Tracing Domain Data Concepts in Layered Applications

Mohammed Daubal

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2014

Committee:

Hazeline U. Asuncion

Brent Lagesse

Munehiro Fukuda

Mark Kochanski

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

Tracing Domain Data Concepts in Layered Applications

Mohammed Daubal

Chair of the Supervisory Committee:

Assistant Professor Hazeline Asuncion, Ph.D.

Computing & Software Systems

A goal of reusing source code is to lower development cost. Existing reuse techniques usually require additional costs in creating generic source code and in retrieving relevant source code from a code repository. This Master's thesis presents a novel traceability technique that facilitates reuse in layered applications in a cost-effective way, called the Domain Data Concept (DDC) Tracer. While current traceability techniques focus on establishing links between software artifacts, DDC Tracer is focused on tracing concepts between software and data, across different layers of an application, and across heterogeneous implementation files. The DDC Tracer has been evaluated using various techniques: industry case studies from Aramco, an experiment comparing DDC Tracer tool with related techniques, a user study with industry software engineers, and a feature comparison. The results of this Master's thesis indicate that the DDC Tracer is a lightweight alternative to source code reuse techniques, is feasible to use in practice, and is more effective than existing information retrieval (IR) tools in locating related source code for a given set of DDCs that implement a feature in a software system.

Acknowledgement

First of all, I would like to express my gratitude to my advisor Prof. Hazeline Asuncion for giving me the opportunity to work in her research group and for her support, patience, and guidance throughout this Master's thesis. As a result, I gained much knowledge in software traceability.

Working with her is similar to working in a real team environment where I feel that she is one of my colleagues. I like the way that she precedes to tackle a research problem by simplifying the problem and treating each challenge individually and gradually to deliver a complete solution.

I also would like to thank all the members of our research software traceability group, especially Nathan Duncan and Delmar Davis for working with me in this Master's research.

I would like to thank my thesis committee: Dr. Brent Lagesse, Dr. Munehiro Fukuda and Dr. Mark Kochanski for their review and feedback.

I wish to thank all department staff, especially, Megan Jewell for her advising throughout my Master degree program. Also, my special thanks goes to Laurie Anderson for all her help with the writing and presentation of my Master's thesis.

I am very grateful to Aramco and its management for giving me this opportunity to participate in the advanced degree program and for all of their support. I would like to express my gratitude and appreciation to my colleague Hussein AL-Helal, who took care of my requests to the company to complete my Master's research. Also, I wish to thank the software engineers for their feedback on the tool.

Finally, I would like to thank my family: my parents, my wife, and my sons for their patience and for all their daily support and encouragement.

This work is based in part upon work supported by the US National Science Foundation under Grant No. CCF 1218266 and ACI 1350724 and the Aramco Advanced Degree Program Scholarship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or Aramco.

Table of Contents

CHAPTER 1 INTRODUCTION 1

1.1 Current Approach for Source Code Reuse 1

1.1.1 Black-Box Reuse 1

1.1.2 White-Box Reuse 2

1.1.3 Limitations of Current Tools to Support White-Box Reuse 3

1.2 Research Focus 4

1.3 Motivation 4

1.4 Goal and Criteria 5

1.5 Approach Overview and Contributions 6

1.6 Outline 7

CHAPTER 2 RELATED WORK 8

2.1 Background on Data-Centric Layered Applications 8

2.1.1 Client-Server Architecture Overview 8

2.1.2 Client-Server Architecture Exemplar 9

2.1.2.1 Database 10

2.1.2.2 Server 13

2.1.2.3 Client 14

2.2 Basic Underlying Techniques 14

2.2.1 Textual Analysis 14

2.2.2 Topic Modeling 16
2.2.3 Static Analysis 18
2.2.4 Dynamic Analysis 19

2.3 Research Areas 19

2.3.1 Software Traceability 21
2.3.2 Software Reuse 22
2.3.3 Feature Location 22
2.3.4 Code Search and Code Clone 25
2.3.5 Database Analysis 25
2.3.6 Recommender Systems 25

2.4 Summary 26

CHAPTER 3 TECHNIQUE: DOMAIN DATA CONCEPT TRACER 27

3.1 Overview 27

3.2 Step 1: Determine the Starting Point for Tracing 29

3.3 Step 2: Create Mappers at Layer Boundaries 30

3.3.1 Back-End to Application Server 30
3.3.2 Server-Side to Client-Side 31

3.4 Step 3: Create Mappers within Layer Boundaries 32

3.4.1 Within Application Server 33
3.4.2 Within Client-Side 34

3.5 Step 4: Connect Mappers to Create a Traceability Link Chain 37

3.6 Step 5: Traceability Chain Metrics 38

3.6.1 SQL Statements Similarity 39

3.6.2 SQL Statements Complexity 40

3.7 Step 6: Provide Search Capability 41

CHAPTER 4 TOOL SUPPORT 43

4.1 Architecture Overview 43

4.1 Implementation Technologies 44

4.2 Implementation Detail 45

4.2.1 Back-End 45

4.2.1.1 DDCs mapping data extractor 46

4.2.1.2 Static analysis 47

4.2.1.3 Topic modeling 47

4.2.2 Server 48

4.2.2.1 DDC search 48

4.2.2.2 Call graph retriever 49

4.2.2.3 File retriever 49

4.2.2.4 Mapper connector 50

4.2.3 Client 50

4.3 Implementation Challenges 51

4.3.1 Challenges with Pre-Processing the XML Files 51

4.3.2 Challenges with Pre-Processing the SQL Statements 52

4.3.3 Other Challenges 53

4.4 Usage Scenarios 53

CHAPTER 5 EVALUATION 56

5.1 Evaluation Goals and Methods Overview 56

5.2 Case Study on Aramco Software Projects 57

5.2.1 Aramco Dataset Overview 58

5.2.2 Find Relevant Server-Side Code for a Given Set of DDCs 59

5.2.3 Find Relevant Client-Side Code for a Given Set of DDCs 60

5.2.4 Find a Complete Relevant Traceability Chain for a Given Set of DDCs 62

5.3 DDC Tracer Correctness Experiment 64

5.3.1 Experiment Setup 65

5.3.2 Correctness Measurement 65

5.3.2.1 Google desktop search (IR) 66

5.3.2.2 Topic modeling 66

5.3.3 Discussion 67

5.4 User Feedback 68

5.4.1 Survey Question and Feedback 68

5.4.2 Discussion 70

5.5 Feature Comparison 70

CHAPTER 6 CONCLUSION 72

6.1 Implications 72

6.2 Limitations 73

6.3 Future Work.....73

List of Figures

Figure 1: Client-server application [8].....9

Figure 2: Three-tier client-server architecture 10

Figure 3: Tracing DDCs across various implementation files.....12

Figure 4: Feature location (a) versus Domain Data Concept traceability (b).....24

Figure 5: Domain Data Concept Tracer (our technique) [8].....29

Figure 6: Controller as a hub on the client-side.....34

Figure 7: Traceability links within client using topic modeling [47] 36

Figure 8: DDC Tracer architecture 44

Figure 9: DDC Tracer extracting mapping information (bottom database and files are in the back-end).....45

Figure 10: DDC Tracer tool.....55

Figure 11: Correctness metric.....62

List of Tables

Table 1: Comparison of related research areas with our DDC Tracer technique 20

Table 2: SQL complexity metrics used in DDC Tracer tool adapted from [57] 41

Table 3: Evaluation methods and goals 56

Table 4: Statistics of Aramco software projects used in the evaluation 59

Table 5: Find relevant Java files to given DDCs[8] 60

Table 6: P & R in recovering the related model & views for a given controller 61

Table 7: Summary of P & R UI traceability links recovering 62

Table 8: Tool correctness for project No.1 63

Table 9: Tool correctness for project No.3 63

Table 10: Tool correctness for project No.4 64

Table 11: DDC Tracer tool correctness vs. topic modeling and Google IR tool 68

Table 12: DDC Tracer tool rating [8] 69

Table 13: Feature Comparison [8] 71

CHAPTER 1 INTRODUCTION

Literature has shown that software reusability [21] [42] lowers development costs by reducing development effort and time which also results to an increase in software productivity. In general, within a particular business domain, software systems are often variants of existing systems, in which they share similar features to satisfy different user needs [62]. As shown in [42], there were 60% to 70% of common functionalities among the studied software systems within a particular domain. Therefore, most of the different software artifacts produced during the development, such as design structures, documentation, and source code, can be reused to build a new software system instead of creating them from scratch [42]. Since source code is the generally available artifact, many studies on software reusability focus on source code reuse to lower development costs [33].

This chapter presents the current approaches for source code reuse and their challenges, the focus, motivation, goal and criteria, overview of the approach, and contributions of the thesis.

1.1 Current Approach for Source Code Reuse

Source code reuse can be categorized into two main classes: black-box and white-box reuse.

1.1.1 Black-Box Reuse

Black-box source code reuse focuses on a systematic approach, such as creating a reusable library of components, using domain engineering (e.g., model driven development, product line engineering), or using software architecture, where software artifacts are created according to the software reuse plan within the organization [21] [33].

However, the ability of black-box reuse techniques remains challenging based on the return on investment. Indeed, their success is determined by the economic benefit to the organization as whether there is a good return on investment on the upfront costs for reuse

strategy [21] [42]. These costs include determining, building, and managing reusable artifacts and educating developers on how to reuse the artifacts [42]. Another challenge is the ability of the black-box reusable code to be applied to different contexts of a given task [32]. In fact, reused code requires frequent modifications to be deployed in a new context [31].

1.1.2 White-Box Reuse

White-box reuse is another approach for source code reuse, in which developers search for source code that can be reused from existing systems. This approach does not require large upfront costs; instead, it allows a developer to extract and modify the existing code to fit a new context for the given development task. Thus, a developer can search, extract, and modify the code from existing software and reuse it to implement a given task in another system [40]. This approach of reusing the source code that is not designed for reuse is referred to as code scavenging, ad hoc reuse, opportunistic reuse, copy-paste reuse, and recently pragmatic reuse [32].

The literature has shown that white-box source code reuse is an effective and a common approach in industry [32] [33]. For instance, a survey conducted at NASA showed that 47% of the reused code was performed using a white box technique, even though the organization had planned reuse program (i.e., black-box reuse) [32]. Moreover, another study found 85% of developers' reuse activities involved copy-paste and modify code to create new Java classes [32].

The white-box source code reuse has a major drawback. When a bug is fixed in the original code, this change must be reflected in all places where the same source code was reused [32]. There are currently no techniques to automate this process [32].

White-box source code reuse is not an easy task. Before code can be reused, a developer must search the entire software code repository and identify all files related to a specific

feature. Once the implementation files are found, a developer can understand the context and the potential for reusability of the source code. This is a time consuming, tedious, and error-prone process, especially in a collection of large software projects [32] [33]. Manual search requires collecting huge information, such as location of source code and its dependencies to make the right decision for code reuse [32], which is another challenge for a new developer or someone who is unfamiliar with the source code [26]. Consequently, manually identifying related source code may not be successful because files may be missed or incorrectly identified.

1.1.3 Limitations of Current Tools to Support White-Box Reuse

A suitable search tool is a prerequisite for source code reuse in white-box manner. Code search tools have been used to facilitate code reuse by enabling developers to search by textual similarity, program structure, or code semantics [8].These techniques do not support reuse for applications that access and manipulate data itself.

Moreover, while identifying related source code has traditionally fallen within the purview of software dependency analysis [25], contemporary development of layered applications often use third-party technologies that combine source code with configuration files and scripts, resulting in a heterogeneous mix of implementation files (e.g.,[16] [24] [46]). Therefore, traditional static and dependency analysis tools are unable to trace through these implementation files.

As a result, textual search or dependency analysis does not yield all the files that operate on specified Domain Data Concepts (DDCs) because of the various technologies used. Thus, current techniques are inadequate to determine the connections between related files in an implemented system.

1.2 Research Focus

This Master's thesis focuses on white-box reuse of source code for data-centric applications that use layered software architecture and are implemented using a heterogeneous mix of technologies[1].

1.3 Motivation

In large corporations, software development is often distributed among different teams and explicit collaboration among these independent teams is usually difficult [15]. Software developers may encounter situations where a new development task is similar to previously developed software for a different project. Reusing those implementation files from existing systems remains a challenging task because it depends on the developer's memory, communication channels among developers, and the ability to extract the source code. To further illustrate the applicability of this research to software development, this section now describes challenges in reusing software within the context of large corporations.

Communication among employees is essential to support source code reuse, in order for developers to know what others have implemented [42]. As a result, teams are not aware of source code they can reuse from other teams, resulting in redevelopment of software from scratch with similar functions. Developers try to reuse code by manually searching for existing code that was developed by that developer, a developer's colleague, or by the developer's team. This time consuming task does not always result in a successful retrieval of reusable code.

[1] Portions of this work is published in the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)

In addition, software development projects in the context of large corporations operate on data that is often stored in back-end databases [28]. Usually, a common thread among these independent projects is its access to a centralized database, which contains information that is a core business asset to the company. For example, these databases may contain data about a company's customers. Thus, accessing the same database (e.g., schema) leads to have the same or similar features among different software systems [63] that can be reused.

1.4 Goal and Criteria

This Master's research aims to address the gap of tracing heterogeneous implementation files in data-centric applications to support code reuse.

To achieve this goal, this Master's thesis presents the Domain Data Concept Tracer (DDC Tracer), a technique that automates tracing Domain Data Concept in layered architecture systems. DDC Tracer can be used to identify all implementation files from the database to the user interface that operates on a given Domain Data Concept. In order for the DDC Tracer to successfully support software reuse, it must satisfy the following criteria:

- **C1: Correctness of results**: The technique should produce a higher percentage of correct results over incorrect results. This means that the results of the DDC Tracer should provide the same or better results than current techniques. The meaning of correctness may depend on a given reuse task.
- **C2: Light-weight technique:** This technique should require minimal training time and usage time from the developers. That is, the DDC Tracer should take less time for developers to learn and use compared to existing techniques or processes.
- **C3: Utility in reusing implementation files:** The tool should be able to locate relevant source code files that operate on a given set of DDCs across the software

layers, including data access, business logic, and user interface. In addition, it should be able to provide a recommendation for relevant identified source code.

1.5 Approach Overview and Contributions

The DDC Tracer connects data concepts with implementation files in layered architecture to create a searchable traceability link chain. By centering traceability links on DDCs and leveraging the knowledge of how DDCs are related to each other (via a database schema) and how source code is structured (e.g., communication with a framework, caller graph), we can create a traceability link chain from DDCs to implementation files, regardless of the textual similarity of DDCs to source code, and regardless of the technologies used for the implementation files (e.g., source code, configuration files, SQL). This approach also enables software developers to search for highly relevant implementation files for reuse.

Specific contributions of this thesis are:

- A data-centric approach for tracing Domain Data Concepts in layered software architecture.
- A prototype tool support that automates the technique.
- A technique for creating a complete traceability link chain from the server code (back-end) to client code (front-end).
- A lightweight technique to facilitate source code reuse.
- A set of metrics for determining the relevance of traceability link chains.
- A set of evaluations based on an industrial case study, experiment, feature comparison, and user study.

1.6 Outline

This Master's thesis is organized as follows. The next chapter covers the background about the layered software architecture style and related techniques from various research areas that address similar problems. Chapter 3 discusses the DDC Tracer approach and Chapter 4 presents tool support. The evaluation methods are discussed in Chapter 5. The research concludes with limitations of DDC Tracer and future work.

CHAPTER 2 RELATED WORK

This chapter provides background on data-centric applications that follows a layered architecture style to demonstrate the challenges in tracing Domain Data Concepts across different software layers. This chapter also provides background on fundamental techniques that have been used in our DDC Tracer and that are commonly used by different research areas tackling similar problems. Finally, this chapter discusses related research areas.

2.1 Background on Data-Centric Layered Applications

This section provides an overview of layered architecture software system followed by detailed exemplar as a case study. This exemplar demonstrates the addressed problem of constructing the complete traceability link chain that connects the heterogeneous implementation files together.

2.1.1 Client-Server Architecture Overview

Software that follows the layered architectural style has source code organized in different layers and each layer uses the services provided by an adjacent layer [59]. A client-server style is a special case of the layered architectural style wherein the number of layers is limited to two or three layers (see Figure 1). This architecture style is popularly used in business applications wherein the logic and data model are centralized [59]. The big boxes in Figure 1 represent components while the small boxes represent connectors that provide communication between components.

Often, client-server architecture systems are web-based applications designed in a layered architecture that consists of a heterogeneous mix of technologies: data manipulation or extraction (e.g., SQL query statements), business logic (e.g., Java, C++), and client-side logic

(e.g., JavaScript, HTML) [39]. To add to this mixture, companies may use frameworks (e.g. [46]) to lower development and maintenance costs and to standardize the code base styles.

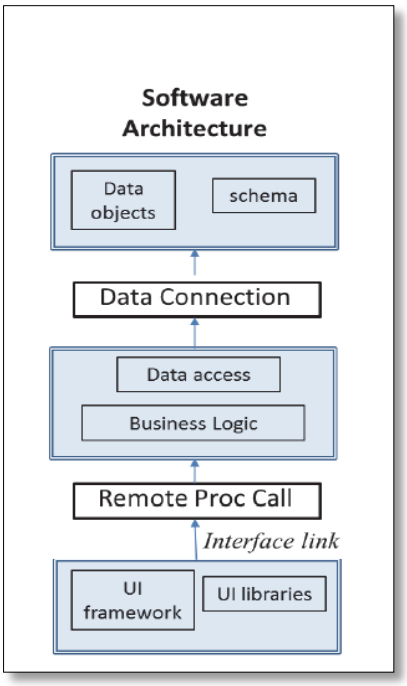

Figure 1: Client-server application [8]

2.1.2 Client-Server Architecture Exemplar

To better illustrate the challenges with tracing across heterogeneous implementation files, this section describes an exemplar client-server system (see Figure 2). This exemplar is a three-tiered web application that uses various third-party technologies.

The database holds the core business information that represents Domain Data Concepts (e.g., schema concepts). A rich interactive interface runs on the client while the server executes the business task that is hosted in a separate machine from the database. Details regarding each layer are as follows:

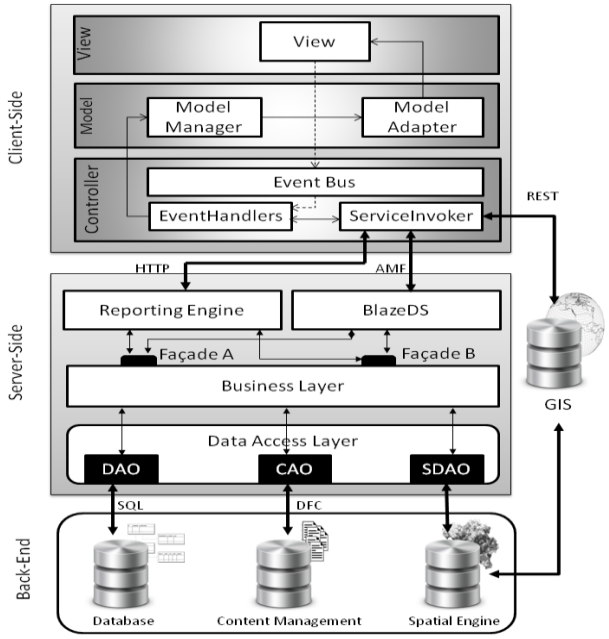

Figure 2: Three-tier client-server architecture [2]

2.1.2.1 Database

A database is a collection of interrelated data that represents some concepts from the real world [12]. The definitions of these concepts are provided by a data model such as relational, object-oriented, and object-relational. The relationship between real-world concepts are represented in a schema [16] (See the database box in Figure 3).

In a relational database, a data concept is stored in a table (e.g., employee table) and the attributes of this concept are stored as columns in the table (e.g., EmployeeName, EmploymentStartDate, EmployeeClassification) [7]. Related data concepts are specified through table relationships. One way this relationship is achieved is through the use of primary and foreign keys. A primary key is used to uniquely identify each row in a table. In our Employee table example, EmployeeID may be used to uniquely identify each tuple or row. A foreign key is used to refer to a primary key in another table. For example, any

[2] Al-Helal Hussein, in discussion with the author, 2012.

number of employees may work within a department. This relationship between the Department concept and Employee concept may be expressed through the use of EmployeeID as a foreign key within the Department table.

Structured Query Language (SQL), originally called Structured English Query Language (SEQUEL), identifies a set of simple operations on tabular structures [6]. These operations include data lookup (SELECT), data update (UPDATE), data insert (INSERT), and data delete (DELETE). For instance, the SELECT operation consists of two required clauses: SELECT and FROM. The FROM clause specifies the table in the database from which to obtain results, while the SELECT clause specifies the attributes or columns in the table to be retrieved. For example, if we wish to list all the employee names, we can state "SELECT EmployeeName FROM Employee ". Optionally, the WHERE clause can be used to specify the conditions of the rows to be returned. For instance, if we wish to only see the names of employees whose salary is greater than $8000, we can use the following statement "SELECT EmployeeName FROM Employee WHERE Salary >'8000' ". From this statement, we can extract the data concept Employee and its attributes EmployeeName and Salary.

Other SQL operations also contain data concepts. The UPDATE and INSERT operations specify the table name and attributes to update or insert. The DELETE operation specifies the table name or the specific records in the table to delete. In the latter case, table attributes are also specified.

Thus, these DDCs (e.g., table names and attributes) in SQL statements establish the base for creating a traceability chain in layered applications.

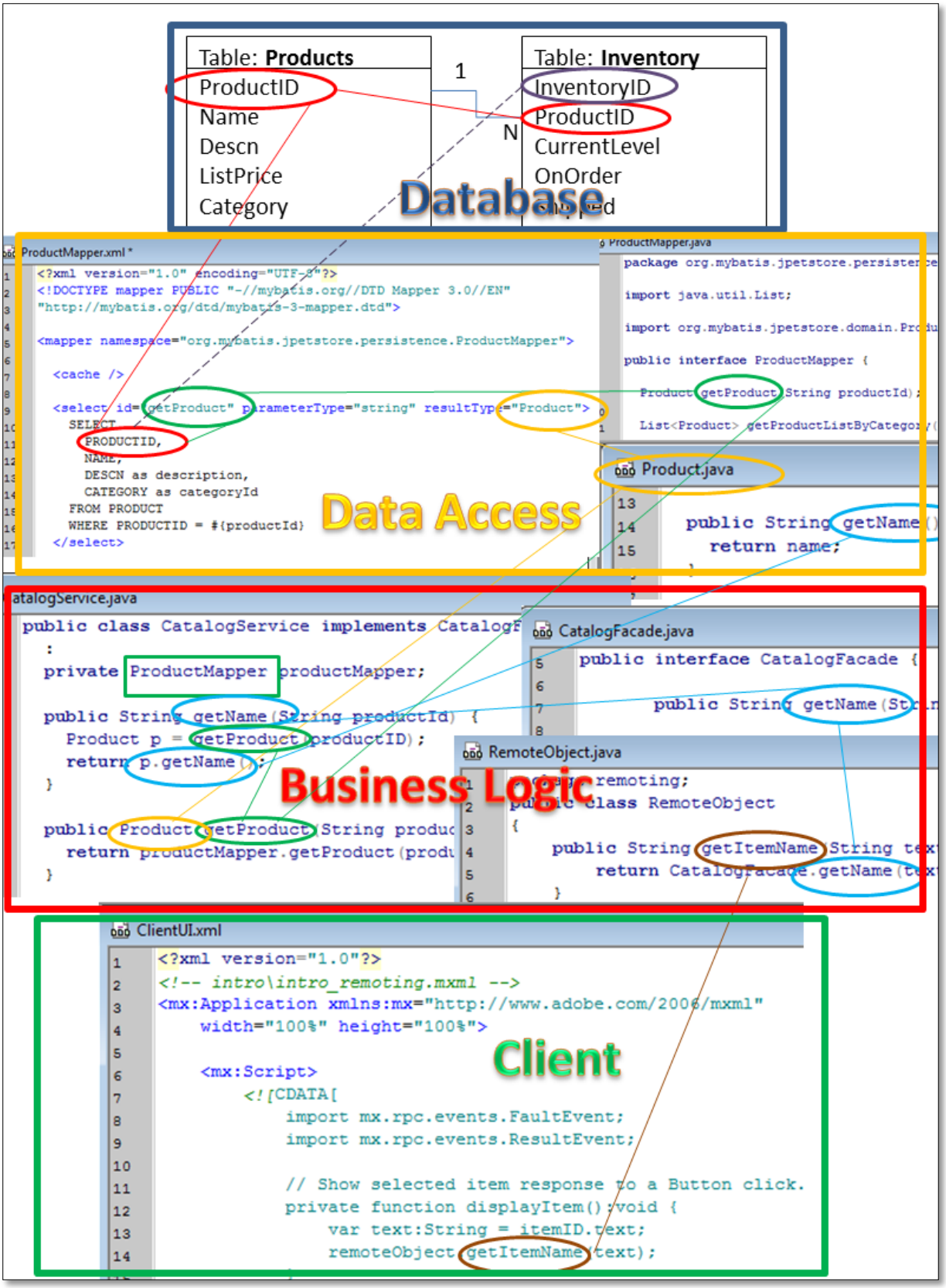

Figure 3: Tracing DDCs across various implementation files

2.1.2.2 Server

The server tier consists of two components:

The Data Access component provides access to the back-end (e.g., database) that is implemented using mapper framework, such as iBatis/myBatis [46], to translate relation data into objects (see Product.java in Figure 3). This framework encapsulates the SQL statements in an XML mapper file (see ProductMapper.xml in Figure 3) that contains a unique SQL ID and optional configuration parameters (e.g., parameter type and result type). A corresponding mapper source code calls the SQL statement using the unique ID (see ProductMapper.java in Figure 3).

The Business Logic component hosts application-specific business logic. This layer is implemented using Java (see CatalogService.java, CatalogFacade.java, RemoteObject.java in Figure 3) and Façade design pattern that defines the exposed functionality to the client through Java Remote Method Invocation [49]. This façade design pattern reduces client dependence on the implementation details and simplifies client-server interactions, so all Façade source code files are exposed for remote access by the client code. For example, a client developed by Adobe Flex framework accesses remote server objects (see ClientUI and RemoteObject.java in Figure 3) through BlazeDS using the Action Message Format (AMF) over HTTP [34]. AMF is a binary format used to increase the performance in moving large amounts of data. This BlazeDS is integrated with the Spring framework that provides an inversion of control (IoC) container responsible for managing the component's lifecycle on the server-side. It provides the required resources through injection (e.g., inject databases source code to the business logic) [24] so that clients can invoke these resources remotely.

2.1.2.3 Client

The client-side is designed based on the model-view-controller (MVC) [60] pattern. The views are written using Adobe's Flex [34] and specify user interface controls via an XML configuration file (see ClientUI in Figure 3). Adobe Flex provides rich web-applications similar to desktop applications. Also, it uses frameworks to implement the MVC design pattern at the client layer, such as Mate [23] and Swiz [19]. These frameworks are based on events, driven by tags, and use the inversion of control mechanism. Thus, the views render the model data and dispatch the events to invoke the remote methods on the business logic layer. The controller handles these events to communicate with the server.

As shown in this detailed exemplar, Domain Data Concepts can be traced across different layers from the database, to business logic, and to the user interface. Because we are tracing based on data concepts, we can connect heterogeneous implementation files, as shown in Figure 3: ProductMapper.xml (**SQL**) → ProductMapper.java (**DA**) → CatalogService.java (**BL**) → CatalogFacade.java (**BL**) → RemoteObject.java (**BL**) → ClientUI.xml (**UI**) (i.e., Controllers and other views and models).

2.2 Basic Underlying Techniques

The most common techniques used to identify the relationships (i.e., recover traceability links) between source code files are textual analysis, topic modeling, static analysis, and dynamic analysis.

This section describes each technology and its limitations in identifying relevant documents to a given set of DDCs.

2.2.1 Textual Analysis

Textual analysis is a class of techniques that uses textual similarities between documents and the user's query's words to retrieve and rank the relevant documents [11] [55]. It analyzes

the words that represent the domain knowledge expressed in natural language in the software artifacts (e.g., requirements, user manual, reports, and source code) [2]. Textual analysis has been applied widely in identification of the traceability links automatically among heterogeneous text software artifacts [2] [3] [5] [17]. For instance, in the source code, a developer may write comments and use meaningful names of the classes, methods, variables, etc. that represent the domain knowledge to add semantic information to aid understandability [2] [11] [39]. This textual data is derived from the source code (i.e., identifiers and comments) and pre-processed [11] [17] [39]. This process involves removing the stop words and stemming the words to recover traceability links between source code files, assuming files are relevant based on the idea that they use equivalent words in the software system [11] [17] [39].

Simple pattern matching between a user query and words in the code identifiers and comments may not achieve the desired results because of the vocabulary problem [11]; developers may use inconsistent words throughout the system because the same word may have different meanings or different words may have the same meaning [13]. Furthermore, developers often use abbreviations and incomplete sentences for naming the program identifiers [13]. As a result, information retrieval (IR), an advanced textual analysis technique, is used to overcome the simple text matching to identify the relationship between the query and documents.

There are several textual analysis techniques from information retrieval research areas, including Vector Space Model (VSM) and latent Semantic Indexing (LSI) [2] [11] [13], that have been used to analyze the word similarities between two given textual inputs (e.g., query and source code). The Vector Space Model (VSM) creates a vector space of the words extracted from documents and queries, and then the distance between all the vectors of documents against the query's vector is computed to retrieve and rank the relevant documents

for the given query [2] [17]. LSI is a statistical model that takes into account the frequency of the word in the vectors in computing the similarity distance (e.g., cosine angle) between the vectors [55].

However, the main challenge of applying textual analysis is that it is difficult to achieve high recall and high precision [4], which are dependent on the availability and the quality of domain knowledge in source code files, and the quality of the user's query [11] [30]. In addition, if the programmer wants to query the codebase repository, one needs to provide the query terms that were used by other developers in implementing the code [11] [30]. Furthermore, IR methods have been applied widely to automatically recover the traceability links among the source code in a single software system [2] [3] [17] [62]; however, applying IR directly on a collection of similar software systems (i.e., same domain), will increase the false positive links (low precision) because the search space has been increased to a collection of software systems that can have similar implemented functionalities [17] [62]. Thus, the search results of retrieved files for a specific feature will be mixed from all software projects without distinguishing between the relevant files within each project.

2.2.2 Topic Modeling

Topic modeling is a machine learning technique that analyzes a text corpus to group related documents together based on the idea that relevant documents share the same topic(s) [51][56]. For instance, given a topic "t", a set of documents will be in the same cluster that has topic "t".

The commonly used topic modeling algorithm is Latent Dirichlet Allocation (LDA). LDA, a probabilistic statistical model, is widely used to analyze unstructured text to extract the distributions of semantic topics assuming each document was constructed using distribution of these topics [3] [17] [51] [56] [62]. This LDA model is an unsupervised machine learning

that does not require training data with training labels [3]. It is an advanced version technique of LSI, shows more effective results than LSI because it takes in account both the semantics and importance of the words in the documents [3] [51] [55].

LDA also has been used to apply topic modeling on software artifacts (i.e., source code) for software engineering tasks, including traceability link recovery, feature location, and impact analysis to group related files together [3] [47] [51] [56]. When applying LDA on source code for topic modeling, the input is a corpus (i.e., a document collection) that is generated from code identifiers and comments according to the desired level of granularity (e.g., classes, methods) [47] [51]. In addition to the input dataset, the model requires two configuration parameters as inputs that determine the outputs [3] [51]:

1. The number of topics to be extracted from dataset.
2. The number of iteration for sampling the topics.

As a result, the LDA topic model produces two outputs: list of topics and document-topic distributions. The list of the words in a topic is ranked based on the highest probability that the word belongs to the topic [47]. The document–topic distribution contains the probability distributions of documents by topics [3] [47] [51] [56]. As a result, the model can be queried to the desired mining task, such as clustering to group related files together, information retrieval, visualizing, etc. [3]. Thus, LDA avoids the challenge in creating the user's query's words to locate relevant files.

There are several challenges for applying topic modeling, including all challenges that are inherited from textual analysis for the same discussed reasons, such as:

- It requires a heuristics calibration of parameters [47] [51] to achieve the desired results because the text in software artifacts (i.e., source code) is different than text in

natural language. This limitation has been studied in [51] to determine the ideal configuration automatically based on the dataset being analyzed.

- No automatic labeling of the semantic of the topic (i.e. list of words). Thus, one infers the meaning of the topic by reading the words in the topic [3]. Automatic topic labeling is a current research area.

2.2.3 Static Analysis

Static Analysis techniques analyze the source code's structural information, such as data flow and control flow dependencies, method calls, and hierarchical relationships, to identify relevant source code regardless of lexical contents in the code [11] [29] [30] [39]. For instance, the call graph structural model shows the related files based on method call dependencies [29] where nodes represent methods and edges represent control flow [11] [54].

There are several challenges for applying Static Analysis technique such as:

- It works only for a homogeneous type of technology and only for programing languages (e.g., java or C programing language [22]). As mentioned in the exemplar in Figure 3, because a layered architecture consists of a mix of technologies (e.g., xml, java, and html), static analysis techniques are unable to generate a complete traceability link chain.
- It requires the starting point to be provided by a developer (e.g., method) to identify the relevant code files by manual or automatic navigation through structural data [11].
- It often returns false positive files with respect to the desired functionality. Therefore, it requires user feedback about each node to filter irrelevant code [11] [29] [39] assuming that a developer is familiar with the code [11] [30] [50].
- It gives imprecise results for dynamic configuration method calls where a specific code is activated based on a dynamic configuration [43] [50].

2.2.4 Dynamic Analysis

Dynamic analysis techniques depend on recording activated files during runtime by executing test cases (i.e., scenarios) to identify the relevant files [13] [11] [30] [37] [39]. This traceability information is collected by instrumenting or by profiling the system [11]. This technique is considered the most accurate [11] [37], but it is an expensive approach in terms of the time required for developing test cases (i.e., execution scenarios) [50].

In addition, there are challenges for applying the Dynamic Analysis technique, such as:

- The accuracy of the results relies on the quality of the developed test cases that may not invoke all the related files [11] [37] [39].
- The test cases may produce a large amount of traceability information that invokes irrelevant files due to the difficulty in developing a scenario that executes only the desired task of the system [11] [47].
- Legacy software systems may no longer be executed [50]; therefore, a specified feature in a software system cannot be executed [11].

Based on this survey, each technique complements the other to achieve desired results. Therefore, combining the above techniques is a common approach used in various research areas as discussed in the next section.

2.3 Research Areas

This section discusses several research areas that tackle similarly addressed problems. Some areas focus on recovering traceability links; connecting together relevant software artifacts produced throughout software development lifecycle, such as software traceability, feature location, and code clone. Vertical traceability connects one type of artifact, whereas horizontal traceability connects different types of artifacts [4]. Also, there are other research areas that have a different focus, but they employ similar techniques used in the DDC Tracer.

To the best of my knowledge, no study considers a complete traceability link chain of source code artifact, from the back-end to the front-end (i.e., database to UI), across multi-layered software architecture and across heterogeneous implementation files, leveraging Domain Data Concepts (i.e., schema concepts) (see Table 1). This traceability chain comprises the first phase for reusing source code from existing software systems in a white-box manner in layered architecture, using a lightweight technique.

Research Area	**Objective**	**DDC Tracer Goals**
Software Traceability	Establishing traceability links for different software artifacts	DDC Tracer focuses on source code traceability only
Software Reuse	Reusing the source code from existing software systems	DDC Tracer assists with source code identification to facilitate reuse task
Feature Location	Establishing traceability links between source code artifacts that only implement a specific feature based on the description provided by the user	DDC Tracer locates source code for all features that access the given DDCs as query
Code Search and Code Clone	Finding code examples (i.e., snippets) and duplicated code fragments	DDC Tracer locates a complete traceability chain from back-end to the user interface
Database Analysis	Leveraging SQL statements to recover schema or extract the feature model	DDC Tracer focuses on establishing traceability between DDCs and implementation files to support source code reuse
Recommender Systems	Determining the context of a user to provide appropriate source code recommendations	DDC Tracer provides recommendations based on similarity of DDCs and complexity of SQL statements

Table 1: Comparison of related research areas with our DDC Tracer technique

2.3.1 Software Traceability

Software traceability has traditionally focused on requirements traceability (high-level to low level requirements), with the main goal of demonstrating that a delivered system meets the user requirements [27]. Other approaches suggest linking different software artifacts by centering the traceability links on the code (i.e., recovering traceability links between code and artifact to identify if any two different artifacts are related) [14] to support maintenance tasks, or centering the links on the architecture of software systems [5] to provide a high level understanding of the code structure.

There are varying levels of tool support for recovering the traceability links across different software artifacts. One class of techniques, information retrieval techniques IR [8], uses textual similarities between source code and various documents to recover the traceability links automatically. Another technique captures the links between heterogeneous artifacts while a developer activates the required artifact throughout the software development process [5].

Thus, software traceability research mainly focuses on establishing the traceability across different types of software artifacts, such as requirements, architectural designs, and code [8]. Our approach centers the traceability links on Domain Data Concepts (DDCs) that are used by domain practitioners (see Figure 4 b). Focusing traceability links on DDCs enable finding heterogeneous implementation files, including SQL statements, source code, and configuration files that manipulate specified DDCs. This set of files forms a traceability link chain that developers can follow to find reusable implementation files for given DDCs.

The challenges of tracing across one type of software artifacts (e.g., implementation files), referred as vertical traceability, have also been examined in the industry [38]. Yet, our

technique focuses on challenges encountered in tracing DDCs in a contemporary layered application.

2.3.2 Software Reuse

There are several techniques for reuse, including using reusable libraries, reusable components, product line engineering (i.e., domain engineering), software architecture, and source code generators [8]. The challenge with these systematic reuse techniques is the upfront cost for implementing such reuse before any cost savings can be realized [21].

Recent reuse techniques aim to address this challenge, such as using variant analysis between different source code to determine the commonality and variability, using flexible code generator that allows developer to weave in their manual modification, and connecting use cases to their implementations of source code [8].

Other approaches [32] [39] [40] focus on pragmatic code reuse by extracting the selected source code from the original software systems and integrating it into a new system for a desired functionality that is being developed. One technique transfers the client source code of web applications with a tool support (Firecrow) [40], whereas another technique transfers the server source code (e.g., Java) through a developed tool, Gilligan [32]. Thus, the main goal of pragmatic code reuse is the actual movement of existing source code into a new system, while identifying the source code that will be reused is left to developers. Our technique addresses this lack of reusable code identification by enabling a developer to find candidate reuse source code in the layered architecture applications based on specified Domain Data Concepts.

2.3.3 Feature Location

Concept or feature location is a closely-related area of research that aims to find human concepts within a software system implementation [41] [52]. Concepts may take on a

different definition, depending on the context of its usage [8]. For example, concepts may be related to software change, functions, or domain concepts [8]. In general, this research considers the definition of feature location as identifying an initial set (or gold set) of source code files that implement a specific functionality of the software system [11].

Feature location techniques generally use static and dynamic analysis, IR techniques, or hybrid techniques, such as IR with formal concepts analysis, or user feedback [8]. One technique, SCAN, assigns concepts to methods in execution traces using IR techniques and formal concept analysis [44]. Hybrid techniques aim to complement the limitations of using standalone approach to get better results [11], such as combining IR with static analysis techniques [11]. This combination of IR and static analysis techniques requires the user to provide a query expressed in natural language to describe a specific feature implemented in a software system. Thus, textual analysis reduces irrelevant code files produced by static analysis, and static analysis finds additional code from the initial identified code produced by textual analysis [11] [29] [64].

Indeed, in similar business database applications, the same features often have similar data access, and the semantic meaning of the feature (feature description) can be inferred from the data access code [63]. Thus, feature location (i.e., IR with static) can locate the implementation files for a feature(s) based on a user's request query (description). Figure 4 (A) shows the recovered traceability links (i.e., arrows) between the implementation file and the feature description query. However, a challenge with these techniques is determining the appropriate search query to obtain all desired implementation files [10] [56], particularly when developers work on software projects with which they are unfamiliar. Consequently, a developer needs to evaluate the search result and continually refine the query [10].

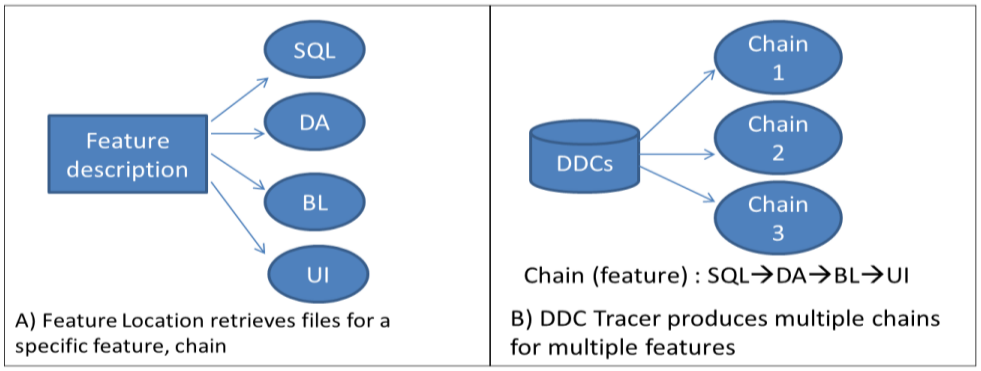

Figure 4: Feature location (a) versus Domain Data Concept traceability (b)

Similar data access can also have different feature implementation to satisfy different requirements. For example, the data concept "department name" can be used to implement "get employee information" and get "daily production report" features. Thus, this Master's work focuses on Domain Data Concepts (DDCs), which are a lower level of abstraction than features, to trace all features (which may be similar or different) that access specified DDCs. This focus allows connecting concepts that are familiar to domain practitioners (i.e., data model), such as those found in a database, with the implementation files, enabling them to traverse the problem space to the solution space, through the use of concrete terms (i.e., data schema concepts). As a result, DDC Tracer can guarantee locating all software features that access the given DDCs regardless of the feature description. Figure 4 (B) shows the traceability link chains that are centralized to the DDCs (chain to DDCs arrows).

In general, Domain Data Concept traceability differs from feature location because it locates source code according to given DDCs and does not need to find the DDCs in source code itself to be identified in the traceability chain. It is unlike automated feature location (e.g., IR with static) which relies on word matches, and domain knowledge embedded in source code, and user query.

2.3.4 Code Search and Code Clone

Code search and code clone tools are often used to aid developers to find code examples (i.e., fragments), find code snippets that can be reused, or find similar code [8]. Code search tools use various techniques to match the query against the code, including regular expression, structure of the code along with vector space model, combination of IR techniques and program analysis, or static and dynamic specifications with program analysis [8]. Some tools also include support for searching through SQL statements [8]. Code clone tools assist developers in locating similar code and determine the evolution of the old code [8].

These techniques are concerned with locating a piece of code independently rather than locating a complete set of files that implements a particular feature. Meanwhile, the technique in this Master's thesis supports searching for implementation files found in a traceability link chain and searching for similar code based on similar accessed DDCs.

2.3.5 Database Analysis

There are also techniques that leverage information found in SQL statements or database access code. These include database schema recovery using SQLs, identification of features using data access code, and static analysis of both SQL and source code to uncover SQL errors to avoid runtime errors [8]. On the other hand, this Master's thesis focuses on establishing traceability to support reuse.

2.3.6 Recommender Systems

Recommender systems for software engineering (RSSEs) have been developed to assist developers in locating relevant examples, guide software changes, or find reuse opportunities [8]. The main functionalities of recommender systems are to collect data, run recommendation heuristics, and provide the results in an understandable format to the developer [8]. RSSEs, based on recommender systems for online shopping applications, also

have the challenge of determining the context of the query [53]. Our approach has some similarities with recommendation systems, but is different in the underlying purpose. Our focus is on tracing Domain Data Concepts across the various layers of a layered application, and not on determining the context of a user to provide appropriate source code recommendations.

2.4 Summary

The previous sections discussed the similarities and differences of DDC Tracer with related research areas. As shown in Table 1, each research area is crafted to support its particular goal. Meanwhile, the goal of this Master's thesis is to establish the traceability links between source code files that access a given set of DDCs. In particular, it focuses on tracing a given set of DDCs across layered applications and across heterogeneous implementation files from back-end and all the way to the user interface. Because DDCs may also be related to features, a side-effect of this work is the ability to locate implementation files for features in software systems.

As shown in Figure 2 and Figure 3, the detailed exemplar, each layer has its own set of technologies; therefore, solely using text analysis or static analysis is inadequate to trace all heterogeneous implementation files that manipulate a given set of Domain Data Concepts (DDCs). As will be discussed in the next chapter, the DDC Tracer is built upon textual analysis including topic modeling and static analysis techniques.

CHAPTER 3 TECHNIQUE: DOMAIN DATA CONCEPT TRACER

This chapter presents the key steps of our technique (DDC Tracer), which solves tracing the Domain Data Concepts across layered applications and across heterogeneous implementation files that was discussed in the detailed exemplar in Chapter 2.

The chapter starts with an overview about our technique (DDC Tracer) and then each section describes the key steps in more detail.

3.1 Overview

The data access (e.g., SQL statements) mediates all communication between the business logic and the data stored in the database, as shown in the software architecture in Figure 5. As a result, identifying Domain Data Concepts as found in query statements allows tracing these concepts, not only to the business layer, but all the way to the client-side code. This establishes a complete traceability link chain of files that operate on a set of specified Domain Data Concepts (DDCs) (see Figure 5 Traceability link chain). This chain of files includes:

- Schema concepts (e.g., table & column names) that define Domain Data Concepts (DDCs)
- Data access files that access the schema concepts
- Business logic files that manipulate the data and run the business tasks
- User interface files that render data to the user

As a result, identifying the complete links of traceability chain of source code, files from DDCs and all the way to the user interface, facilitates locating the candidate source code to be reused based on the requested DDCs.

We can trace a data concept to every layer by leveraging the knowledge in a database schema (i.e., entity relationship), source code files (i.e., domain concepts), and in the structure of the code (static information). Figure 3 shows the traceability links we can create from the Domain Data Concept productID in the database table all the way to the user interface, which displays the product name of a given product ID.

Thus, in our technique, we defined two types of traceability links between the files as shown in Figure 5:

- Layer boundaries traceability links: Data to Server (labeled B.1) and Server to Client (labeled B.2).
- Within layer boundaries traceability links: within Server (labeled C.1) and within Client (labeled C.2).

Accordingly, our technique, Domain Data Concept tracer (DDC Tracer) consists of the following key steps, which are discussed in detail in the next sections:

1. Determine the starting point for tracing.
2. Create mappers at layer boundaries.
3. Create mappers within layer boundaries.
4. Connect mappers to create a traceability link chain.
5. Provide search capability.
6. Traceability chain metrics.

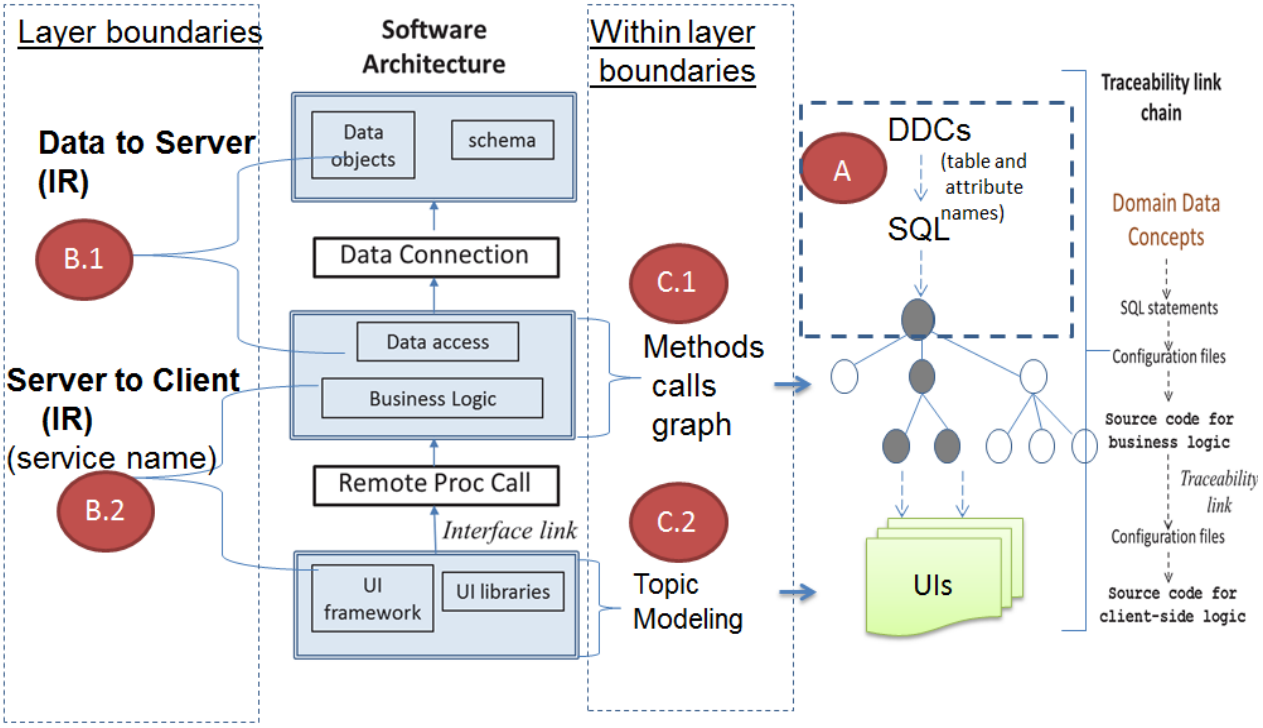

Figure 5: Domain Data Concept Tracer (our technique) [8]

3.2 Step 1: Determine the Starting Point for Tracing

A starting point for tracing may be a data model (e.g., schema) or data access code. A data model may be a starting point if developers are provided with a set of Domain Data Concepts and they wish to reuse existing code that accesses this set of concepts. With this starting point, some concepts may not have been used within a project before, and thus, have no traceability links to the code.

In this step, the Domain Data Concepts that a software project uses are identified. SQL statements are often used to access or manipulate the data stored in a database, so we use SQL statements as the basis of extracting Domain Data Concepts instead of a database schema, because we only wish to trace the database objects that are used in projects that an organization wishes to reuse, not all the data concepts in a database. For example, in Figure 3, Domain Data Concepts productID, name, and product can be extracted from the SQL, which is written in the file ProductMapper.xml (database layer label).

This step is performed by scanning all the SQL statements within a project and extracting the following elements: table names, attribute names, procedures, and reserved words. We consider table and attribute names as Domain Data Concepts as shown in Figure 5, labeled A. Also, we extract reserved words since they indicate the type of data access performed on the database objects (e.g., select, update, and delete).

An alternative is to start from a given project and extract the data access code from there. This option is more feasible in a case where the developer knows he/she wishes to reuse code from a given software project, or in a case where a data model is not accessible. It is important to extract Domain Data Concepts in data access code because data used in non-data access code may not semantically represent domain concepts, even if the same terms are used.

3.3 Step 2: Create Mappers at Layer Boundaries

Layer boundary mappers are crucial to bridge the syntactic gap that exists between layers of implementation files. Therefore, the second step is creating a mapper between data (i.e., back-end) and software (i.e., server) as shown in Figure 5 (labeled B.1), and between software layers; server and clients as shown in Figure 5 (labeled B.2).

3.3.1 Back-End to Application Server

The data access layer acts as a bridge between data stored in a database (back-end) and software. Once we have a list of domain concepts extracted, we can create a mapper between the data (back-end) and software (server) boundary by performing domain concept name tracing of the data access code.

Since the data access code is the first layer where data is either extracted or manipulated, often through the use of SQL statements, the mapping between Domain Data Concepts and SQL statement is based on whether the SQL statement contains specified Domain Data

Concepts. This mapping guarantees that a data concept embedded within the data access code is the same as the data concept in a data source (i.e., back-end).

SQL statements refer to table and attribute names that are present in a database schema. Table names have unique names. However, attribute names across different tables may not be unique, but SQL statements always specify the table on which it operates. As shown in Figure 3, productID in Products table (labeled Database) is mapped to the productID inside SQL statement written in ProductMapper.xml (labeled Data Access). Accordingly, a mapping (or traceability links) can be created between each of the Domain Data Concepts and all data access code that contains the concepts.

If certain technologies are used, as in the exemplar where iBatis or myBatis is used, data access code is found within configuration files instead of being embedded within a programming language [46]. In this case, data concept name tracing can still be used between data concepts and configuration files where SQL statements are written, then mapping between the configuration files and a programing language can be created by resolving the data access identifiers (e.g., SQL IDs). Thus, a mapper can be created from the Domain Data Concepts to configuration files via the domain concept name tracing (i.e., SQL statement) and from configuration files to the programing language via data access identifiers. For example, in Figure 3, productID in the Products table (within Database) is mapped to ProductMapper.xml (within Data Access) via name tracing to SQL (e.g., ID "getProduct"), and then to ProductMapper.java via the SQL ID.

3.3.2 Server-Side to Client-Side

The programing language used for business logic, which resides on the server, is often different from the programing language used for user interfaces, which resides on the client machine. It is also often the case that user interfaces are not necessarily implemented with a

programing language, but with scripts (e.g., javaScript) or frameworks as discussed in the exemplar.

To cross these language boundaries, a mapper can be created using the following steps:

1. Create a list of all the exposed services on the business logic layer that are used to access or manipulate Domain Data Concepts.
2. Find client-side implementation files that directly call these remote services in the business layer. Among these client-side implementation files, a service name tracing for each exposed service can be resolved. If the Model-View-Controller (MVC) design pattern is used in the client-side, these exposed services are often found in the controllers.

As a result, the mapping from the exposed services (remote service) on the business logic to the user interfaces files that call those services (e.g., controllers) can be created using service name mapping. For example, in Figure 3, the service name "getItemName" in RemoteObject.java (labeled Business logic) is exposed in the client source code ClientUI.xml

An alternative is to use static analysis techniques that examine the caller graph in case the server and client are implemented using the same programing language.

3.4 Step 3: Create Mappers within Layer Boundaries

Once the layer boundaries are identified, mappers within each layer are crucial to connect the traceability links at both server and client. Therefore, the third step is creating a mapper within the server as shown in Figure 5 (labeled C.1), and within the client as shown in Figure 5 (labeled C.2).

3.4.1 Within Application Server

Generally, the business logic is implemented using one programing language. In such cases, creating a mapping between implementation files is straightforward with a static analysis tool. The Data access code in data access layer that is related to one of the Domain Data Concepts through SQL calls, as shown in Figure 3 (ProductMapper.java), forms the starting point to get static information.

From this data access code (see Figure 5 label C.1), a static analysis can produce a method call graph until it reaches the files with exposed services at the business layer. For example, in Figure 3, we can create a call graph from ProductMapper.java (i.e., data access) to the rest of the source code in the business layer: CatalogService.java, CatalogFacade.java, and RemoteObject.java.

Thus, this call graph acts as a mapper among source code files in the server side. It is important to use a static analysis tool that provides the call graph at the method level, in order to only include source code files that call on the SQL statement of interest. Call graph represents a strong relationship between the files in abstraction format of large connections [29]. Furthermore, it is cheaper to be extracted than other structural information [29]. Using a call graph is consistent with other research areas, such as feature location [29] and requirement to code traceability [37].

There are cases where the business logic itself is implemented with multiple programing languages. In this case, files that reside at the language boundary must be identified, henceforth referred to as the Language Boundary File (LBF). The caller LBFs can be identified with reserved words that call on external files. For example, in Java, ProcessBuilder or Runtime [48] allows Java files to call executable files, which are created using other languages (e.g., C++, Visual Basic, C#). Java Native Interface (JNI) may also be

used to call source in another language [48]. After LBFs are identified for one programing language, the calls to the other programming language can be located. A mapper can then be created between caller LBFs in one language and callee LBFs in another language, using class-method name tracing.

3.4.2 Within Client-Side

Within the user interface layer, if only one language is used, then it is possible to perform a static analysis and create a mapper from call graphs as previously discussed on the server side. But, the user interfaces are often implemented with a mix of technologies (e.g., style sheets, xml, scripts, HTML) [40] and may use a framework to support design patterns (e.g., MVC), as shown in the exemplar (Figure 2). Some frameworks are event-based (e.g., Mate [23]), that handle events (e.g., button clicks) or dispatch events. In this case, the controller often represents the hub that handles all events responsible to invoke the remote object (i.e., exposed service in business logic) to access or manipulate data (see Figure 6). As a result, traces between user interface files can be created based on the event name. In this case, a mapper among user interface files can be created by using event-name tracing. In addition, since the UI files often contains semantic concepts represented in a natural language (i.e., fields caption) [45], we used the topic modeling technique to connect the controller, a starting point in the UI, to its views and models.

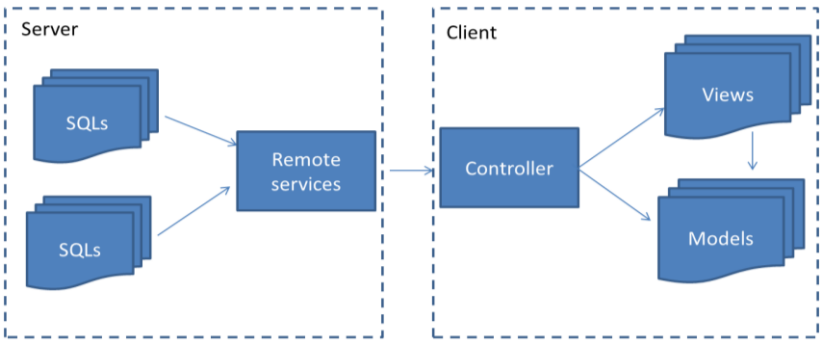

Figure 6: Controller as a hub on the client-side

Before we can apply topic modeling, source code files must be pre-processed to keep only the meaningful words that represent the domain knowledge because it treats the files as documents [9] [56]. The following steps (see Figure 7) describe how we pre-process and extract the topics from the source code, and how we connect the controller to its views and model:

A. Create a corpus of files using SQL statements and UI files. Data access provides the intended domain business functionality of the code [63]; therefore, we extract metadata from the database and scan all SQL statements in a software project to associate each attributes with its natural language description. Then, we create a corpus of each UI file by extracting meaningful text from UI source code. UI files contain semantic concepts represented in a natural language (i.e., fields' caption) [45] that correspond to attributes in SQL statements, such as comments and identifiers.
B. We split the compound words (e.g., camel-cased words, words with underscore) to represent the word in natural language. In addition, all reserved keywords in various technologies (e.g. Java, iBatis, Adobe Flex) and other common words (e.g.," the", "get", "set") are removed to keep semantic words only.
C. Once files pre-processing is completed, we run the topic modeling algorithm using the implementation described in [3] to extract the semantic topics. As a result, each document is assigned to its topics distributions.
D. Lastly, we connect related files. The top topic for the input file (e.g., controller) can be determined from its topic distributions. Then we obtain all the files (e.g., views, models) that contain that topic by a specified threshold. This threshold can be obtained by first checking which distribution thresholds yield the most accurate results.

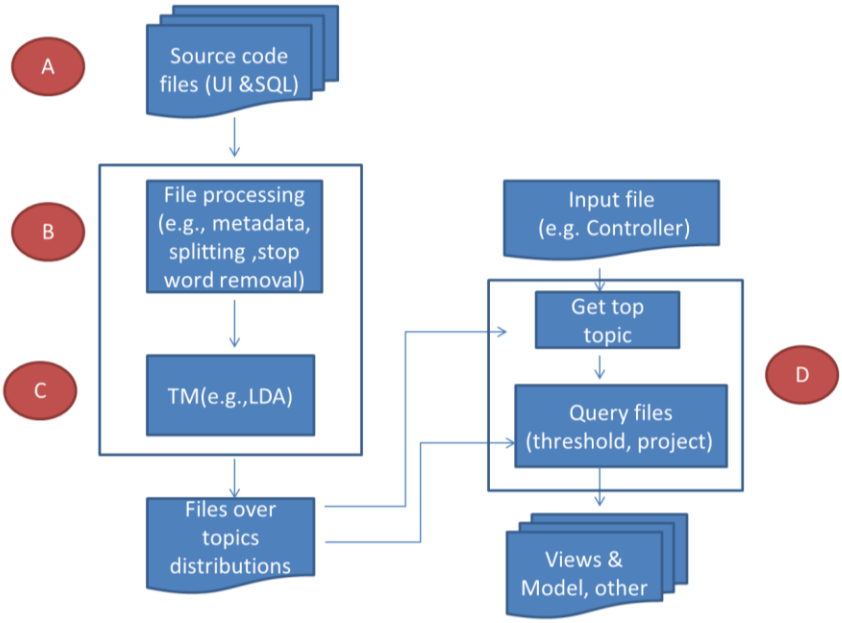

Figure 7: Traceability links within client using topic modeling [47]

As a result, topic modeling allows recovering the traceability links automatically among client code files and retrieving the related files. Furthermore, noisy traceability links can be eliminated by specifying a threshold for a topic distribution over documents, or we can rank the results according to similarity to the input document's topic distribution. Thus, we can create a mapping between each controller and its related files (e.g., views, models).

Since the controller represents the manager for all server and client communication (see Figure 6), the topic modeling technique connects all its relevant files (i.e., multiple views and models) regardless of the given DDCs; in other words, a subset of these UI files is only related to the given DDCs.

To improve the accuracy of this technique, we can use Manhattan Distance (L1) between the SQL statement and the retrieved UI files (e.g., views and models) to calculate the similarity between the retrieved relevant files and the SQL statement (DDCs). Often, views and models correspond to the attributes of the SQL statement [45]. Thus, with the vector space representation of Domain Data Concepts inside the SQL (a schema concept description) and

the vector space of the UI files, the L1 distance after pre-processing the files (i.e., splitting, stop words removal) can be calculated. As a result, the smaller L1 distance is the most relevant file to the SQL statement (DDCs). Another way is to use the topic modeling distribution of the SQL statement as the input file and get its relevant UI files. A drawback to this method is that UI files (e.g., forms) often access multiple SQL statements. Thus, the topic distribution of the UI files may not correctly match the topic distribution of a related SQL file.

3.5 Step 4: Connect Mappers to Create a Traceability Link Chain

The fourth step is to create a complete traceability link chain. Once the mappers are created between layer boundaries and within layer boundaries, a complete traceability link chain can be composed from Domain Data Concepts, data access, business logic, and user interface as shown in Figure 5 (Traceability link chain). In our exemplar (Figure 3), this traceability link chain can be created from heterogeneous implementation files starting from the given **Domain Data Concepts** productID in Products table (**database**), to the SQL statement in ProductMapper.xml called by ProductMapper.java source code *(***data access**), to **business logic** source code CatalogService.java, CatalogFacade.java, and RemoteObject.java, and to the **user interface** source code ClientUI.xml (e.g., controller and other views and models).

In addition, indirect traceability links can be recovered from the data model or source code structural information. If a data model exits (i.e., schema), connections among domain concepts can be obtained by searching for specified concepts in the schema and examine how they are related through entity-relationship diagram, for example (see database in Figure 3), Products and Inventory tables are related by ProductID. In object-oriented programming paradigm, structural information, such as inheritance and user-defined object that composed

the SQL attributes (e.g., Product.java in Figure 3) may provide additional indirect traceability links.

Thus, direct traceability links are those links in which a direct path to the DDCs is created, all the way to the user interface, whereas indirect traceability links are those links that can be inferred based on the direct relationships. These direct traceability links are the target because they connect the initial implementation files for the given DDCs.

Because the mappers can be automatically generated, the traceability link chain can be regenerated any time a change occurs within a software projects. As we will show later in the next section on Tool Support, our technique also provides reusable implementation files.

3.6 Step 5: Traceability Chain Metrics

For this step, a DDC can have multiple traceability links to SQL statements; consequently; there are multiple traceability links chains of implementation files from back-end to the user interface. Ranking those chains is essential to provide a recommendation for reusing the source code.

Since these traceability links chains consist of heterogeneous implementation files (e.g., DA, BL, UI), there are different options for ranking those retrieved chains. According to the literature, one approach [32] allows the developer to construct a chain of source code files within a programming language (e.g., java) and then the tool recommends which chain is less complex than others to be reused, based on computing dependency analysis. Another tool [1] defines the complexity metrics for the user interface files (e.g., alignment, grouping). Meanwhile, DDC Tracer uses data access (SQL) for ranking traceability links chains because it is crucial in determining the relevancy of the chains as similar features often access similar Domain Data Concepts (SQL) with the same or different of the rest of implantation files [63]. Thus, measuring the relevant SQL statements is a key step in DDC Tracer approach for

ranking the traceability links chains. It is the starting node of the chain to provide the initial ranking that can be integrated with business logic and UI in to provide the recommendation as an entire chain, considering all elements.

There are two different ways to measure relevant SQL statements to a given set of DDCs: one based on SQL similarity and another based on the SQL complexity.

3.6.1 SQL Statements Similarity

Since a similar feature has similar data access, one way of ranking the traceability chains is based on the similarity of the data access code. Thus, the relevance of the SQL statements to the DDCs has a cascading effect to the traceability links to the other implementation files. Highly, relevant SQL statements may indicate that the traces implementation files are also highly relevant.

In addition, the list of all the possible call chains for a given set of DDCs can quickly grow, especially for a large number of projects that operate on commonly used DDCs. To avoid information overload, the ranking of relevant SQL statements becomes crucial since the call chains are determined from the SQL statements that are used. If the most relevant SQL statements are placed at the top 10-20 results [36], then it is more likely that software engineers will find relevant traceability link chains or a set of implementation files that they can reuse.

With the vector representation of SQL statements, we can determine the relevance of a set of DDCs to linked SQL statements by using the Manhattan Distance (L1). The smaller the (L1) distance, the more relevant the statements are to the DDCs. SQL statements with 0 distances to a set of DDCs are considered semantically equivalent, since they contain the same concepts. Equivalence does not necessarily imply that the SQL operations are equivalent; rather, they are semantically equivalent because they access the same concepts.

3.6.2 SQL Statements Complexity

To determine SQL statements complexity, we present a data access code (SQL statement) as a vector of table names, attributes names, and query-specific words (e.g., reserved words, producer's calls). A vector may be a bitmap representation of DDCs (concept bitmap or **CB**), which indicates whether DDCs are present or absent from a query statement. A vector may also be a frequency representation of DDCs (concept frequency or **CF**), indicating the frequency of occurrences of these words in a SQL statement (as in the case of conditions in the WHERE clause or a nested SQL statement). For instance, if a DDC appears in SELECT and WHERE clauses, it may indicate a higher complexity than a DDC that only appears in a SELECT clause. In addition, a vector may be a representation of the presence or absence of SQL-specific words (**SS**).

These metrics may be used in combination with each other with weights assigned to each metric. The Concept bitmap metric can be combined with the SQL-specific words metric where the CB metric is weighted at 100% while the SS metric is weighted at 50% (**CB&SS**). Also, the concept frequency metric can be similarly combined with the SQL-specific metrics. The concept frequency metric may be weighted at 100% while the keyword metric is weighted at 50% (**CF&SS**). In practice, when using the DDC Tracer tool, we find that the CB and CF metrics perform similarly, but better than SS alone.

Still another way to rank is based on the complexity of query statements (adapted from Quah and Thwing [57]) (see Table 2). We determined the complexity of an SQL statement by assigning points to parts of the SQL (see Table 2). We assigned higher complexity points to SQL set operators (e.g., JOIN queries), since these types of queries are generally more difficult to understand. In addition, a SELECT statement nested within another SELECT statement would be given two points, since the word SELECT appears twice. The higher the complexity points of an SQL statements, the less relevant it is to reuse, since it is more

difficult to understand. This metric is useful in showing engineers simpler queries by first implying a simpler implementation chain. For instance, if there are two chains that accessed similar DDCs, the ranking will be determined based on the total points of measuring the complexity metrics for the SQL statement. The results are presented in increasing order of complexity measure.

Description	Points
1. Number of arithmetic operators (e.g., +,-, /, *), logical operators (e.g., not, and), comparison operators (e.g., number of times >, <, =>, =< are used)	0.5 / operator
2. Number of SQL operations (SELECT, INSERT, UPDATE, DELETE)	1 pt. / operation
3. Number of table names in FROM-clause	1 pt. / table
4. Number of set operators (e.g., JOIN, UNION, INTERSECT, MINUS)	1.5 pts. / set operator

Table 2: SQL complexity metrics used in DDC Tracer tool adapted from [57]

3.7 Step 6: Provide Search Capability

In the last step, we provide the search interface that enables the user to perform the searching tasks. In this interface, the user can search by single DDC or multiple DDCs to retrieve all implementation files from back-end to front-end represented as a chain. Each chain consists of data access, business logic, and user interface showing the connection between each node within the chain. Moreover, the results are ranked based on relevant metrics measurements to recommend the reusable source code. The project name in the results shows where the traceability chain can be found to help the developer get the source code quickly.

Once we have a traceability link chain, we can search for any files along the chain. For example, we can search for DDCs and obtain related implementation files or search for a source code and obtain related DDCs (i.e., reverse direction). Furthermore, we can display

search results by complete traceability link chains, to aid reuse, or by specific type implementation files (e.g., user interface code or business logic code).

CHAPTER 4 TOOL SUPPORT

This chapter describes the implemented tool support for the DDC Tracer, which automates the described steps in the previous chapter: extracting DDCs, creating mappers at layer boundaries, creating mappers within layer boundaries, connecting mappers, ranking the results, and providing search capabilities to recovery the traceability links of implementation files.

This chapter provides an overview of the tool architecture, implementation technologies, implementation details, implementation challenges, and the usage scenarios.

4.1 Architecture Overview

Figure 8 shows the architecture overview of the developed DDC Tracer. The tool has been developed using client-server architecture and web technology. This layered architecture provides the flexibility for integrating various components that were implemented throughout this Master's research. Furthermore, the web application provides the tool accessibility to all software engineers in a company.

The centralized back-end layer consists of 4-four repositories of extracted mapping data from the codebase repository. The mapping data is extracted by source code pre-processing as shown in Figure 9. This involves metrics data and the mapping information between DDCs and the data access layer, between the server and client, and within the layers that are stored in different data repositories: database, Apache Lucene file indexing, UI files correlation, and static analysis data.

The server layer presents the components that connect and retrieve this mapping data from the repositories. The client layer provides the graphical interface to display the search results

of mapping data for the provided DDCs. The detail of each layer is discussed in the Implementation Detail section.

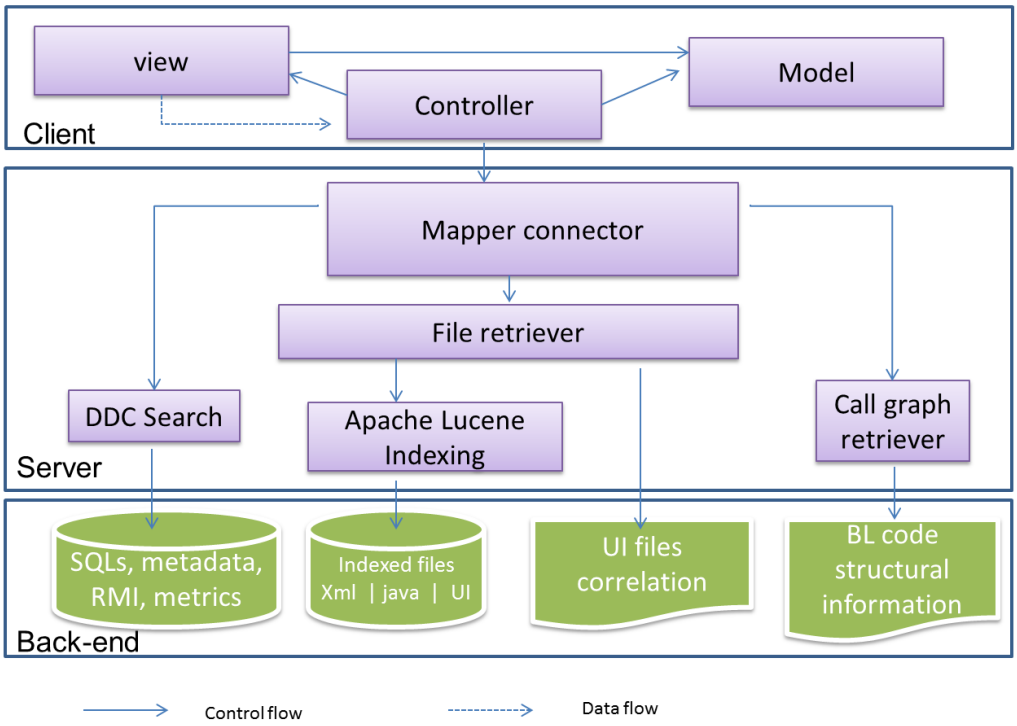

Figure 8: DDC Tracer architecture

4.1 Implementation Technologies

The tool has been developed as a web application (e.g., J2EE) and client-server architecture using different technologies:

- Java as programing language.
- Python as scripting language to process source code.
- MySQL database as back-end to store the traceability information.
- Apache tomcat 7 as web server.
- JavaServer Pages to create web pages (HTML) based on the Java language.
- Topic modeling algorithm, using the implementation described in [3].
- Apache Lucene as full-text search tool.

- Static analysis tool to generate structural information of source code.
- SQL parser tool to parse SQL statements and extract the required information.

4.2 Implementation Detail

This section presents the detailed implementation of each layer presented in Figure 8.

4.2.1 Back-End

The back-end layer consists of 4-four repositories that hold the mapping information extracted from the codebase repository (see Figure 9). The project repository shows software projects that are added to the tool. Each rectangle represents the components that have been implemented to process the source code extracting mapping information: DDC mapping data extractor, Apache Lucene, static analysis, and topic modeling.

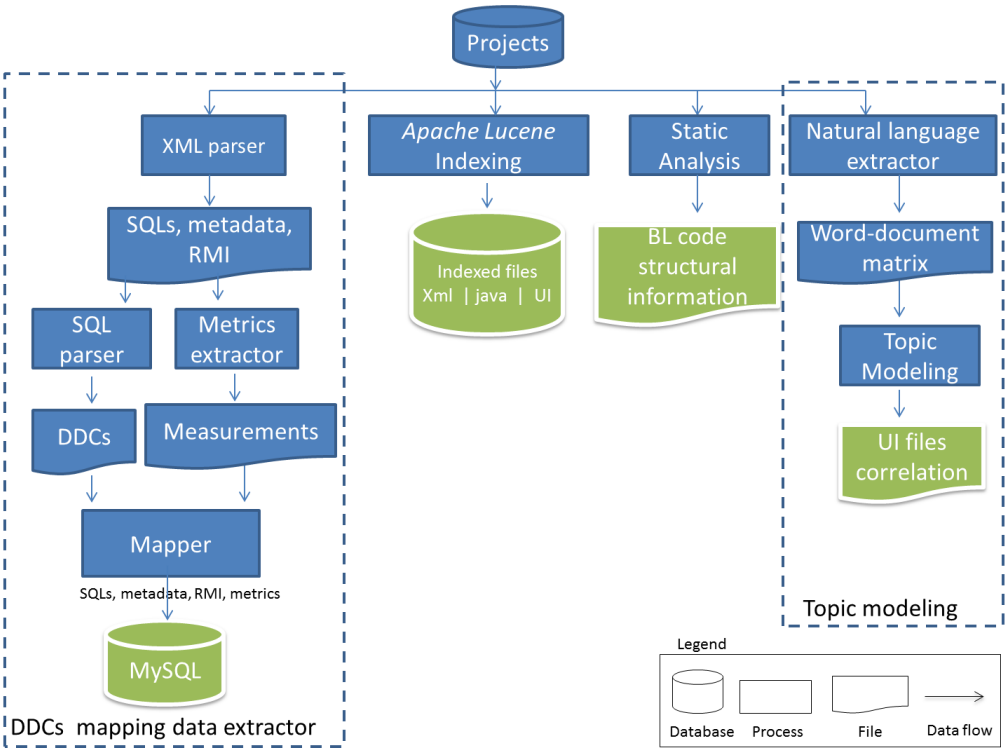

Figure 9: DDC Tracer extracting mapping information (bottom database and files are in the back-end)

4.2.1.1 DDCs mapping data extractor

This component (left dashed rectangle) in Figure 9 aims to create the traceability information across the layers' boundaries: back-end to server and server to client. To achieve this, many sub-components have been implemented using Python scripts and Java to pre-process the configuration files, the SQL statement, and the source code as following:

XML parser

Python scripts were created to pre-process all configuration files (i.e., xml format files), to extract DDCs, metadata, and remote services from the software projects. First, since some of these configuration files (e.g., iBatis and myBatis mappers) contain the SQL statements one task of these scripts is to scan all configuration files and extract SQL statements. Next, another task is to scan for remote objects that defined the exposed services by UI. This information includes the mapping between the source code and its remote ID in the software system that can be extracted from the Spring framework configuration files. Finally, metadata, such as the project name, mapper file name, and SQL ID were extracted along with scanning the configuration files. Eventually, all extracted information was dumped into text files according to a specific format representation for further processing by the mapper component.

SQL parser

Once the SQL statements were extracted, the SQL parser was used to extract the actual Domain Data Concepts (e.g., tables and attributes) from SQL statements. It was implemented in Java using the General SQL parser library [58] that scans all extracted SQL statements and pre-processes the SQL to clear the iBatis/myBatis syntax and then extracted the Domain Data Concepts. All extracted Domain Data Concepts for each SQL were dumped into text files for further processing by the mapper component. In addition, the SQL

parser was used to extract the similarity and complexity metric information such as the DDCs in SQL statements.

Metrics Measurements

All extracted SQL statements were parsed to extract the similarity and complexity metrics (discussed in Chapter 3 step 5) to rank the retrieved results: such as number of tables, SQL syntax keywords, arithmetic, and logic operators. These metrics measurements were extracted by implementing Java using the SQL parser and textual parser and dumped in MS Excel files by scanning all SQL statements.

Mapper

As a result, the mapper processed the data generated from the following: the XML parser, the SQL parser, and the metrics measurements to import the traceability links information between DDCs and data access code (i.e., back-end to server), and between the server and the client (i.e., remote object) into MySQL database. Furthermore, it imports metadata and similarity and complexity metrics data.

4.2.1.2 Static analysis

The main goal of the static analysis component is to create traceability information among the source code (e.g., Java) within the server. Dependency Finder tool [18] was used to extract structural information (e.g., method calls). It analyzes Java codebase statically generating dependencies data for each given software project presented in XML format.

4.2.1.3 Topic modeling

This component (right dashed rectangle in Figure 9) aims to create the traceability information among the client-side code. The topic modeling using LDA implementation by

[3] was used to extract the semantic topic of UI files that generates the file correlation analysis.

After we pre-processed the source code files to extract the natural language (meaningful text) from the identifiers and comments as discussed in Chapter 3 (Create Mapper within Layer Boundaries), we ran the LDA topic modeling with the following parameters: number of topics (value: 100), number of words per topic (value: 10), and number of iterations (value: 10,000). We used a heuristic during pre-processing to ensure no more stop words were shown in the topics, and to ensure that the topics are semantically meaningful.

As a result, topic modeling generated the correlation data of UI files based on semantic topics distributions. This data was generated in a text file that was processed into MS Excel format.

4.2.2 Server

The objective of the server layer is connecting the traceability information produced by all mappers in the back-end for a user request. There are several components in this layer as shown in Figure 8.

4.2.2.1 DDC search

At the beginning of the search process, when the user provides the set of Domain Data Concepts in the request query, this DDC search accesses the stored data in MySQL to retrieve the mapping information between the provided DDCs and data access code (e.g., xml mapper files of iBatis/myBatis, and SQL Java code caller) along the metadata (e.g., project).

To identify the data access code, once the mapping between DDC and SQL is obtained, DDC Search scans through the source code files that relate to SQL statements (via the SQL ID). Apache Lucence [20] a keyword-based search, was used to scan through the source code to access the caller (e.g., DA) to SQL by ID. The two search parameters used are “SQL ID” and

the “project name” associated with the call syntax used in the source code that enable the DDCs search to get the data access source code.

Moreover, to avoid false positive search results, a Java parser was implemented to parse the identified data access code to obtain which method exactly calls the SQL being searched (i.e., SQL ID). Identifying the actual method in data access code is crucial for static analysis mapper. As a result, the mapping between DDCs and the start node, the method in Java source code calling those DDCs, is determined. This traceability information is the starting point in the traceability chain (DDCs→DA) that serves as input (starting method) to the call graph retriever.

4.2.2.2 Call graph retriever

Once the data access code (i.e., starting method) is determined, the call graph retriever generates all method calls invocations starting from the root (i.e., starting method) reaching the remote service that is configured to be exposed by the user interface layer. This was implemented as a Depth First Search algorithm using Java that traverses the call graph data generated from the static analysis tool (Dependency Finder) represented in xml format. It takes the root node (method signature) and the project as parameters, then finds this root in the xml data and explores all paths from the root through leaves in the call methods invocation chain. These leaves nodes became the remote services; they serve as input to the file retriever component to get the UI file exposing the service. Thus, the traceability information within the server is determined (DA→BL).

4.2.2.3 File retriever

The file retriever is responsible for querying the results generated by both IR techniques: Apache Lucene and topic modeling (i.e., file correlation). First, when the remote service is determined, the file retriever uses Lucene to locate all UI files that expose this remote service

by resolving the service name in the UI files and according to the given project name. The actual service name (i.e., ID) is obtained from the mapping data (i.e., remote object mapping) in MySQL database. Thus, the traceability links between the server and client are determined (BL→UI).

Moreover, in the MVC client design, the UI files that often expose the remote services are controllers, responsible for managing other UI files, mainly models and views. As a result, the file retriever uses the files' correlation data produced by topic modeling to connect the controller with its related files. It depends on finding at least one controller file that exposes a remote service identified by call graph retriever and this file is used as input to obtain all related files based on sharing topics according to the specified threshold of topics distributions and the project name.

The file retriever was implemented in Java to query indexed files using the Lucene library to get the input files that invoke a given remote service, and then to query MS Excel with topics correlation data to get other related UI files. Thus, the traceability links between server and client and within client are determined.

4.2.2.4 Mapper connector

The mapper connector component is the bridge (see Figure 8) between the client layer and server layer in The DDC Tracer tool. It is the manager that retrieves the mapping information from each repository and connects them together. Thus, the complete traceability chain is constructed by this component to be transferred to the client graphical interface.

4.2.3 Client

The client layer was implemented in MVC to lower the maintenance tasks of developing the user interface of the DDC Tracer. This layer is responsible for accepting the user query and

communicating with the server layer to extract the mapping information for the provided DDCs. The search results are displayed to the user graphical interface.

As a result, the tool, DDC Tracer, is able to create the complete traceability links chains from back-end to data access code to business logic to the user interface via Domain Data Concepts (DDCs→DA→BL→UI, others). In addition, the traceability links chains can be ranked based on metrics measurement stored in MySQL database.

4.3 Implementation Challenges

In developing the DDC Tracer tool, several challenges were encountered with xml files pre-processing, SQL statements parser, and the mix of technologies in dataset.

4.3.1 Challenges with Pre-Processing the XML Files

First, it was a challenge to extract the SQL statements from the xml configuration files. Initially, we used XSLT (Extensible Stylesheet Language Transformations) that transforms the xml content to plain text, and thus we split (using split script) the xml file into individual SQL statements before applying XSLT, and then associated the metadata (e.g., SQL_ID, project name) manually because XSLT extracts only text between SQL operators tags (e.g., SELECT, UPDATE, DELETE, INSERT).

Another challenge in automating the SQL extraction was identifying, which XML configuration files are mapper files due to multiple technologies used in the implemented software (e.g., error logger, Adobe Flex, Spring Framework). To work around this challenge, each xml file was scanned and examined to determine whether it contains the SQL operators (e.g., SELECT, UPDATE, DELETE, INSERT) prior to pre-processing.

The next challenge was the two different versions of the mapper framework used in the projects (e.g., iBatis and myBatis mappers). The tags in the xml file were examined to determine the version of the mapper and then parsed accordingly.

4.3.2 Challenges with Pre-Processing the SQL Statements

1. The software system uses two types of SQL statements: dynamic and static. The dynamic SQL statement is constructed at run-time and currently not fully supported, whereas the static SQL statements are fully supported since the full statement is available for parsing. To work around dynamic SQL, extracting the text and creating a bag of words can be used.
2. Performing a straight parse is not effective because different text representations can represent the same database object. For example “object1”,”table1.object1”, or a completely different name (through the use of aliases), can all represent the same database object. Thus, we used a commercial SQL parser tool.
3. The same symbol may have two different meanings. For example, an equal sign may be a comparison operator or it may represent a JOIN set operator. Thus, when calculating the complexity metrics, an equal sign is assigned 1 point for either case.
4. SQL has several proprietary dialects and existing SQL parsers may cater to only some of the dialects. Consequently, it was difficult to find a suitable SQL parser. In our case, ZQL [65] was used as an appropriate middle ground because it was able to parse basic SQL statements which allows us to obtain the database object. A limitation associated with ZQL parser was that it was not able to directly accommodate sub-queries (i.e., nested queries) that represent the majority of data access code, and thus we also programmatically handled these cases. Therefore, we ended-up with using SQL parser, a commercial tool, but it was a trial version that supported what we need.

Also, we have tried ANTLR parser that requires the user to develop the grammar, but we did not succeed with this library.

4.3.3 Other Challenges

We also encountered other challenges, such as finding the static analysis tool that generates the structural data in plain text format. Moreover, the mix of technologies used in projects caused compatibility issues with different versions of Eclipse IDEs. Thus, it required the appropriate IDE for each project to investigate the source code. Furthermore, this mix of technologies kept the requirement changes during the tool implementations, and also added a challenge in determining stop words for topic modeling.

4.4 Usage Scenarios

The web technology implementation of the tool increased the accessibility of the tool to all the software engineers within the company. The tool first extracts DDCs and other important data from source code, then presents this information for the user based on the provided query, and finally provides a recommendation for the user to reuse source code.

Figure 10 shows a simple example of how the tool may be used. A user can enter any number of Domain Data Concepts (e.g., tables, attributes) in the search interface and submit the search query (see Figure 10 A), and then the tool returns the results of traceability links chains (see Figure 10 B). Each chain contains a set of heterogonous files across the layers connected together and along the software project where this complete chain is available (see Figure 10 C). A chain includes:

- Specified Domain Data Concepts (DDCs) (e.g., firstName, email).
- Data access code:
    - SQL statement contains the DDCs.
    - Configuration file where SQL is written (e.g., contact.xml).

    - Source code that calls the SQL (e.g., DBFactory.java).
- Business logic that manipulates the concepts (e.g., call graph), the edges shows the direction of method invocations.
- UI files that render the results (e.g.,MyContact.mxml).

In addition, the tool is able to rank the results of traceability links chains by the similarity and complexity metrics studied in this Master's thesis.

Some of the usage scenarios we envision include the following:

- Junior/new developers to a project can find examples on how to access and manipulate the various database concepts. In this scenario, it would be useful for developers to have a list of reusable code ranked by complexity, where the easiest to reuse code is listed first.
- Developers who wish to refactor similar code can search for similar code based on their relationships with Domain Data Concepts. When the database model changes (e.g., data base concepts are added, updated, or deleted), maintenance engineers can also search for software that operate on related concepts or specific database concepts and modify those pieces of code.
- Technical leads and project managers can use the tool to find Domain Data Concepts that are most frequently accessed by the software to optimize the database.

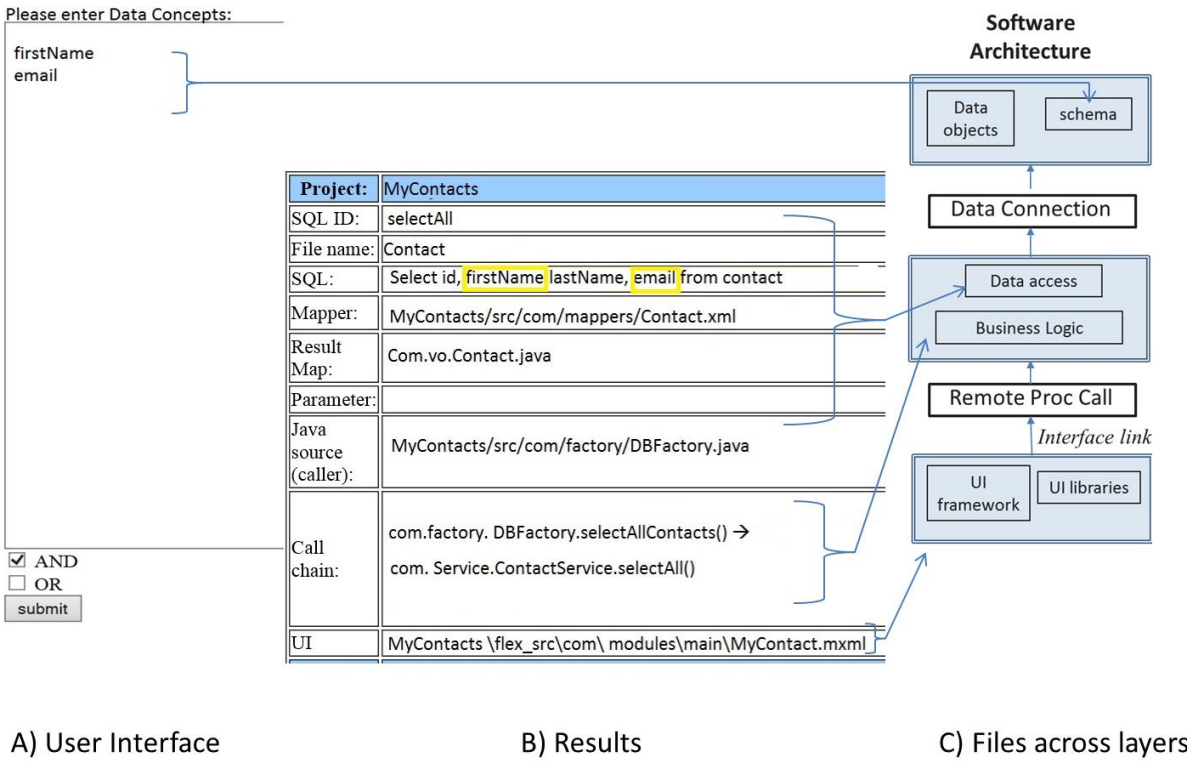

Figure 10: DDC Tracer tool

CHAPTER 5 EVALUATION

This chapter aims to validate The DDC Tracer to determine whether it meets the stated criteria in Chapter 1 of tracing Domain Data Concepts (DDCs) in layered applications.

It presents the evaluation goals and methods that involve a case study, an experiment, user feedback for validating the automated DDC Tracer tool, and feature comparison.

5.1 Evaluation Goals and Methods Overview

To validate the satisfaction of the criteria stated in Chapter 1 for the DDC Tracer, we defined the following evaluation goals and methods as shown in Table 3.

Evaluation method	C1: **Correctness**	C2: **Light-weight**	C3: **Utility**
1. Case Study on Aramco Dataset	• Find relevant server-side code for a given set of DDCs (e.g., Java files) • Find relevant client-side code for a given set of DDCs • Find a complete relevant traceability chain(i.e., server & client code) for a given set of DDCs		
2. Experiment	Measure DDC tracer effectiveness compared with other tools		
3. User study		Survey from Aramco software engineers	
4. Feature comparison			Compare the tool capabilities with similar tools and techniques

Table 3: Evaluation methods and goals

1. The first evaluation goal validates the correctness criteria, which assesses the ability and suitability of DDC Tracer to return the correct results of server and client implementation files. Accordingly, we defined the following evaluation task on Aramco dataset as a case study:
    a. Find relevant server-side code for a given set of DDCs (e.g., Java files).
    b. Find relevant client-side code for a given set of DDCs.
    c. Find a complete traceability chain (i.e., server & client code) for a given set of DDCs.
2. The second evaluation goal validates the correctness criteria in terms of the effectiveness of the DDC Tracer tool compared with other techniques. Thus, we conducted an experiment to measure the correctness of DDC tracer tool compared with information retrieval tools (topic modeling and Google Desktop search) using feature location technique.
3. The third evaluation goal validates that the DDC Tracer is light-weight, that is it should take less time for developers to learn and use compared to existing techniques or process. To evaluate this, we obtained user feedback from Aramco software engineers.
4. The fourth evaluation goal validates the utility that is the tool should be able to locate all relevant source code files that operate on a given set of DDCs across the software layers. To evaluate this, we obtained user feedback from Aramco software engineers and performed a feature comparison with other tools and techniques.

5.2 Case Study on Aramco Software Projects

The objective of this case study was to determine the fitness of each technology used in the DDC Tracer to return the complete traceability chain of implementation files that includes both server and client code. Due to the heterogeneity of the implementation files in layered

architecture, current techniques do not return all implementation files for a given set of DDCs. Therefore, we evaluated the correctness of returning the server code and client code separately to determine the fitness of each technology at each layer. We then evaluated the correctness of the entire traceability chain.

This section gives an overview on Aramco dataset and then presents different evaluation tasks: find relevant server-side code for a given set of DDCs, find relevant client-side code for a given set of DDCs, and find a complete relevant traceability chain for a given set of DDCs.

5.2.1 Aramco Dataset Overview

Saudi Arabian national oil and gas company (Aramco) is one of the largest oil companies in the world, managing proven conventional reserves of 260.2 billion barrels of oil and gas reserves of 284.8 trillion cubic feet and has the largest daily oil production [61]. We chose datasets from Aramco because they are industry software systems that consist of heterogeneous technologies of layered architecture, which are hard to find in open source projects. Moreover, these projects vary in complexity.

The case study consists of multiple software projects. These software projects are developed for oil and gas exploration business (i.e., same domain) serving different branches of hydrocarbon exploration to satisfy different business needs (i.e., different customer requirements). They access a central database (i.e., schema) that holds the core business information, and they are implemented in various technologies following layered architecture, such as iBatis/myBatis mapper framework [46] for data access (where SQL statements are written in the configuration files), Java programming language (business logic), Java Remote Method Invocation [48] (client and server communication), Spring

framework [24] (managing remote objects), Adobe Flex framework [34] (for developing rich web user interface (i.e., similar to desktop)), and MVC frameworks (for the user interface).

This combination of technologies leads to heterogeneous implementation files and the statistic of each software project is reported in Table 4.

Software project	No. of SQL stmts.	No. of configuration files (e.g., XML)	No. of Server Java files	No. of UI files (AS &MXML)
Project 1	425	38	197	195 +149= 344
Project 2	338	40	158	85+104= 189
Project 3	388	59	227	251+225=476
Project 4	162	32	112	68+71=139
Project 5	82	23	74	57+169=226

Table 4: Statistics of Aramco software projects used in the evaluation

5.2.2 Find Relevant Server-Side Code for a Given Set of DDCs

This task aims to show the capability of the tool to locate correct Java files using the static analysis technique. Once the starting node is correctly determined, it is straightforward to get all other related Java files exploring the structural information (e.g., method call).

In this task, we compared the results of DDC Tracer tool with a proprietary code search tool called “Find it EZ”, which performs string and regular expression matching on a directory of files specified by a user [35].

We ran two different types of searches: Type 1 and Type 2. Type 1 search is a single DDC search while Type 2 search is two DDCs combined with an “and”. Correct results for Type 2 search contain both DDCs. We ran single DDC queries five times and multiple DDC queries five times. It was run at least two times with different DDCs for each query type. We ran the query on two projects (project 1 & project 2 listed in Table 4), which consisted a total of 763 SQL statements, 78 configuration files (i.e., xml mapper file), and 355 Java files. With regards to accuracy, both DDC Tracer and Find it EZ retrieved correct file location (i.e., xml

mapper) of SQL statements that contained the DDCs that were queried (see Table 5). Find it EZ was not able to find related Java source code files because they do not contain the exact DDC text.

This limitation is similar to other text search tools. In addition, textual search may return false positive result because the search term could be written inside SQL statement (i.e., alias) that represents different term. As a result, in DDC Tracer tool, mapping each data concept to its SQL statement and then each SQL statement to its Java source code caller (i.e., SQL ID) is more effective. Thus, we can obtain other related Java files (e.g., through call graph) once the starting method has been identified.

Query type	No. of times	Task: Find SQL statements	Task: Find Java
Single DDC	5	Both tools were able to find SQL statements	"Find it EZ" was not able to find Java files
Multiple DDCs	5		

Table 5: Find relevant Java files to given DDCs[8]

5.2.3 Find Relevant Client-Side Code for a Given Set of DDCs

The main objective of this task is to determine the correctness of the related user interface files returned. In the MVC design pattern at the client-side, the controller connects the server code and the client code that operate on specific DDCs. Moreover, the controller is the component that updates the model and views the components to render the data for a given set of DDCs. Thus, correctly identifying the related files (e.g., views and models) for a controller, we can get the relevant client-side code for a given set of DDCs.

To evaluate the DDC Tracer capability to locate related client code, we assessed the correctness using two common IR metrics: precision & recall. Recall measures the percentage of retrieved files in relation to all recommended files, while precision measures the percentage of correctly retrieved files in relation to all recommendation files [37]. Some

projects from the Aramco dataset are implemented using third-party MVC frameworks, such as Mate and Swiz. These frameworks help create the mapping between the controller and its views and models that were constructed using the project directories and tracing events handling. This mapping was used to evaluate precision and recall of the tool capabilities to determine correct files at the client-side.

To evaluate the DDC Tracer tool, we assumed the controller file already had been identified in the traceability chain through resolving the server remote service name. Then, we analyzed 22 controller files from project 1&3 that use third-party MVC frameworks and measured recall and precision (see Table 6). Table 7 shows the summary of the results that indicate the tool was able to detect related client files with an average recall 76% and precision 71%.

Controller	**Metrics**		**Controller**	**Metrics**	
	P	R		P	R
file1	0.63	0.57	file12	1	1
file2	0.57	0.71	file13	0.43	0.9
file3	0.91	0.72	file14	0.15	1
file4	0.71	0.94	file15	0.82	0.78
file5	0.45	0.93	file16	0.28	1
file6	0.91	0.74	file17	0.92	0.8
file7	0.77	0.5	file18	1	0.89
file8	0.89	0.76	file19	0.68	0.74
file9	1	0.095	file20	0.98	0.88
file10	0.36	0.63	file21	0.55	0.53
file11	0.88	0.74	file22	0.91	1

Table 6: P & R in recovering the related model & views for a given controller

Software Project	Project3	Project1	ALL
Avg. precision	68.706 %	82.4 %	71.818 %
Avg. recall	75.382 %	80.8 %	76.614 %

Table 7: Summary of P & R UI traceability links recovering

5.2.4 Find a Complete Relevant Traceability Chain for a Given Set of DDCs

In this task, we evaluated the tool ability and correctness for tracing Domain Data Concept for the entire traceability links chain from the database to the user interface.

We used checked-in records to evaluate the correctness metric [9]. This idea is similar to where a developer assigns the level of confidence to each result related to the search query [43].

Accordingly, we defined the correctness metric (level of correctness) as if two adjacent nodes (i.e., files) in the traceability chain are checked-in together, within five minutes of each other, then the link between them is correct (see Figure 11). (Note: The code repository we used stores the file history by timestamps instead of transactions.) Thus, the developer might check-in one file or a group of files at a time. Five minutes was used based on the analysis of the check-in history records and based on my experience at Aramco. In fact, at the client-side, adjacent nodes are those links between the controller and each view or model file.

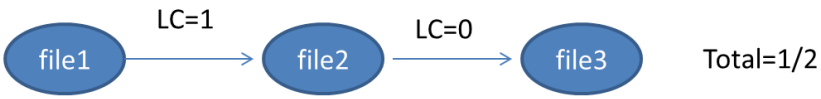

Figure 11: Correctness metric

To evaluate the accuracy, we developed a Java code that extracted the check-in records of source code for the Aramco dataset (Table 4) from the code management repository. This check-in record includes the history for each file, such as check-in time, by whom (e.g., developer), file type, and project name. We removed non-source code files, such as style

sheets and images, and then imported into a MySQL database. In addition, the big transaction of checked-in files together is a noise record because it might represent checking-in the project files; therefore, we removed any big transaction if it had more than 100 files.

We conducted the evaluation on three software projects: 1, 3 and 4 from Aramco dataset because they follow the MVC design pattern at client side. We generated all possible chains for each project by passing all DDCs (i.e., tables) to the tool. This produced duplicated chains and therefore we removed any chain that had the same nodes (i.e., files). Then, we went through each adjacent two files in the chain examining files that were checked-in together at least once (level of correctness). Thus, if the two files were checked-in together at least one time by a developer, we gave the link a score of 1, otherwise 0. For example, if the chain has three links between 4-four files, and one of the adjacent nodes has been assigned to a score of 0; the total score is 2/3 (for example, see Figure 11). Once the score for each chain was calculated, we calculated the average of all the scores of chains for each evaluated project as shown in Tables 8, 9, and 10.

Software project	**Check-in time**	**LC**	**Server**	**Client** (9 controllers) 25% TM	**Avrg.** (31 chains)
project No.1	within 5 min	1	100 %	69.275 %	72.083 %

Table 8: Tool correctness for project No.1

Software project	**Check-in time**	**LC**	**Server**	**Client** (27 controllers) 25% TM	**Avrg**. (122 chains)
project No.3	within 5 min	3	89.071 %	57.62 %	82.891 %

Table 9: Tool correctness for project No.3

Software project	Check-in time	LC	Server	Client (8 controllers) 50% TM	Avrg. (29 chains)
project No.4	within 5 min	1	86.839 %	57.679 %	78.689 %

Table 10: Tool correctness for project No.4

In project No.3 (Table 9) we used three times the check-in frequency for the correctness metric (LC); the files were checked-in at least three times, because the history shows all files were checked-in by one developer and mostly in a big transaction, so we kept the big transactions for this project. Furthermore, we used 25 % as a best obtain threshold, for topic modeling for recovering the traceability links between the controller and other relevant UI files, but we increased it to 50% as a best obtained cut-off for project No.4, because this project is not implemented with a pure MVC framework. The threshold was determined based on experiencing different cut-off that leads to get higher percentage of correct results. The performance of the threshold was evaluated by manual analysis of the source code.

The results indicated the acceptability of The DDC Tracer approach. The result of the client side is less than 70%, because it is not a common practice that the controller file necessarily will be checked-in with its views or models by the developer. Moreover, in project No.3, the check-in frequency level was 3, which is a large number relatively for a history created by one developer. In addition, project No.4 was not implemented using a pure MVC framework which could affect topic modeling results. This means that it is not necessarily that each controller has views and models, which increase the false positive files according to the specified threshold.

5.3 DDC Tracer Correctness Experiment

The goal of this experiment was to show the effectiveness of DDC Tracer tool compared with other tools in the literature. This experiment measured the correctness of the returned results of the complete traceability implementation files (i.e., server & client). Based on the

literature, feature location research area is suitable for the problem of connecting the implementation files together that implement specific software's feature. Thus, we compared DDC Tracer with another feature location approach.

5.3.1 Experiment Setup

The DDC Tracer accepts a set of DDCs as the user query, while feature location identifies the implementation files based on the feature description provided by user as a search query. It calculates the word similarity between the query and source code identifiers and comments to retrieve the files. In a database application, the intended business functionality of all implementation files can be determined by underlying data access code [63]. Thus, the description of each feature in the Aramco dataset case study was determined by the SQL statement ID as the analysis of the source code that shows the SQL ID represents the intended business feature.

Feature location technique uses IR, static analysis, dynamic analysis, or a hybrid of these techniques. Due to the heterogeneity of the source code in the layered architecture; we used feature location with IR tool as a comparable solution. DDC Tracer was compared with topic modeling and Google Desktop Search.

5.3.2 Correctness Measurement

In this experiment, we evaluated the correctness of 30 chains produced by DDC Tracer from three projects: project No. 1, 3, and 4 following exactly the same procedure and configuration (level of correctness and topic modeling threshold assigned to each project) discussed in Section 5.2.4 (Find a complete relevant traceability chain (server & client) for a given set of DDCs). Then, likewise, the correctness was measured for each chain using feature location baseline tools described in sub-sections (5.3.2.1 & 5.3.2.2). The description of the feature

implemented by each chain produced by DDC Tracer was extracted from the SQL statement ID.

Since those baseline tools techniques generated a list of files rather than a chain representation, measuring the accuracy between two adjacent nodes is not applicable. Therefore, we centralized all links to the SQL mapper files where SQL statements are written. Thus, the correctness was measured between the mapper file and each retrieved file by the baseline. The mapper file was assigned manually for evaluating the list of files generated by the baseline tools, which was taken from the corresponding chain produced by DDC Tracer.

5.3.2.1 Google desktop search (IR)

We used the Google Desktop Search as a representative feature location tool (i.e., it can accept feature description as a query term). Pre-processing the source code is a mandatory step to ensure achieving high recall and precision [10]. Accordingly, all source code files for the case study dataset (project No. 1, 3, 4) were processed to split camelcase and underscore producing a word-document matrix following the steps described in Chapter 3.

Then, each project was fed to the tool separately prior to performing the search to avoid retrieving a mix of files from different projects. Thus, for each chain produced by the DDC Tracer, the description was taken from the SQL ID and used as a query for Google tool pointing to the project where this SQL presents. As a result, a list of files produced by the Google tool was compared against the ground truth to determine its correctness.

5.3.2.2 Topic modeling

Topic modeling is a technique that can find relevant free-text files regardless of the technologies used in creating those files. It is a suitable comprehensive solution to search for

relevant heterogeneous implementation files. Furthermore, it does not require a user query which avoids challenges in constructing the appropriate query.

As a prerequisite for topic modeling, pre-processing the source code files before running the model was performed on all source code including SQL statements, Java, and UI files. Each SQL statement was parsed and the description of its DDCs was associated. Then all server and client code was processed to split the words and remove stop words producing a word-document matrix following the steps described in Chapter 3.

Thus, for each chain produced by the DDC Tracer, we identified its SQL and then we took the highest topic for this SQL from the baseline. After that, we queried all files that share the determined topic of the SQL according to the specified threshold and experience 10% better results than 15%. As a result, a list of files produced by the topic modeling baseline for each chain generated by the DDC Tracer was compared against the ground truth to determine its correctness.

5.3.3 Discussion

Table 11 shows the result of comparing 30 traceability chains produced by the DDC Tracer and their corresponding list of files generated by topic modeling and Google tool. The result shows the DDC Tracer achieves higher correct results than both topic modeling and Google tool. The results between the DDC tracer and Google tool is relatively close at the server which indicates any advanced feature location that combines IR and static analysis might achieve similar results of correctness.

As a result, the DDC Tracer in this Master's thesis can be considered as an extension to feature location research area that includes the client code covering end-to-end file location. Moreover, the DDC is based on querying specified data concepts rather than a feature description. Also, presenting the result organized as a complete chain facilitates the code

understanding and reusing for layered applications where code is distributed across different layers. In fact, getting the correct UI files helps the user judge the candidate reusable code more quickly, since the client code can be viewed in design view (i.e., graphical interface). This process is similar for the user when one searches and evaluates the usefulness of open source applications to borrow the source code.

Approach	Correctness (Avrg.) (30 chains)			check-in records measurement
	Server	Client	All	
DDC Tracer	95.8 %	83.7 %	85.7 %	two adjacent nodes in the chain
TM (10% cutoff)	45.1 %	44.9 %	49.5 %	center all links to the mapper file
Google Desktop Search	78.7 %	41.7 %	73.1 %	

Table 11: DDC Tracer tool correctness vs. topic modeling and Google IR tool

5.4 User Feedback

We also solicited the feedback of eight system analysts from the company in our study. Each one is experienced in developing J2EE applications using Eclipse IDE on a daily basis. The range of experience of these engineers is from 2 months to 15 years, and 6 out of the 8 analysts stated that they performed SQL and code search many times in the past. We presented the technique to the engineers and spent 10-15 minutes to demonstrate the tool to each of them.

5.4.1 Survey Question and Feedback

We asked their feedback regarding the following research questions:

1) Does DDC Tracer facilitate finding SQL statements and source code that could be reused for another project?

Seven out of the eight software engineers stated that the DDC tool can help them find SQL statements and source code that they can reuse for their project.

2) How does DDC tracer compare to the current process of searching for SQL statements and source code to reuse?

Five out of the eight software engineers stated that using the DDC Tracer is much easier than their current process of searching for reusable code. Two of these engineers pointed out that compared to their current process of searching for reusable SQL statements and source code, the tool can provide them significant time savings. Another engineer prefers to use the tool over his current process because of its capability—the ability to search for multiple key words and the ranking of search results. In addition to the five engineers, one engineer stated that the tool will enable him to find all the possible solutions that he can use for any of his projects.

3) Are software engineers willing to use DDC Tracer?

Seven out of the eight software engineers say that they are willing to use the DDC Tracer for their future software projects. On a scale of 1-5, where 1 is most useful and 5 is not useful at all, three of the five engineers rated the tool as 1, three engineers rated the tool as 2, and one engineer rated the tool as 3 (somewhat useful) (see Table 12). The rating of 3 came from an engineer who has not searched SQL statements or source code before.

Scale	**Most useful**		**Somewhat useful**		**Not useful**
	1	**2**	**3**	**4**	**5**
# of system analysts	3	3	1	0	0

Table 12: DDC Tracer tool rating [8]

5.4.2 Discussion

A majority of the subjects who participated in the study provided positive feedback regarding our tool, especially those who regularly search for SQL statements or source code. Some of the subjects provided suggestions for improvement, such as integrating the tool with software repositories. One of the subjects who stated that ranking the search results was most useful would like to be able to customize the ranking feature.

A possible threat to external validity may be present in our study. While the feedback was based on responses from software engineers who chose to participate in the study and may be specific to their particular application domain, these preliminary results indicate that using DDC to trace related heterogeneous implementation files is a viable reuse technique in an industry where layered applications are developed.

5.5 Feature Comparison

Table 13 shows a feature comparison between our approach (DDC Tracer) and existing techniques in the literature. We compared our technique with tools or techniques that have the most similar features to our technique, such as code search, database search, recommendation systems, and dependency analysis tools. The "possible" notation in Table 13 indicates additional steps or information may be required by the technique to support the feature.

Table 13 shows that our approach provides capabilities that are not currently supported in other tools, such as tracing DDCs to various parts of the implemented software including SQL statements, source code, and configuration files. Thus, this comparison shows the uniqueness of the DDC Tracer compared with other tools and techniques in the literature.

Feature	DDC Tracer (our tool)	Code Search	Database Search	Recommender Systems	Dependency Analysis
Trace DDC to SQL statements	Yes	Yes	Only search within a database	Possible	Possible
Trace DDC to source code	Yes	Possible	No	Possible	Possible, if DDC is traced to a method or a file
Trace DDC to various configuration files	Yes	Possible	No	Possible	No
Create traceability link chain from DDC to UI	Yes	No	No	No	Traceability link chain within programming language only

Table 13: Feature Comparison [8]

CHAPTER 6 CONCLUSION

This Master's thesis has presented the DDC Tracer, a data-centric traceability approach that facilitates code reuse in a fairly straightforward manner. By focusing on data concepts, we can find similar data concepts at the database layer that identify code that access and renders specified data concepts, and trace across the various layers of source code regardless of programing language or technology used; as a result, we can connect heterogeneous files that manipulate a given set of DDCs, including SQL statement, business logic, and user interface.

A major advantage of our technique, the DDC Tracer, is that each step is automated to provide tool support. Furthermore, it accepts only Domain Data Concepts (e.g., tables and columns) as solid query terms to construct the search query avoiding the challenges in creating the appropriate words for the search query compared with other related research areas.

The approach was evaluated using industrial case study from Aramco, experiment, feature and tool comparison, and feedback from software engineers.

6.1 Implications

The results of the evaluations indicate that the DDC Tracer is effective in facilitating reuse and is feasible to use in practices. Thus, the DDC Tracer can help practitioners in locating the source code and avoid effort duplication and wasting time of redeveloping same or similar features. As a result, development costs can be reduced and the quality and productivity of source code can be increased.

Although this work has only been evaluated on specific software architecture, the technique can be applied to any layered applications.

6.2 Limitations

There are several limitations of the DDC Tracer:

- Dynamically constructed SQL statements at run-time are not fully supported. The implemented XML parser in the DDC Tracer does basic parsing for some tags to recover the SQL statements, but it is not always successful for a complex dynamic SQL. Currently, we analyzed SQL as a bag of words to map DDCs to its SQL.
- Static analysis has a main shortcoming to determine the relationship between dynamic code dependencies (e.g., methods). For example, if different methods are called within an If statement, methods that are not related to the given DDCs may be included in the method call graph [64].
- The DDC Tracer tool relies entirely on the developer to assess the source code reusability and perform the reuse task (i.e., transfer the source code to the new system).
- The DDC Tracer tool requires administration tasks to regenerate the mapping traceability information: DDCs extraction, static analysis extraction, topic modeling file correlation extraction.

6.3 Future Work

Future avenues of work include analyzing the SQL statement structure, ranking the complete relevant traceability link chains, and creating visualization tools (e.g., dashboard). Also, further user evaluations will be conducted to improve the DDC Tracer technique in this Master's thesis. Furthermore, scaling the approach to large-scale of software projects is another avenue for future work.

Bibliography

[1] Khalid Alemerien and Kenneth Magel. Guievaluator: A metric-tool for evaluating the complexity of graphical user interfaces (s). In *26th International Conference on Software Engineering and Knowledge Engineering (SEKE), At Vancouver, Canada*, 2014.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, October 2002.

[3] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 95–104, New York, NY, USA, 2010. ACM.

[4] Hazeline Uy Asuncion. *Architecture-centric Traceability for Stakeholders (Acts)*. PhD thesis, Long Beach, CA, USA, 2009. AAI3369939.

[5] HazelineU. Asuncion and RichardN. Taylor. Automated techniques for capturing custom traceability links across heterogeneous artifacts. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 129–146. Springer London, 2012.

[6] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[7] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[8] Mohammed Daubal, Nathan Duncan, Delmar Davis, and Hazeline Asuncion. Tracing domain data concepts in layered applications. In *26th International Conference on Software Engineering and Knowledge Engineering (SEKE), At Vancouver, Canada*, 2014.

[9] Davis-D. Potts K. & Asuncion H. U. Dave, N. Uncovering file relationships using association mining and topic modeling. In *In eKNOW 2014, The Sixth International Conference on Information, Process, and Knowledge Management (pp. 105-111).*, 2014.

[10] B. Dit, Latifa Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 11–20, June 2011.

[11] Revelle-M. Gethers M. Dit, B. and Poshyvanyk. Feature location in source code: a taxonomy and survey. *Software: Evolution and Process*, page 25: 53–95. doi: 10.1002/smr.567.

[12] Klaus R. Dittrich, Dimitrios Tombros, and Andreas Geppert. Databases in software engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 293–302, New York, NY, USA, 2000. ACM.

[13] M. Eaddy, A.V. Aho, G. Antoniol, and Y.-G. Gueheneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 53–62, June 2008.

[14] Alexander Egyed, Stefan Biffl, Matthias Heindl, and Paul Grünbacher. Determining the cost-quality trade-off for automated software traceability. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 360–363, New York, NY, USA, 2005. ACM.

[15] Kate Ehrlich and Marcelo Cataldo. All-for-one and one-for-all?: A multi-level analysis of communication patterns and individual performance in geographically distributed software development. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 945–954, New York, NY, USA, 2012. ACM.

[16] Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL3. *SIGMOD Rec.*, 28(1):131–138, March 1999.

[17] H. Eyal-Salman, A.-D. Seriai, and C. Dony. Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, pages 209–216, August 2013.

[18] Dependency Finder. depfind.sourceforge.net.

[19] Swiz Framework for Adobe Flex. https://swizframework.jira.com/wiki/display/SWIZ/Home.

[20] Apache Software Foundation. Lucene. http://lucene.apache.org, 2013.

[21] W.B. Frakes and Kyo Kang. Software reuse research: status and future. *Software Engineering, IEEE Transactions on*, 31(7):529–536, July 2005.

[22] Frama-C. Formal verification, static code analysis. http://www.frama-c.com.

[23] Mate Flex Framework. http://mate.asfusion.com/page/documentation.

[24] Spring Framework. http://projects.spring.io/spring-framework/.

[25] Malcom Gethers, Amir Aryani, and Denys Poshyvanyk. Combining conceptual and domain-based couplings to detect database and code dependencies. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 144–153. IEEE, 2012.

[26] Achraf Ghabi and Alexander Egyed. Observations on the connectedness between requirements-to-code traces and calling relationships for trace validation. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 416–419. IEEE Computer Society, 2011.

[27] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *Software Engineering, IEEE Transactions on*, 32(1):4–19, January 2006.

[28] G. Henriques, L. Lamanna, D. Kotowski, H. Hlomani, D. Stacey, and P. Baker. An ontology-driven approach to mobile data collection applications for the health-care industry. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pages 578–583, August 2012.

[29] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 14–23, New York, NY, USA, 2007. ACM.

[30] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Investigating how to effectively combine static concern location techniques. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 37–40, New York, NY, USA, 2011. ACM.

[31] Reid Holmes and Robert J Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, pages 447–457. IEEE Computer Society, 2007.

[32] Reid Holmes and Robert J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology*, 21(4):20:1–20:44, February 2013.

[33] Danail Hristov, Oliver Hummel, Mahmudul Huq, and Werner Janjic. Structuring software reusability metrics for component-based software development. In *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pages 421–429, 2012.

[34] Adobe Systems Incorporated. Flex. http://www.adobe.com/products/flex.html.

[35] Find it EZ Software. http://www.finditez.com/.

[36] Steven Kirsch. Infoseek's experiences searching the internet. *SIGIR Forum*, 32(2):3–7, September 1998.

[37] Hongyu Kuang, P. Mader, Hao Hu, A. Ghabi, Liguo Huang, Lv Jian, and A. Egyed. Do data dependencies in source code complement call dependencies for understanding requirements traceability? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 181–190, September 2012.

[38] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Softw. Pract. Exper.*, 26(10):1161–1180, October 1996.

[39] J. Maras, M. Stula, J. Carlson, and I. Crnkovic. Identifying code of individual features in client-side web applications. *Software Engineering, IEEE Transactions on*, 39(12):1680–1697, Dec 2013.

[40] Josip Maras. Pragmatic reuse in web application development. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1094–1097, New York, NY, USA, 2011. ACM.

[41] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, November 2004.

[42] Carma McClure. *Software Reuse Techniques: Adding Reuse to the System Development Process*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[43] C. McMillan, M. Grechanik, D. Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120, May 2011.

[44] S. Medini, G. Antoniol, Y. Gueheneuc, M. Di Penta, and P. Tonella. SCan: An approach to label and relate execution trace segments. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 135–144, October 2012.

[45] N. Mfourga. Extracting entity-relationship schemas from relational databases: A form-driven approach. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society.

[46] Mybatis. SQL mapping framework for java. http://code.google.com/p/mybatis/.

[47] Kunming Nie and Li Zhang. Software feature location based on topic models. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 547–552, December 2012.

[48] Oracle. Java platform se. http://docs.oracle.com/javase/.

[49] Oracle. Remote method invocation. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[50] Maxime Ouellet, Ettore Merlo, Neset Sozen, and Martin Gagnon. Locating features in dynamically configured avionics software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1453–1454, Piscataway, NJ, USA, 2012. IEEE Press.

[51] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 522–531, May 2013.

[52] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23:1–23:34, February 2013.

[53] M.P. Robillard, R.J. Walker, and T. Zimmermann. Recommendation systems for software engineering. *Software, IEEE*, 27(4):80–86, July 2010.

[54] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in java. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 1–11, New York, NY, USA, 2004. ACM.

[55] Julia Rubin and Marsha Chechik. A survey of feature location techniques.

[56] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk. Topicxp: Exploring topics in source code using latent dirichlet allocation. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–6, Sept 2010.

[57] Tong seng Quah, Mie Mie, and Thet Thwin. Prediction of software development faults in pl/SQL files using neural network models. *Information and Software Technology*, pages 519–523, 2004.

[58] Gudu Software. General SQL parser. http://www.sqlparser.com/.

[59] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[60] Wikimedia. Model–view–controller. http://en.wikipedia.org/wiki/Model-view-controller.

[61] Wikipedia. Saudi Aramco. http://en.wikipedia.org/wiki/Saudi_Aramco.

[62] Yinxing Xue, Zhenchang Xing, and S. Jarzabek. Feature location in a collection of product variants. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 145–154, October 2012.

[63] Yiming Yang, Xin Peng, and Wenyun Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 215–224, October 2009.

[64] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: towards a static non-interactive approach to feature location. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 293–303, May 2004.

[65] ZQL. http://sourceforge.net/projects/zql/?source=directory.