Tracing Domain Data Concepts in Layered Applications

Mohammed Daubal, Nathan Duncan, Delmar B. Davis, Hazeline U. Asuncion

Computing and Software Systems
University of Washington, Bothell
Bothell, WA, USA
{daubalm, njd91, davisd1, hazeline}@u.washington.edu

Abstract—A goal of reusing source code is to lower development costs; however, reuse techniques usually require additional costs in creating generic source code and retrieving the appropriate source code from a code repository. In this paper, we present Domain Data Concept (DDC) Tracer, a novel traceability technique that facilitates reuse in layered applications in a cost-effective way. 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. We present a feature comparison which highlights the uniqueness of our approach. In addition, we conducted an industry case study where we analyzed software artifacts and solicited feedback from software engineers. Our case study indicates that our approach is a lightweight alternative to source code reuse techniques and is feasible to use in practice.

Keywords—Software traceability, reuse, structured query language, layered architecture style

I. INTRODUCTION

Literature has shown that code reuse lowers costs by lowering the amount of software that needs to be developed [25]. The ability to reuse code remains challenging in practice because a key criteria for success is its economic benefit to the organization (i.e., whether there is a good return on investment on the upfront costs for reuse) [25]. Current techniques for reuse include: creating a reusable library of components, using domain engineering (e.g., model-driven development, product line engineering), or using software architecture [25]. Code search tools have also been used to facilitate code reuse by enabling developers to search by textual similarity, program structure, or code semantics [13, 43]. 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 [27], 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., [4, 6, 24]). Thus, traditional static and dependency analysis tools are unable to trace through these implementation files.

This paper aims to address this gap in supporting code reuse in data-centric applications. More specifically, we support reuse for software applications that connect to databases by tracing concepts from the database layer to the user interface layer. Conceptually, these domain data concepts (DDCs) map to elements in the schema, such as table and column names. Our technique, Domain Data Concept Tracer (or DDC Tracer), connects data concepts with implementation files in a 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., communications within a framework, caller graph, etc.), 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 implementation file (e.g., configuration files, structured query language (SQL) statements). This approach also enables software developers to search for highly relevant implementation files for reuse.

The contributions of this paper include: (1) a data-centric approach to tracing DDCs in a layered architecture, (2) automatic mapper generators, and (3) evaluations based on an industrial case study and feature comparison.

The paper is organized as follows. The next section details the motivation behind our technique. Section 3 covers related work and background information. Section 4 presents our technique, DDC Tracer, and Section 5 presents our tool support. Our evaluation methods are discussed in Section 6. We conclude the paper with future work.

II. MOTIVATION

In large corporations, software development is often distributed among different teams and explicit collaboration among these independent teams is usually difficult [22]. Such is the case with Saudi Aramco, a Saudi Arabian national oil and gas company [7]. The company 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.

Because development is distributed along different business units, explicit collaboration among these independent teams is usually difficult in a large company setting like Aramco [22]. 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, since the search is limited to the developer’s memory or communication channels with other developers. In addition, a text search or a dependency analysis does not yield all the files that operate on a specified DDC because of the various technologies used. Thus, current techniques are inadequate to determine the connections between related files in an implemented system.

Additionally, software development projects in the context of large corporations operate on data that is often stored in back-end databases [31]. 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.)

III. RELATED WORK AND BACKGROUND

A. Software Traceability

Software traceability has traditionally focused on requirements traceability, with the main goal of demonstrating that a delivered system meets the requirements [30]. Other approaches have also suggested centering links on the code [21] or on the architecture of the software [12]. Our approach centers traceability links on DDCs to connect together heterogeneous files that manipulate specified DDCs.

There are varying levels of tool support for capturing traceability links. One class of techniques, information retrieval (IR) techniques [17], uses textual similarities between source code and various documentations to automatically recover traceability links. These techniques fail to recover traceability links that cross layer or language boundaries.

The challenges of tracing across one type of software artifact (e.g., implementation files), referred to as vertical traceability, have also been examined in industry [35]. Our technique, meanwhile, focuses on the challenges encountered in tracing DDCs in a contemporary layered application.

B. Software Reuse

There are techniques for reuse, including using reusable libraries and reusable components [25, 41], but an existing challenge is the upfront costs for reuse before any cost savings can be realized [25]. Recent reuse techniques aim to address this challenge, such as using variant analysis between different source code to determine the commonality and variability [20], using a flexible code generator that allows developers to weave in their manual modifications [34], and connecting use cases to their implementations [45]. Our approach enables developers to find reusable implementation files based on specified DDCs.

C. Concept or Feature Location

Concept or feature location is a closely-related area of research that aims to find human concepts within a software implementation [26, 36, 40]. Concepts may take on different definitions, depending on the context of its usage [26]. For example, concepts may be related to software change [26], functions [9, 36, 40], or domain concepts [11]. These techniques generally use static and dynamic analyses [23, 42], IR techniques [36], or IR techniques with formal concept analysis [37, 40] or user feedback [26]. A challenge with these techniques is determining the appropriate search term to obtain the desired implementation files. Our work, meanwhile, focuses on DDCs which are at a lower level of abstraction than features. This data focus allows us to connect concepts that are familiar to domain practitioners with the implementation files, enabling us to traverse the problem space to the solution space, through the use of concrete terms.

D. Recommender Systems

Recommender systems for software engineering (RSSEs) have been developed to assist developers locate relevant examples, guide software changes, or find reuse opportunities [32, 44]. RSSEs, based on recommender systems for online shopping applications, also have the challenge of determining the context of the query [44]. Our focus is on tracing DDCs across the various layers of a layered application, and not on determining the context of a query to provide source code recommendations. QueRIE is an SQL recommender system that aims to assist non-expert users of scientific databases to recommend queries that may be useful to a particular user [38]. Our technique is similar to QueRIE, but extends the connection beyond SQL queries to other implementation files.

E. Code Search and Code Clone Tools

Code search and code clone tools are often used to aid developers find code examples, find code that can be reused, or find similar code [13, 14, 43]. Code search tools use various techniques, such as text analysis [2], regular expression [3], code structure with vector space model [13], IR techniques with program analysis [29], or static and dynamic specifications with program analysis [43]. Some tools also include support for searching through SQL statements [5]. Meanwhile, code clone tools assist developers in locating similar source code [14]. Our approach supports searching for implementation files found in a traceability link chain and searching for a similar code based on similarly accessed DDCs.

F. Analysis of Data Intensive Applications

There are also techniques that leverage information found in SQL statements or data access code. These include database schema recovery using SQLs [15], identification of features using data access code [18], and static analysis of both SQL and source code to uncover SQL errors to avoid runtime errors [28]. Our tool, meanwhile, focuses on establishing traceability to support reuse.

G. Background on Databases and SQL

A database is a collection of interrelated data which represents concepts from the real world [19]. The definitions of these concepts are provided by a data model and the most popularly used data models are relational, object-oriented, and object-relational. The relationship between real-world concepts are represented in a schema [24].

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, StartDate, EmployeeClassification) [16]. Related DDCs are specified through table relationships. One way this 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 example,
EmployeeID may be used to uniquely identify each tuple or row in the Employee table. A foreign key is used to refer to a primary key in another table. For example, employees may work in any department. This relationship between the Department concept and the Employee concept may be expressed through the use of DepartmentID as a foreign key within the Employee table.

IV. DOMAIN DATA CONCEPT TRACER

This section discusses key steps in our approach while the next section discusses how these steps can be automated. Before we discuss our technique, we first provide a general overview of software systems that can use our approach.

Software that follows the layered architectural style has source code organized in different layers [46]. Each layer uses the services provided by an adjacent layer. A client-server style is a special case of the layered style where the number of layers is either two or three. We use a three-tiered client-server application using current technologies (see Figure 1) as an exemplar. A client-server style is popularly used in business applications, where the logic and data model are centralized [46]. As shown in the figure, the system consists of a heterogeneous mix of technologies: data manipulation or extraction (e.g., query statements), business logic (e.g., Java, C++), and client-side logic (e.g., Javascript, HTML). On top of these, companies may use frameworks (e.g., [4]) to lower development and maintenance costs and to standardize the code base.

A. 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 DDCs and they wish to reuse existing code that access this set of DDCs. With this starting point, some DDCs may not be used within a project and, thus, have no traceability links to the code.

An alternative is to start from a given project and extract the data access code from there. This option is more feasible in cases where developers know they wish to reuse code from a given software project or in cases where a data model is not accessible. Note that DDC extraction must occur in the data access code (e.g., where database access occurs or where file access is performed) because data used in non-data access code may not necessarily represent domain concepts, even if the same terms are used.

B. Create Mappers at the Layer Boundaries

Layer boundary mappers are crucial to bridge the syntactic gap that exists between layers of implementation files. We now discuss how mappers can be created between data and software (e.g., data and business logic layers) and between software layers (e.g., business logic and user interface layers).

1) Data to business logic layer boundary:

Once we have a list of DDCs we wish to trace, we can create a mapper between data and software boundary by performing DDC name tracing at 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), this guarantees that a data concept embedded within the data access code is semantically the same as the data concept in a data source (e.g., database schema). Thus, a mapping can be created between each DDC and all data access code that contains the DDC.

It is also possible that configuration files are used to store data access code instead of embedding them within a programming language (e.g., [4]). In this case, DDC name tracing can still be used between DDCs and configuration files. Then, mapping between the configuration file and a programming language can be created by resolving the data access identifiers (e.g., SQL IDs) in the configuration file and the programming language. Thus, a mapper can be created from the DDCs to configuration files via DDC name tracing and from configuration files to the programming language via data access identifiers.

2) Business logic to user interface layer boundary:

The programming language used for business logic, which resides on a server, is often different from the programming language used for user interfaces, which resides on a client machine. It is also often the case that user interfaces are not necessarily implemented with a programming language, but with scripts (e.g., Javascript) or frameworks (e.g., [33]). To cross these language boundaries, one can create a mapper using the following steps. First, create a list of all the exposed services (i.e., provided interfaces) on the business logic layer that are related to at least one of the DDCs selected in the previous section. Then, find client-side implementation files that directly call these exposed services in the business layer. Among these client-side implementation files, one can perform a service name tracing for each of the exposed services. Thus, a mapping from the exposed services (on the business logic) to the user interface files that call those services can be created using service name mapping.

C. Create Mappers within Layer Boundaries

1) Within business layer:

Generally, the business logic is implemented using one programming language. In such cases, creating a mapping...
between implementation files is straightforward with a static analysis tool. One can start with data access code that is related to one of the DDCs. From this data access code, a static analysis can produce a call graph at the file-method level until it reaches the files with exposed services at the business layer. This call graph can act as a mapper among source files in the business layer.

There are cases where the business logic itself is implemented with multiple programming languages. In this case, one must identify which files reside at the language boundary, 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 [39] 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 code in another language [39]. After LBFs are identified for one programming language, one can locate the calls to the other programming language. A mapper can then be created between caller LBFs in one language and callee LBFs in another language using class-method name tracing.

2) Within user interface layer

Within the user interface layer, if only one language is used, then it is possible to perform a static analysis and create mappers from call graphs as previously discussed. User interfaces are often implemented with a heterogeneous mix of files (e.g., stylesheets, scripts, HTML). In addition, user interface frameworks may be used to handle events (e.g., button clicks) or to dispatch events. If an event framework is used, 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.

D. Connect Mappers to Create a Traceability Link Chain

Once the mappers are created between boundaries (e.g., layer boundary, programming language boundary) and within boundaries, a complete traceability link chain can be composed from DDCs to the user interface layer. In addition, if a data model exists (e.g., schema), connections among DDCs can be obtained from the data model. This can be accomplished by searching for specified DDCs in the schema and examining how the DDCs may be related (e.g., usage of foreign and primary keys). These connections can provide additional indirect traceability links between source code and DDCs.

Because the mappers can be automatically generated, the traceability link chain can be regenerated any time a change occurs within a software project. As we will show later in the paper, our approach also yields reusable implementation files.

E. Provide Search Capability

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 source code and obtain related DDCs. We can display search results by complete traceability link chains, to aid reuse, or by specific type of implementation file (e.g., user interface code or business logic code).

V. Tool Support

Most of the features of the DDC Tracer have been implemented: DDC extractors, boundary mappers, call graph mappers, and search tool. The DDC extractor from data access code requires a parser. For this, we used the ZQL parser [8] to parse SQL statements. To support various types of name tracing, we used the Lucene search library [10]. To implement boundary mappers, we used Python scripts and Java to pre-process configuration and source code files. To implement our call graph mappers, we used DependencyFinder [1]. We used MySQL to store the traceability links between DDCs, SQL statements, and various implementation files. To increase the accessibility of the tool to software engineers within a company, we developed a web user interface.

Some of the usage scenarios we envision include the following. Junior and/or new developers to the 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 DDCs. When the database model changes (e.g., database concepts are added or deleted), maintenance engineers can also search for software that operate on related concepts or specific database concepts and modify those pieces of code. Finally, technical leads and project managers can use the tool to find DDCs that are most frequently accessed by the software to optimize the database.

VI. Evaluation

In this section, we provide a case study on industry projects, feedback from industry software engineers, and a feature comparison.

A. Case Study on Aramco Software Projects

In our case study, we selected software projects that are implemented in various technologies, such as iBatis/myBatis mapper framework [4], Java Remote Method Invocation [39], Spring Framework [6], and Adobe Flex [33]. While these technologies use Java as a programming language, these technologies also use configuration files and SQL statements as part of their implementation. The baselines are selected because they are an appropriate alternative for the given tasks. As previously mentioned, traceability techniques do not currently support connecting DDCs and implementation files. Mapping DDCs to features is also not straightforward, making it difficult to use a feature location technique as a baseline. We performed the following tasks:

1) Given a set of DDCs, find the relevant Java files

In this task, we compared the results of our tool with a proprietary tool called “Find it EZ,” which performs string and regular expression matching on a directory of files specified by a user [3]. We ran single DDC queries five times and multiple DDC queries five times. We ran the query on two projects, which consist of 763 SQL statements, 687 configuration files, 984 Java files. With regards to accuracy, both DDC Tracer and Find it EZ retrieved correct SQL statements that contain the DDCs that were queried. Find it EZ was not able to find...
related source code because they do not contain the exact DDC text. This limitation is similar to other text search tools.

2) Given a Java source code, return the relevant DDCs

In this task, our baseline is a similarity metric based on similar words between Java source code and a SQL statement. We created a word-document matrix where each Java source code file is associated with DDCs that are present within the file. We then compared the results of our baseline with our tool. We analyzed 32 business logic Java source code from three software projects, with each file averaging 209 lines of code. Among these files, the baseline technique was not able to detect related DDCs for 19 files, but our technique was able to return at least one correct DDC for each of the 32 files.

3) Find closely related Java code

In this task, our baseline is a similarity metric based on similar words between Java source code (with stop words removed). DDC Tracer, meanwhile, identified closely related Java code based on the number of similar DDCs that the files accessed. In this task, our technique is able to detect more semantically-related code than textual similarity techniques. For example, our baseline detected that there is a close similarity between File3 and File2, whereas DDC Tracer detected a close relationship between File3 and File9.

File3 has the following functionalities: getERigs, getERigsByDates, getDrillActivities, getEActivitiesByDates.

File2 has the following functionalities: getReportByDates, getWellList, getReportList, getTasksList, getTaskByPK, updateAllTasks, deleteTasks, getTaskTrackingList, updateAll, addWell, getUAPList, sendMail.

File9 has the following functionalities:getEWellSiteRigs, getWellSiteActivities.

Based on these method names, we see that File3 is indeed semantically closer to File9 than File2.

B. Feedback from Aramco Software Engineers

We also solicited the feedback of eight system analysts from the company. The range of experience of these engineers is from 2 months to 15 years and six out of the eight 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. We then asked their feedback regarding our research questions:

Q1: Does DDC Tracer facilitate finding SQL statements and source code that could be reused for another project?

Q2: How does DDC tracer compare to the current process of searching for SQL statements and source code to reuse?

Q3: Are software engineers willing to use DDC Tracer?

1) Results

Q1: 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.

Q2: Five out of the eight software engineers state that using the DDC Tracer is much easier than their current process for 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.

Q3: 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). The rating of 3 came from an engineer who has not searched SQL statements or source code before.

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 industry where layered applications are developed.

C. Feature Comparison

Table 1 shows a feature comparison between our approach and existing techniques in the literature. The “possible” notation in the table indicates additional steps or information may be required by the technique to support the feature.

TABLE I. FEATURE COMPARISON

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace DDC to SQL statements</td>
<td>Yes</td>
<td>Yes</td>
<td>Only search within a database</td>
<td>Possible</td>
</tr>
<tr>
<td>Trace DDC to source code</td>
<td>Yes</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
</tr>
<tr>
<td>Trace DDC to various configuration files</td>
<td>Yes</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
</tr>
<tr>
<td>Create traceability link chain from DDC to UI</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
VII. CONCLUSION

In this paper, we presented DDC Tracer, a data-centric traceability approach that facilitates code reuse in a fairly straightforward manner. By focusing on DDCs, we can find similar data concepts at the database layer, identify code that accesses and renders specified data concepts, and trace across the various layers of software regardless of the programming language or technology used. We evaluated our approach using an industrial case study that includes feedback from software engineers. We also compared DDC Tracer with existing techniques in the literature. The results of these evaluations indicate that DDC Tracer is effective in facilitating reuse and is feasible to use in practice.

Future avenues of work include analyzing the SQL statement structure, ranking the relevant traceability link chains, and creating visualization tools. We also plan to conduct further user evaluations to improve our technique.

ACKNOWLEDGMENT

We thank the Aramo software engineers for feedback. We are also grateful to Hussein Al-Helal’s suggestions and support.

REFERENCES