

[87] C. Fox, "A stop list for general text," *SIGIR Forum*, vol. 24, no. 1, 1989, pp. 19–21.

[88] W. Frakes and B. Nejmeh, "Software Reuse Through Information Retrieval," *SIGIR Forum*, vol. 21, no. 1, 1987, pp. 30–36.

[89] N. Fuhr and C. Buckley, "A Probabilistic Learning Approach for Document Indexing," *ACM Transactions on Information Systems*, vol. 9, no. 3, 1991, pp. 223–248.

[90] G. Furnas, S. Deerwester, S. Dumais, T. Landauer, R. Xarshman, L. Streeter, and K. Lochbaum, "Information Retrieval Using a Singular Value Decomposition Model of Latent Semantic Structure," *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1988, pp. 465–480.

[91] M. Fyson and C. Boldyreff, "Using Application Understanding to Support Impact Analysis," *Journal of Software Maintenance: Research and Practice*, vol. 10, no. 2, 1998, pp. 93–110.

[92] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using Wikipedia-based explicit semantic analysis," *International Joint Conference on Artifical Intelligence*, 2007, pp. 1606–1611.

[93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[94] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," *International Conference on Automated Software Engineering*, 2010, pp. 245–254.

[95] R. Gligorov, Z. Aleksovski, W. Kate, and F. Harmelen, "Using google distance to weight approximate ontology matches," *International Conference on World Wide Web*, 2007, pp. 767–776.

[96] O. Gotel, J. Cleland-Huang, J. Huffman-Hayes, A. Zisman, A. Egyed, P. Grünbacher, and G. Antoniol, "The quest for Ubiquity: A roadmap for software and systems traceability research," *International Conference on Requirements Engineering*, 2012, pp. 71–80.

[97] O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Intrenational Conference on Requirements Engineering*, 1994, pp. 94–101.

[98] O. Gotel and A. Finkelstein, "Contribution Structures," *International Symposium on Requirements Engineering*, 1995, pp. 100–107.

[99] O. Gotel and S. Morris, "Out of the Labyrinth: Leveraging Other Disciplines for Requirements Traceability," *IEEE International Requirements Engineering Conference*, 2011, pp. 121–130.

[100] J. Gracia, R. Trillo, M. Espinoza, and E. Mena, "Querying the web: a multiontology disambiguation method," *International Conference on Web Engineering*, 2006, pp. 241–248.

[101] S. Grant and J. Cordy, "Estimating the optimal number of latent concepts in source code analysis," *International Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 65–74.

[102] S. Greenspan and C. McGowan, "Structuring Software Development For Reliability," *Microelectronics Reliability*, vol. 17, no. 1, 1987, pp. 75–83.

[103] T. Griffiths and M. Steyvers, "Finding scientific topics," *The National Academy of Sciences*, 2004, pp. 5228–5235.

[104] A. Grzywaczewski and R. Iqbal, "Task-specific information retrieval systems for software engineers," *Journal of Computer and System Sciences*, vol. 78, no. 4, 2012, pp. 1204–1218.

[105] L. Guerrouj, "Normalizing source code vocabulary to support program comprehension and software quality," *International Conference on Software Engineering*, 2013, pp. 1385–1388.

[106] J. Guo and Y. Wang, "Towards consistent evolution of feature models," *international conference on Software product lines: going beyond*, 2010, pp. 451–455.

[107] Y. guo, M. Yang, J. Wang, P. Yang, and F. Li, "An Ontology-based Approach for Traceability Recovery," *International Symposium on Knowledge Acquisition and Modeling*, 2009, pp. 160–163.

[108] S. Gupta, J. Hartkopf, and S. Ramaswamy, "Event Notifier: A Pattern of Event Notification," *Java Report*, vol. 3, no. 7, 2000, pp. 131–154.

[109] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," *Working Conference on Reverse Engineering*, 2010, pp. 35–44.

[110] V. Hamilton and M. Beeby, "Improving Comprehensibility of Source Code Via Traceability Information: A Controlled Experiment," *IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, 1991, pp. 1–3.

[111] E. Han and G. Karypis, "Centroid-Based Document Classification: Analysis and Experimental Results," *European Conference on Principles of Data Mining and Knowledge Discovery*, 2000, pp. 424–431.

[112] M. Hata, F. Homae, and H. Hagiwara, "Semantic relatedness between words in each individual brain: An event-related potential study," *Neuroscience Letters*, vol. 501, no. 2, 2009, pp. 72–77.

[113] T. Hazen, "Direct and latent modeling techniques for computing spoken document similarity," *Spoken Language Technology Workshop*, 2010, pp. 366–371.

[114] M. Hearst, *Search User Interfaces*, Cambridge University Press, 2009.

[115] M. Hearst and J. Pedersen, "Reexamining the cluster hypothesis: scatter/gather on retrieval results," *international ACM SIGIR conference on Research and development in information retrieval*, 1996, pp. 76–84.

[116] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *International Conference on Software Maintenance*, 2012, pp. 837–847.

[117] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?," *International Conference on Software Maintenance*, 2012, pp. 243–252.

[118] J. Huffman-Hayes and A. Dekhtyar, "A framework for Comparing Requirements Tracing Experiments," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 05, 2005, pp. 751–781.

[119] J. Huffman-Hayes and A. Dekhtyar, "Humans in the traceability loop: can't live with 'em, can't live without 'em," *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2005, pp. 20–23.

[120] J. Huffman-Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval," *International Conference on Requirements Engineering*, 2003, pp. 138–147.

[121] J. Huffman-Hayes, A. Dekhtyar, and S. Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, 2006, pp. 4–19.

[122] IEEE, "IEEE-830. Guide to Software Requirements Specification,", 1984.

[123] IEEE, "IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries,", 1991.

[124] B. Jansen, A. Spink, J. Bateman, and T. Saracevic, "Real life information retrieval: a study of user queries on the Web," *ACM SIGIR Forum*, vol. 32, no. 1, 1998, pp. 5–17.

[125] J. Jeng and B. Cheng, "Specification Matching for Software Reuse: A Foundation," *SIGSOFT Software Engineering Notes*, vol. 20, no. SI, 1995, pp. 97–105.

[126] K. Jones, "Automatic summarising: The state of the art," *Information Processing and Management*, vol. 43, no. 6, 2007, pp. 1449–1481.

[127] D. Jurafsky and J. Martin, *Speech and language processing*, Prentice Hall, 2000.

[128] H. Kagdi, *Mining software repositories to support software evolution*, Doctoral thesis, Kent State University, 2008.

[129] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions Software Engineering*, vol. 28, no. 7, 2002, pp. 654–670.

[130] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, Carnegie Mellon University, 1990.

[131] M. Katić and K. Fertalj, "Towards an Appropriate Software Refactoring Tool Support," *WSEAS International Conference on Applied Computer Science*, 2009, pp. 140–145.

[132] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley & Sons Ltd., 1990.

[133] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Asia-Pacific Software Engineering Conference*, 2004, pp. 184–193.

[134] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[135] L. Kit, C. Man, and E. Baniassad, "On finding duplication and near-duplication in large software systems," *Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006, pp. 383–396.

[136] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions Software Engineering*, vol. 32, no. 12, 1998, pp. 971–987.

[137] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study: Practice Articles," *Journal of Software Maintinance and Evolution*, vol. 18, no. 2, 2006, pp. 109–132.

[138] W. Kong, J. Huffman-Hayes, A. Dekhtyar, and J. Holden, "How do we trace requirements: An initial study of analyst behavior in trace validation tasks," *International Workshop on Cooperative and Human Aspects of Software Engineering*, 2011, pp. 32–39.

[139] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *Working Conference on Reverse Engineering*, 2006, pp. 253–262.

[140] G. Kowalski, *Information Retrieval Architecture and Algorithms*, Springer, 2010.

[141] R. Krovetz, "Viewing morphology as an inference process," *International ACM SIGIR conference on Research and development in information retrieval*, 1993, pp. 191–202.

[142] C. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, 1992, pp. 131–183.

[143] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information Software Technolgy*, vol. 49, no. 3, 2007, pp. 230–243.

[144] K. Laitinen, "Estimating understandability of software documents," *SIGSOFT Software Engineering Notes*, vol. 21, no. 4, 1996, pp. 81–92.

[145] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons Ltd., 1985.

[146] J. Lawrance, R. Bellamy, and M. Burnett, "Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance," *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, pp. 15–22.

[147] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," *SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 1323–1332.

[148] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart, "Reactive information foraging for evolving goals," *SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 25–34.

[149] J. Lawrance, M. Burnett, R. Bellamy, K. Rector, and S. Fleming, "How Programmers Debug, Revisited: An Information Foraging Theory Perspective," *IEEE Transactions on oftware Engineering*, vol. 39, no. 2, 2010, pp. 197–215.

[150] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing Source Code Vocabulary," *Working Conference on Reverse Engineering*, 2010, pp. 3–12.

[151] D. Lawrie, H. Feild, and D. Binkley, "Extracting Meaning from Abbreviated Identifiers," *International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 213–222.

[152] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, 2007, pp. 303–318.

[153] M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-program Life Cycle," *Journal of System and Software*, vol. 1, no. 3, 1984, pp. 213–221.

[154] T. Lethbridge, J. Singer, and A. Forward, "How Software Engineers Use Documentation: The State of the Practice," *IEEE Software*, vol. 20, no. 6, 2003, pp. 35–39.

[155] S. Letovsky, "Cognitive Processes in Program Comprehension," *workshop on empirical studies of programmers on Empirical studies of programmers*, 2011, pp. 58–79.

[156] A. Leuski, "Evaluating Document Clustering for Interactive Information Retrieval," *International Conference on World Wide Web*, 2001, pp. 33–40.

[157] J. Lin, C. C. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O.Khadra, C. Duan, and X. Zou, "Poirot: A Distributed Tool Supporting Enterprise-Wide Automated Traceability," *International Conference on Requirements Engineering*, 2006, pp. 363–364.

[158] M. Lindvall and K. Sandahl, "Traceability Aspects of Impact Analysis in Object-oriented Systems," *Journal of Software Maintenance*, vol. 10, no. 1, 1998, pp. 37–57.

[159] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," *International Conference on Automated Software Engineering*, 2007, pp. 461–464.

[160] C. Lioma and R. Blanco, "Part of Speech Based Term Weighting for Information Retrieval," *Advances in Information Retrieval*, 2009, pp. 412–423.

[161] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modelling class cohesion as mixtures of latent topics," *International Conference on Software Maintenance*, 2009, pp. 233–242.

[162] D. Lonngren, "Reducing the Cost of Test Through Reuse," *IEEE Systems Readiness Technology Conference*, 1998, pp. 48–53.

[163] M. Lormans, "Can LSI Help Reconstructing Requirements Traceability in Design and Test," *European Confirence on Software Maintenance and Reengineering*, 2006, pp. 47–56.

[164] M. Luisa, F. Mariangela, and I. Pierluigi, "Market research for requirements analysis using linguistic tools," *Requirements Engineering*, vol. 9, no. 1, 2004, pp. 40–56.

[165] S. Lukins, N. Kraft, and L. Etzkorn, "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation," *Working Conference on Reverse Engineering*, 2008, pp. 155–164.

[166] Y. Maarek, D. Berry, and G. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991, pp. 800–813.

[167] P. Mäder, O. Gotel, and I. Philippow, "Rule-based maintenance of post-requirements traceability relations," *International Requirements Engineering Conference*, 2008, pp. 23–32.

[168] A. Mahmoud and N. Niu, "An Experimental Investigation of Reusable Requirements Retrieval," *International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering*, 2010, pp. 330–335.

[169] A. Mahmoud and N. Niu, "Source Code Indexing for Automated Tracing," *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2011, pp. 3–9.

[170] A. Mahmoud and N. Niu, "TraCter: A Tool for Candidate Traceability Link Clustering," *International Requirements Engineering Conference*, 2011, pp. 335–336.

[171] A. Mahmoud and N. Niu, "Supporting Requirements Traceability through Refactoring," *International Requirements Engineering Conference*, 2013.

[172] A. Mahmoud, N. Niu, and S. Xu, "A semantic Relatedness Approach for Traceability Link Recovery," *International Conference on Program Comprehension*, 2012, pp. 183–192.

[173] J. Mai, "Analysis in indexing: Document and domain centered approaches," *Information Processing & Management*, vol. 41, no. 3, 2005, pp. 599–611.

[174] J. Maletic and A. Marcus, "Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding," *International Conference on Tools with Artificial Intelligence*, 2000, pp. 46–53.

[175] C. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[176] M. Mäntylä and C. Lassenius, "Drivers for Software Refactoring Decisions," *International Symposium on Empirical Software Engineering*, 2006, pp. 297–306.

[177] A. Marcus and J. Maletic, "Identification of High-Level Concept Clones in Source Code," *International Conference on Automated Software Engineering*, 2001, pp. 107–114.

[178] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," *International Conference on Software Engineering*, 2003, pp. 125–135.

[179] A. Marcus and D. Poshyvanyk, "The Conceptual Cohesion of Classes," *International Conference on Software Maintenance*, 2005, pp. 133–142.

[180] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using Latent Dirichlet Allocation," *India software engineering conference*, 2008, pp. 113–120.

[181] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *International Conference on Software Maintenance*, 1996, pp. 244–253.

[182] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving Usability of Software Refactoring Tools," *Australian Software Engineering Conference*, 2007, pp. 307–318.

[183] A. Meneely, B. Smith, and L. Williams, *iTrust Electronic Health Care System: A Case Study*, chapter Software and Systems Traceability, Springer, 2012.

[184] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, 2004, pp. 126–139.

[185] D. Milne and I. Witten, "An Effective, Low-Cost Measure of Semantic Relatedness Obtained from Wikipedia Links," *AAAI Workshop on Wikipedia and Artificial Intelligence*, 2008, pp. 25–30.

[186] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," *International Conference on Reuse of Off-the-Shelf Components*, 2006, pp. 287–297.

[187] G. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," *IEEE Software*, vol. 23, no. 4, 2006, pp. 76–83.

[188] G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, 2001, pp. 364–380.

[189] E. Murphy-Hill and A. P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," *International Conference on Software Engineeringl*, 2008, pp. 421–430.

[190] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor and how we know it," *International Conference on Software Engineering*, 2009, pp. 287–297.

[191] L. Murta, A. Hoek, and C. Werner, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links," *International Conference on Automated Software Engineering*, 2006, pp. 135–144.

[192] R. Nallapati, W. Cohen, and J. Lafferty, "Parallelized Variational EM for Latent Dirichlet Allocation: An Experimental Evaluation of Speed and Scalability," *International Conference on Data Mining Workshops*, 2007, pp. 349–354.

[193] NASA, "Preferred Reliability Practices: Independent Verification and Validation of Embedded Software,", Practice No. PD-ED-1228, Marshal Space Flight Centre, 1999.

[194] N. Niu and S. Easterbrook, "Extracting and Modeling Product Line Functional Requirements," *International Requirements Engineering Conference*, 2008, pp. 155–164.

[195] N. Niu and A. Mahmoud, "Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited," *IEEE International Requirements Engineering Conference*, 2012, pp. 81–90.

[196] N. Niu, A. Mahmoud, and G. Bradshaw, "Information foraging as a foundation for code navigation (NIER track)," *International Conference on Software Engineering*, 2011, pp. 816–819.

[197] N. Niu, A. Mahmoud, Z. Chen, and G. Bradshaw, "Departures from Optimality: Understanding Human Analysts Information Foraging in Assisted Requirements Tracing," *International Conference on Software Engineering*, 2013.

[198] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," *Conference on The Future of Software Engineering*, 2000, pp. 35–46.

[199] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," *International Conference on Program Comprehension*, 2010, pp. 68–71.

[200] W. Opdyke, *Refactoring Object-Oriented Frameworks*, Doctoral thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[201] W. Opdyke and R. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[202] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," *International Conference on Software Engineering*, 2013, pp. 522–531.

[203] S. Patwardhan, S. Banerjee, and T. Pedersen, "SenseRelate::TargetWord: a generalized framework for word sense disambiguation," *Interactive poster and demonstration sessions*, 2005, pp. 73–76.

[204] J. Pedersen, C. Silverstein, and C. Vogt, "Verity at TREC-6: out-of-the-box and beyond," *Information Processing and Management*, vol. 36, no. 1, 2000, pp. 187–204.

[205] N. Pennington, "Comprehension Strategies in Programming," *Empirical Studies of Programmers: Second Workshop*, 1987, pp. 100–113.

[206] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2011, pp. 109–116.

[207] P. Pirolli, "Computational models of information scent-following in a very large browsable text collection," *ACM SIGCHI Conference on Human factors in computing systems*, 1997, pp. 3–10.

[208] P. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information*, Oxford University Press, 2007.

[209] K. Pohl, "PRO-ART: Enabling Requirements Pre-Traceability," *International Conference on Requirements Engineering*, 1996, pp. 76–84.

[210] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast Collapsed Gibbs Sampling for Latent Dirichlet Allocation," *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008, pp. 569–577.

[211] F. Porter, *An algorithm for suffix stripping*, Morgan Kaufmann Publishers Inc., 1997, pp. 313–316.

[212] D. Poshyvanyk, "Using information retrieval to support software maintenance tasks," *International Conference on Software Maintenance*, 2009, pp. 453–456.

[213] D. Poshyvanyk, Y. gal Guhneuc, and A. Marcus, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, 2007, pp. 420–432.

[214] M. Pucher, "WordNet-based Semantic Relatedness Measures in Automatic Speech Recognition for Meetings," *Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, 2007, pp. 129–132.

[215] B. Ramesh and M. Jarke, "Towards Reference Models for Requirements Traceability," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, 2001, pp. 58–93.

[216] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Lessons Learned from Implementing Requirements Traceability," *Crosstalk – Journal of Defense Software Engineering*, vol. 8, no. 4, 1995, pp. 11–15.

[217] A. Rao, A. Lu, E. Meier, S. Ahmed, and D. Pliske, "Query processing in TREC-6," *Information Processing and Management*, vol. 36, no. 1, 2000, pp. 179–186.

[218] B. Rosario, *Latent semantic indexing: An overview*, INFOSYS 240 Spring Paper, University of California, Berkeley, 2000.

[219] C. Roy and J. Cordy, "A Survey on Software Clone Detection Research," *Technical Report 541, School of Computing TR 2007-541, Queens University*, 2007.

[220] C. Roy and J. Cordy, "An Empirical Study of Function Clones in Open Source Software," *Working Conference on Reverse Engineering*, 2008, pp. 81–90.

[221] G. Salton and M. Lesk, "Computer Evaluation of Indexing and Text Processing," *Journal of ACM*, vol. 15, no. 1, 1968, pp. 8–36.

[222] G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing," *Communications of ACM*, vol. 18, no. 11, 1975, pp. 613–620.

[223] R. Sarukkai, *Foundations of Web Technology*, The Springer International Series in Engineering and Computer Science, 2002, pp. 106–108.

[224] H. Schmid, "Probabilistic part-of-speech tagging using decision trees," *International Conference on New Methods in Language Processing*, 1994, pp. 44–49.

[225] H. Schutze, "Dimensions of meaning," *Supercomputing*, 1992, pp. 787–796.

[226] R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, and C. DePalma, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts," *International Workshop on the Principles of Software Evolution*, 2004, pp. 49–54.

[227] U. Shardanand and P. Maes, "Social information filtering: algorithms for automating word of mouth," *SIGCHI Conference on Human Factors in Computing Systems*, 1995, pp. 210–217.

[228] D. Shepherd, Z. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," *International Conference on Aspect-Oriented Software Development*, 2007, pp. 212–224.

[229] H. Sneed, "Object-oriented COBOL recycling," *Working Conference on Reverse Engineering*, 1996, pp. 169–178.

[230] P. Sorg and P. Cimiano, "Exploiting Wikipedia for cross-lingual and multilingual information retrieval," *Data and Knowledge Engineering*, vol. 74, no. 0, 2012, pp. 26–45.

[231] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," *Handbook of Software Engineering and Knowledge Engineering*, vol. 3, 2004, pp. 395–428.

[232] G. Spanoudakisa, A. Zismana, E. E. Pérez-Miñana, and P. Krausec, "Rule-based Generation of Requirements Traceability Relations," *Journal of Systems and Software*, vol. 72, no. 2, 2004, pp. 105–127.

[233] J. Spool, C. Perfetti, and D. Brittan, *Designing for the Scent of Information*, User Interface Engineering, 2004.

[234] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," *International Conference on Automated Software Engineering*, 2010, pp. 43–52.

[235] D. Stephens and J. Krebs, *Foraging Theory*, Princeton University Press, 1986.

[236] M. Strube and S. Ponzetto, "WikiRelate! computing semantic relatedness using wikipedia," *National Conference on Artificial Intelligence*, 2006, pp. 1419–1424.

[237] H. Sultanov, J. Huffman-Hayes, and W. Kong, "Application of Swarm Techniques to Requirements Tracing," *Requirments Engineering Journal*, vol. 16, no. 3, 2011, pp. 209–226.

[238] S. Sundaram, J. Huffman-Hayes, A. Dekhtyar, and E. Holbrook, "Assessing Traceability of Software Engineering Artifacts," *Requirements Engineering Journal*, vol. 15, no. 3, 2010, pp. 313–335.

[239] A. Takang, P. Grubb, and R. Macredie, "The effects of comments and identifier names on program comprehensibility: An experimental investigation," *Journal of Programming Languages*, vol. 4, no. 3, 1996, pp. 143–167.

[240] Y. Teh, M. Jordan, M. Beal, and D. Blei, "Hierarchical Dirichlet processes," *Journal of the American Statistical Association*, vol. 101, no. 476, 2006, pp. 1566–1581.

[241] S. Teufel, "An Overview of Evaluation Methods in TREC Ad Hoc Information Retrieval and TREC Question Answerin," *Evaluation of Text and Speech Systems*, L. Dybkjaer, H. Hemsen, and W. Minker, eds., 2007, pp. 163–186.

[242] A. Thies and C. Roth, "Recommending rename refactorings," *International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 1–5.

[243] S. Thomas, B. Adams, A. Hassan, and D. Blostein, "Validating the Use of Topic Models for Software Evolution," *IEEE Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 55–64.

[244] T. Tourwé and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," *European Conference on Software Maintenance and Reengineering*, 2003, pp. 91–100.

[245] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, 2009, pp. 347–367.

[246] G. Tsatsaronis, I. Varlamis, and M. Vazirgiannis, "Text relatedness based on a word thesaurus," *Communication of ACM*, vol. 37, no. 1, 2010, pp. 1–40.

[247] S. Ugurel, R. Krovetz, C. Lee Giles, D. Pennock, E. Glover, and H. Zha, "Whats the code? automatic classification of source code archives," *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 632–638.

[248] B. Vinz and L. Etzkorn, "Improving program comprehension by combining code understanding with comment understanding," *KnowledgeBased Systems*, vol. 21, no. 8, 2008, pp. 813–825.

[249] A. von Knethen, "Automatic Change Support Based on a Trace Model," *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.

[250] A. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek, "Systematic Requirements Recycling through Abstraction and Traceability," *International Conference on Requirements Engineering*, 2002, pp. 273–281.

[251] S. Wahid, "Investigating Design Knowledge Reuse for Interface Development," *Conference on Designing Interactive Systems*, 2006, pp. 354–356.

[252] X. Wang, G. Lai, and C. Liu, "Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering," *Electronic Notes in Theoretical Computer Science*, vol. 243, 2009, pp. 121–137.

[253] X. Wei and B. Croft, "LDA-based document models for ad-hoc retrieval," *ACM SIGIR conference on Research and development in information retrieval*, 2006, pp. 178–185.

[254] Z. Wen and V. Tzerpos, "An Optimal Algorithm for MoJo Distance," *IEEE International Workshop on Program Comprehension*, 2003, pp. 227–235.

[255] Z. Wen and V. Tzerpos, "Software Clustering Based on Omnipresent Object Detection," *International Workshop on Program Comprehension*, 2005, pp. 269–278.

[256] D. Wilking, U. Kahn, and S. Kowalewski, "An Empirical Evaluation of Refactoring," *e-Informatica Software Engineering Journal*, vol. 1, no. 1, 2007, pp. 44–60.

[257] P. Willett, "Recent trends in hierarchic document clustering: a critical review," *Information Processing and Management*, vol. 24, no. 5, 1988, pp. 577–597.

[258] S. Wong, W. Ziarko, V. Raghavan, and P. Wong, "On modeling of information retrieval concepts in vector spaces," *ACM Transactions Database Systtems*, 2012, pp. 299–321.

[259] J. Yi and F. Maghoul, "Query Clustering Using Click-through Graph," *International Conference on World Wide Web*, 2009, pp. 1055–1056.

[260] R. Yin, *Case Study Research: Design and Methods*, Sage Publications, 2003.

[261] O. Zamir and O. Etzioni, "Grouper: a dynamic clustering interface to Web search results," *International Conference on World Wide Web*, 1999, pp. 1361–1374.

[262] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja, "Mr. LDA: a flexible large scale topic modeling package using variational inference in MapReduce," *international conference on WWW*, 2012, pp. 879–888.

[263] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "An Ontology-based Approach for Traceability Recovery," *International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering*, 2006, pp. 36–43.

[264] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: a study of term-based enhancement methods," *Empirical Software Engineering*, vol. 15, no. 2, 2007, pp. 119–146.