UNIVERSITY OF CALGARY

Test-Driven Reuse:

Improving the Selection of Semantically Relevant Code

by

Mehrdad Nurolahzade

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

April, 2014

© Mehrdad Nurolahzade 2014

Abstract

Test-driven reuse (TDR) proposes to find reusable source code through the provision of test cases describing the functionality of interest to a developer. The vocabulary and design of the interface of the function under test is used as the basis of selecting potentially relevant candidate functions to be tested. This approach requires that the searcher know—or correctly guess—the solution's interface vocabulary and design. However, semantic similarity neither implies nor is implied by syntactic or structural similarity. According to empirical studies, behaviourally similar code of independent origin can be syntactically very dissimilar.

We believe test cases that exercise a function provide additional facts for describing its semantics. Therefore, the thesis of this dissertation is that by modelling tests—in addition to function interfaces—the odds of finding semantically relevant source code is improved. Additionally, to facilitate similarity comparisons and improve performance, we propose a multi-representation approach to building a reuse library. To validate these claims we created a similarity model using lexical, structural, and data flow attributes of test cases. Four similarity heuristics utilize this model to independently find relevant test cases that exercise similar functionality. We developed a proof of concept TDR tool, called *Reviver*, that finds existing test cases exercising similar functions once given a new set of test cases. Using *Reviver* a developer writing tests for a new functionality can reuse or learn from similar functions developed in the past.

We evaluated the effectiveness of *Reviver* in a controlled study using tasks and their manually generated approximations. We experimented with different configurations of *Reviver* and found that overall the combination of lexical and data flow similarity heuristics is more likely to find an existing implementation of the function under test. Our results confirm that lexical, structural, and data flow facts in the test cases exercising a function—in addition to the interface of function under test—improve selection of semantically relevant functions.

Acknowledgements

I would like to express my gratitude to my supervisors Dr. *Frank Maurer* and Dr. *Robert J. Walker* for their guidance, encouragement and support through out the process. I would also like to extend my appreciation to Dr. *Thomas Zimmermann* and Dr. *Jörg Denzinger* for their helpful advice. Without your insights, this would have very likely been a different thesis.

I would like to thank the members of the Laboratory of software Modification Research (LSMR) and Agile Surface Engineering (ASE) group for the fruitful discussions and support. Special thanks to *Soha Makady*, *Hamid Baghi*, *Rylan Cottrell*, *Brad Cossette*, *David Ma*, *Seyed Mehdi Nasehi*, *Valeh H. Nasser*, *Elham Moazzen* and *Yaser Ghanam* for their comments, feedback and suggestions over the years.

This work was supported partly by scholarships and grants from NSERC and IBM.

Dedication

To Mom and my wife Sepideh, your love made me who I am.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

I don't know exactly where ideas come from, but when I'm working well ideas just appear. I've heard other people say similar things - so it's one of the ways I know there's help and guidance out there. It's just a matter of our figuring out how to receive the ideas or information that are waiting to be heard.

Jim Henson

1.1 Software Reuse and Testing

Software reuse is the use of existing software, or software knowledge, to build new software. Development effort can be reduced by reusing existing software while the quality of production can be increased by using mature previously tested assets (*Basili et al.*, 1996). Programmers have always pragmatically reused[1] sections of code, functions, and procedures. Such source code, that is not necessarily designed in a reusable fashion, might be acquired from the internal organizational repositories or any where on the Internet. With the rise of the open source movement since the late 1990s, a rapidly growing quantity of publicly available source code has become available on the Web. With hundreds of millions of files available in online repositories, building

[1]Reusing source code that was not designed in a reusable fashion has been known by many names: code scavenging (*Krueger*, 1992), ad hoc reuse (*Prieto-Díaz*, 1993), opportunistic reuse (*Rosson and Carroll*, 1993), copy-and-paste reuse (*Lange and Moher*, 1989), and pragmatic reuse (*Holmes*, 2008).

search platforms that allow efficient retrieval of source code is an essential step in enabling code reuse.

Using tests as a means of expressing the function and behaviour of a system is a common practice in many agile development processes. In test-driven development (TDD) the implementation of an improvement or a new feature is led by writing tests (*Beck*, 2002). The process is followed by producing minimal amount of code to pass the tests and eventually refactoring the new code to meet the production standards. The cycle is repeated in small increments until the desired new behaviour is achieved. The proponents of the test-driven development paradigm claim that it improves the design of the system in addition to its testability (*Marchesi et al.*, 2003).

As test-driven development has gained in industrial popularity, the prospect of utilizing test cases as the basis for software reuse has become tantalizing: test cases can express a rich variety of structure and semantics that automated reuse tools can potentially utilize. TDR search queries are automatically generated from test cases written by developers practicing TDD. Such developers are obligated to write tests before they develop functional code. Tests also provide an automated means of verification of search results. Potential candidates are automatically adapted, compiled and tested[2] by the TDR system. Candidates that pass the test cases are presented to the developer as potential solutions. By reducing the effort required for locating and verifying reusable source code, TDR seeks to make reuse an integrated part of the software development process.

[2]Compiling and running arbitrary source code retrieved on the Internet cannot always be fully automated. For further details see the discussion in Section 2.2.2.

1.2 Test-Driven Reuse

As demonstrated in Figure 1.1, existing test-driven reuse (TDR) systems (*Hummel and Janjic*, 2013; *Lazzarini Lemos et al.*, 2011; *Reiss*, 2009a) follow a four step process: (1) a developer has to write code for a function that he suspects might have been developed in the past; (2) she writes a minimal test that describes the desired behaviour of the function sought after; (3) a number of candidates that potentially match the specification of the function exercised in the supplied test are selected from a source code repository; (4) the test is used to verify candidates. If any of the retrieved candidates passes the test it is recommended to the developer as a potential solution. Otherwise, the developer can continue the search by modifying the test.

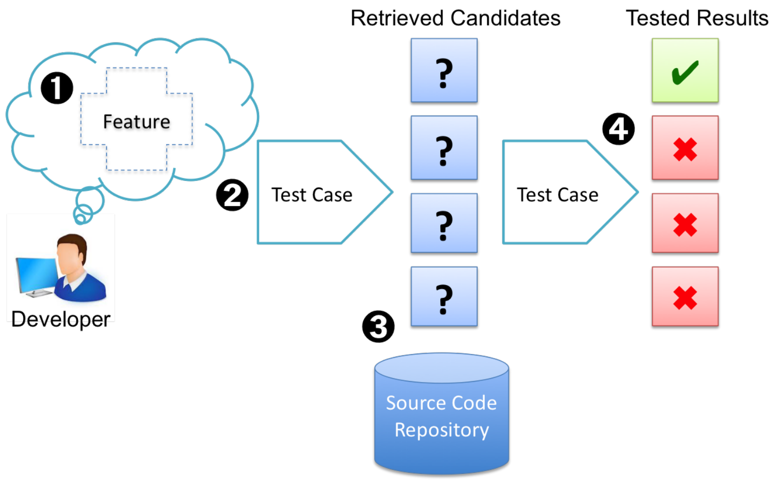


Figure 1.1: A graphical representation of the test-driven reuse cycle; the query test cases might be refined and the search be repeated until appropriate source code is found (or the searcher gives up).

Unit tests produced in TDD are typically written in *XUnit*[3] frameworks. Exist-

[3] *XUnit* is the family name given to a set of testing frameworks that have become widely known

ing TDR approaches select relevant source code by extracting the interface of the function under test from query *XUnit* test cases and use it to perform an interface-based search[4] over a library of existing entities. And therein lies the crux of the potential problem: in TDD, the developer may have at best a fuzzy notion (initially) of the functionality she wants. If a TDR approach places too much weight on the details of the test cases written (such as the particular names of types and methods), it is unlikely to find appropriate source code nor source code that can be trivially transformed to become a perfect match for those test cases.

1.3 Motivational Example

To demonstrate the potential limitations of the interface-based approach in finding relevant source code, consider the unit test in Figure 1.2. The system under test in the given scenario is the Account class and the action being validated is the transfer() method at line 18. Fixture setup and precondition verification takes place in lines 6-16 and the post condition verification takes place by the assertions in lines 20-22.

An existing test-driven reuse system would realize that the developer is looking for the Account class. It would then extract the interface of the Account class (i.e., **interface** Account {Double getBalance(); Transaction transfer(Double, Account);}) from the test and perform an interface-based search to find classes matching this interface in the reuse library. Obviously such a search query only includes a subset of the information in the test case in Figure 1.2. Other essential facts like the pre- and post-conditions of the transfer action, the presence of other collaborator classes like

amongst software developers. The name is a derivation of *JUnit* the first of these to be widely known (*Fowler*, 2013).

[4]The combination of signature and name-based retrieval (*Hummel et al.*, 2007).

```
public class AccountTest
{
  @Test
  public void testValidTransfer()
  {
    Account from = new Account();
    Account to = new Account();
    Bank bank = Bank.getInstance();

    bank.register(from);
    bank.register(to);

    Double fromBalance = from.getBalance();
    Double toBalance = to.getBalance();

    assertNull(bank.getLastTransaction());

    Transaction t = from.transfer(100.0, to);

    assertEquals(fromBalance − 100.0, from.getBalance(), 0.01);
    assertEquals(toBalance + 100.0, to.getBalance(), 0.01);
    assertEquals(t, bank.getLastTransaction());
  }
}
```

Figure 1.2: A *JUnit* test case for testing the fund transfer functionality in a bank.

Bank, and their interactions with the Account class are not taken into consideration. Furthermore, a matching solution AnotherAccount in the reuse library—that can satisfy the testValidTransfer() test case—is not considered as a candidate if it does not match the expected interface. In other words, the relevance criteria is restrictive and rejects a class that might have been implemented using an alternative vocabulary and design.

1.4 Finding Source Code for Reuse

Software developers use open source code available online in their work. Studies have found code duplication across open source projects to be a common phenomenon (*Schwarz et al.*, 2012); hence, suggesting that developers often pragmatically reuse existing source code in absence of appropriate reusable components. Software developers also look upon information and code snippets on the Web and online forums as a learning resource that assists with solving problems (*Gallardo-Valencia and Sim*, 2013; *Barua et al.*, 2012). Evidence of this practice can be found in the number of project hosting sites, code repositories, developer forums, and source code search engines that have appeared (*Umarji et al.*, 2008).

1.4.1 Code Search Engines

Developers often use general search engines such as *Google* for finding reusable source code. While these tools are effective in retrieving documents they can only be used for code searches if the functionality is defined by well-known terms. Code search engines slightly improve this situation by indexing source code based on a number of attributes, e.g., programming language, class, function, file, and license. Nevertheless, source code is more or less treated as a bag of words and the structure and semantics of it are largely ignored. As a result, similar to general search engines, code search engines can only be effective in search scenarios where the program vocabulary is known in advance or not difficult to guess. Code-specific search engines have been found to be more suitable for more coarse grained software searches like libraries rather than blocks of functionality (*Sim et al.*, 2011). Finding smaller chunks of functionality—in which program vocabulary can at best be guessed—is still a challenging task for code search engines.

Some recent code search engines, such as *Sourcerer*[5] (*Linstead et al.*, 2009), index structural relationships between elements in source code. Although structured retrieval improves search effectiveness, semantic similarity neither implies nor is implied by structural similarity. Current code search engines provide searches based on syntactic and structural attributes of software artifacts. Retrieving a function that matches a given behavioural specification—for instance a test case—remains beyond what today's code search engine offer.

1.4.2 Finding Examples

Relevant source code cannot always be integrated into the system; it might not exactly provide the sought after functionality, or the effort required to adapt and reuse it is not justifiable (i.e., costs outweigh benefits), or licensing and quality limitations might prevent the developer from reusing it as-is. However, relevant source code can still serve as a reference example that provides knowledge to the developer to build functionality they need. Developers also look up existing source code snippets to learn about an API, framework, or a programming language (*Umarji et al.*, 2008). Using current code search engines, finding reference examples that can help with development can be much easier than finding source code that can be reused as-is (*Sim et al.*, 2011).

Several code snippet retrieval systems have appeared in the code search and reuse literature (*Holmes et al.*, 2005; *Mandelin et al.*, 2005; *Sahavechaphan and Claypool*, 2006; *Xie and Pei*, 2006; *Bajracharya et al.*, 2010a). These approaches use the de-

[5]http://sourcerer.ics.uci.edu/sourcerer/search/index.jsp

velopment context to look up and suggest potentially related code snippets. The developer can then evaluate a recommended code snippet, learn how it works, and copy and paste it into their own work. Example recommender systems are not a replacement for reuse systems as they are only effective when retrieving source code that uses similar APIs. A TDR system, on the other hand, has to provide recommendations that can pass searcher test cases no matter what underlying APIs are used in them.[6]

1.5 An Alternative Approach to Test-Driven Reuse

After carefully examining existing code search and retrieval techniques, including interface-based retrieval, we decided to pursue the goal of creating an improved approach for selecting relevant source code in a test-driven reuse system. Thus, we decided to investigate which facts from test cases—beyond the interface of the system under test—can be utilized in the TDR selection step and whether they can improve the relevance of the retrieved source code.

1.5.1 A Multi-Representation Reuse Library

The selection step in software reuse and the choice of underlying representations become more important with the growing size of the reuse library (*Frakes and Pole*, 1994). Assets in a reuse library are stored according to a representation scheme. The choice of the representation scheme and associated indexes determine the range of operations that can be performed on the library and the overall efficiency of the reuse

[6]Ideally, TDR users should be able to select the APIs in the implementation of a function if desired.

selection process (*Frakes and Gandel*, 1989). Furthermore, there is empirical evidence that different representation methods can complement each other by retrieving different items (*Frakes and Pole*, 1994). Hence, to improve the performance of the TDR system, we propose a multi-representation scheme where each representation efficiently indexes a different aspect of tests.

We propose to process test cases at index time and model their features in the reuse library using lexical, structural, and data flow representations. The lexical representation features the names of classes, objects, and methods in the test case; the structural representation includes all types and the methods invoked on them; and finally, the data flow representation is a graph of data dependencies between methods invoked. Figure 1.3 demonstrate the three representations of the fund transfer test case in Figure 1.2.

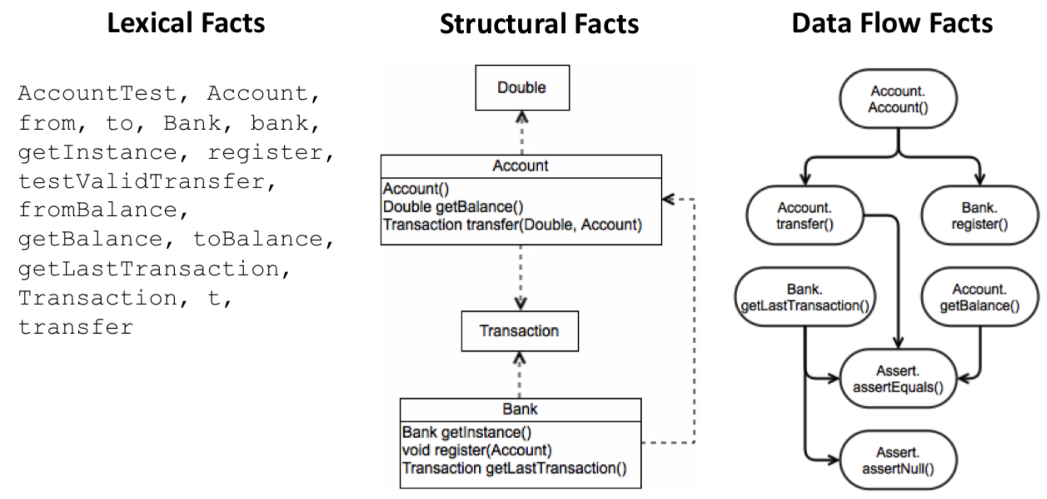

Lexical Facts
AccountTest, Account, from, to, Bank, bank, getInstance, register, testValidTransfer, fromBalance, getBalance, toBalance, getLastTransaction, Transaction, t, transfer
Structural Facts
Double
Account
Account()
Double getBalance()
Transaction transfer(Double, Account)
Transaction
Bank
Bank getInstance()
void register(Account)
Transaction getLastTransaction()
Data Flow Facts
Account. Account()
Account. transfer()
Bank. register()
Bank. getLastTransaction()
Account. getBalance()
Assert. assertEquals()
Assert. assertNull()

Figure 1.3: The three representations of the fund transfer test case in Figure 1.2.

1.5.2 Test Case Similarity As a Measure of Relevance

Existing TDR approaches select relevant source code by extracting the interface of the entity under test from the query test cases and use it as the basis for performing an interface-based search over a catalogue of existing entities. Inspired by case-based reasoning (CBR)[7], we propose a new approach for utilizing test cases to find and reuse existing source code. Figure 1.4 demonstrates the workflow of our proposed approach. We take test cases exercising a system to be a partial description of the problem solved by the system under test. Collecting and indexing test cases and the systems they test in a repository is analogous to building a case library of programming problems described as test cases. A new problem—formulated as new test cases—can then be matched with existing similar problems (i.e., old test cases) to recommend a potential solution (i.e., an existing system).

Our test similarity model uses similarity heuristics operating on the lexical, structural, and data flow facts extracted from test cases to find existing test code that exercise similar entities. Each similarity heuristic operates independent of the other heuristics and returns a result list. Recommendations are made to the developer based on the aggregated similarity score of the heuristic results. We built *Reviver*, a proof of concept prototype based on the *Eclipse* integrated development environment (IDE) that indexes test cases and represents them using text search and relational database platforms[8]. The repository of the prototype solution was populated with

[7]Case-based reasoning (CBR) (*Althoff et al.*, 1998) is a problem solving paradigm based on the solutions of similar problems in the past. A case library is built of past problems and their solutions; to solve a current problem: the problem is matched against the cases in the library, and similar cases are retrieved. The retrieved cases are used to suggest a solution. The solution might then be revised, if necessary. CBR has been applied to software engineering problems in the past, including cost and effort estimation, quality prediction, and software reuse (*Shepperd*, 2003).

[8]A graph database could have been an alternative platform for implementing the graph-based

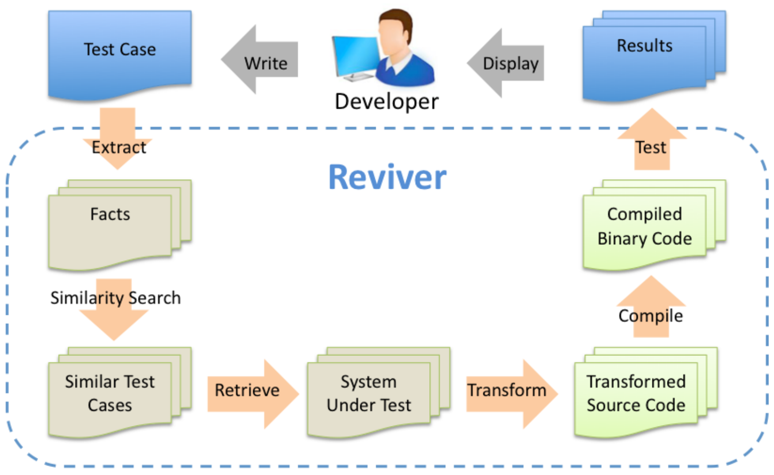


Figure 1.4: An alternative approach to selection of relevant source code in test-driven reuse. Test similarity is used to find similar test cases in the reuse library. The system under test is returned as relevant source code. The rest of the reuse process is similar to existing test-driven reuse systems.

a selection of *Java* open source projects with *JUnit* tests. A controlled experiment produced evidence that using more facts in the test cases improves the precision of a test-driven reuse system in identifying the function under test.

1.6 Thesis Statement

The thesis of this dissertation is that by modelling tests—in addition to function interfaces—the odds of finding semantically relevant source code is improved.

1.7 Outline

Chapter 2 provides an overview of related work. Chapter 3 reports on an evaluation study we conducted on current TDR approaches. We provide an analysis of the

portion of our test similarity model.

shortcomings and underlying problems in existing approaches, and a discussion of potential solutions. The description of our test similarity model and representation methods is presented in Chapter 4. An overview of the *Reviver* proof of concept prototype and evaluation study is given in Chapters 5 and 6 respectively. In Chapter 7 we describe our proposed ideas for future work to extend TDR. First, we describe a technique for extending interface-based retrieval. Then, we describe a pattern-based retrieval technique for finding assets based on their structural and behavioural characteristics. Chapter 8 provides a discussion and suggestions on the research directions that could follow this work. Chapter 9 concludes this thesis with a short summary.

Chapter 2

Related Work

Test-driven reuse is a fairly recent approach to software reuse. The idea of utilizing tests for software retrieval was first proposed by (*Hummel and Atkinson*, 2004). Test-driven development that is related to test-first programming was first introduced by extreme programming (*Beck*, 2000) in 1999. On the other hand, software reuse has a well-established history in both research literature and industrial practice. Due to the broad scope of software reuse, relevant research is found in many different fields including software engineering, programming languages, information retrieval, program understanding, and many fields that utilize domain-specific reuse mechanisms. Research efforts relevant to this thesis have been organized into six primary categories: pragmatic software reuse, test-driven reuse, design for reuse, software recommender systems, source code retrieval, and test similarity. This chapter outlines relevant related work and differentiates the research in this dissertation from previous research efforts.

2.1 Pragmatic Software Reuse

Reusing source code that was not designed in a reusable fashion has been known by many names: code scavenging (*Krueger*, 1992), ad hoc reuse (*Prieto-Díaz*, 1993), opportunistic reuse (*Rosson and Carroll*, 1993), and copy-and-paste reuse (*Lange* *and Moher*, 1989). While research has stated that pragmatic reuse can be effective (*Krueger*, 1992; *Frakes and Fox*, 1995), little research has been performed to identify how industrial developers reason about and perform these tasks. Unlike planned reuse in which a library of reusable assets is maintained within the organization, pragmatic reuse tasks are more opportunistic. A developer simply decides that they want to reuse some existing functionality, regardless of whether it has been designed in a reusable fashion or not, and performs the reuse task manually. Pragmatic reuse tasks can be of any size, but are typically limited by the developers ability to fully understand their reuse task; as such, they tend not to be as large as the largest black-box tasks, and are particularly suited for medium-scale reuse tasks (*Holmes*, 2008).

There are several aspects of pragmatic reuse that make it a difficult problem. For example, a factor that can limit opportunistic reuse of source code online is the quality requirements. A number of approaches have been proposed for measuring trustability of code search results (*Gysin*, 2010; *Gallardo-Valencia et al.*, 2010). These metrics assist the developer with risk-cost-benefit analysis they undertake to find suitable integration candidates. Another factor is the problem of finding source code to reuse. This problem has been tackled in various ways, but there has not been a comprehensive solution that has really worked. Hence, despite the high availability of open source code on the Internet, pragmatic reuse is rather limited, due to the fact that equivalent code is difficult to find (*Reiss*, 2009a).

2.2 Test-Driven Reuse

Podgurski and Pierce (*Podgurski and Pierce*, 1992) proposed behaviour sampling, an approach to retrieve reusable components using searcher-supplied samples of input/output for desired operations. As a pre-filtering step, behaviour sampling uses signature matching to limit the components to be tested. However, the signature matching techniques can deliver thousands of candidates for generic signatures, e.g., well-known data structures like stack. A rigid signature matching technique, on the other hand, will not deliver any recommendations if it cannot find sufficiently similar interfaces amongst software assets in the repository. We will further discuss in Chapter 3 that in general the expectation that the searcher would know the interface of the functionality beforehand is not realistic. A signature or interface based search would not yield relevant results unless part of the desired program interface is known—or correctly guessed.

2.2.1 Existing Test-Driven Reuse Approaches

(*Podgurski and Pierce*, 1992) also proposed extensions to the classic form of behaviour sampling. Test-driven reuse realizes one of these extensions that permits the searcher to provide the retrieval criteria through a programmed acceptance test. Three approaches to test-driven reuse have appeared in the literature: *Code Conjurer* (*Hummel and Janjic*, 2013), *CodeGenie* (*Lazzarini Lemos et al.*, 2011), and *S6* (*Reiss*, 2009a). The prototype tool for each approach operates on source code written in the *Java* programming language.

Table 2.1 provides a comparison of these approaches. In terms of commonality, each tool: (a) represents source code as a collection of terms; (b) uses existing (code)

Attribute	*Code Conjurer*	*CodeGenie*	*S6*
Code representation	Bag of words	Bag of words	Bag of words
Search engine	*Merobase* (based on *Apache Lucene*)	*Sourcerer* (based on *Apache Solr*)	Remote code search engines, local code search engine (based on *Compass*), or both
Unit of retrieval	One main source file and other referenced source files	One method and dependent code	One class or method (that has no external dependencies)
Relevance criteria	Linguistic (type and method names) and syntactic (operation signatures)	Linguistic (type and method names) and syntactic (operation signature)	Linguistic (user supplied terms) and syntactic (operation signatures)
Matching criterion	Passing test cases	Passing test cases	Passing test cases
Ranking function	Weighted sum on linguistic features	Combination of a weighted sum on linguistic features and *Code-Rank* (*Linstead et al.*, 2009)	Code size, complexity, or run-time speed
Query relaxation	Split class name, use class name only, or ignore method names	Ignore names, return type, or argument types	Uses transformations (*Reiss*, 2009a) like name, return type, or argument type refactoring to expand the potential pool of solutions
Dependency resolution	JDK level dependencies and project library lookup	JDK level dependencies, project library and Maven central repository lookup (*Ossher et al.*, 2010)	JDK level dependencies
Code slicing	N/A	Computes a transitive closure of type/field accesses and method calls that are required to run the retrieved method	Computes the subset of a method/class that provides the desired functionality
Test execution	Server-side	Client-side	Server-side
Adaptation and integration support	*ManagedAdapter* (*Hummel and Atkinson*, 2009), parameter permutator (*Hummel*, 2008)	Merge by name (*Tarr and Ossher*, 2001)	Uses transformations to adapt search results to searcher specifications

Table 2.1: A comparison of the three test-driven reuse prototypes: *Code Conjurer*, *CodeGenie*, and *S6*.

search engines as the underlying search providers; (c) retrieves relevant candidates via lexical similarity enhanced by signature matching; (d) relaxes query constraints if enough candidates cannot be retrieved otherwise; (e) requires test cases to pass; and (f) attempts to adapt and resolve the external dependencies of the candidate source code prior to executing tests.

In terms of differences, each tool: (a) utilizes a different underlying search engine; (b) retrieves different extents of code; (c) applies a different ranking function; (d) tries different query relaxation approaches; and (e) tries a different approach for adaptation/integration of retrieved candidates. In addition, *CodeGenie* and *S6* slice the candidate source code to obtain the subset that is required to run the test cases, whereas *Code Conjurer* does not.

Code Conjurer and *CodeGenie* are *JUnit*[1]-based implementations of test-driven reuse. The plug-in to the integrated development environment (IDE) provided by each of the tools automatically extracts operation signatures from the searcher-supplied *JUnit* test code. Search is then performed via a source code search engine and results are presented to the searcher for inspection in the IDE. The *Merobase* and *Sourcerer* (*Linstead et al.*, 2009) code search engines power *Code Conjurer* and *CodeGenie* searches respectively. *CodeGenie* further assists the searcher in slicing the source code to be reused; however, unlike *Code Conjurer*, the current implementation of *CodeGenie* can only be used to search for a single missing method, and not an entire class (*Lazzarini Lemos et al.*, 2011).

S6 complements the use of behaviour sampling with other forms of semantic specifications. It implements the abstract data type extension proposed by (*Podgurski*

[1] *JUnit* is an automated testing framework for *Java* code.

and Pierce, 1992) to handle object-oriented programming. Unlike *Code Conjurer* and *CodeGenie*, *S6* does not utilize *JUnit* test code as a query language, but requires that the searcher provide the interactions of a class's methods through its web user interface. *S6* attempts a small set of limited transformations on candidate source code, in an attempt at relaxing the constraints imposed by literal interpretation of the input query. *S6* can use a local repository or remote code search engine like Sourcerer as the codebase searched through for candidate source code. Similar to *Code Conjurer* and *CodeGenie*, *S6* depends on being able to find an appropriate initial result set.

The overall pipe-line architecture realized by the existing TDR approaches is demonstrated in Figures 2.1. All three approaches initially filter the repository on the basis of lexical or syntactic similarity with the supplied test case/query specification. All three try to execute the test case on each of the filtered results to assess semantic similarity as well; this acts solely to further constrain the set of potential matches.

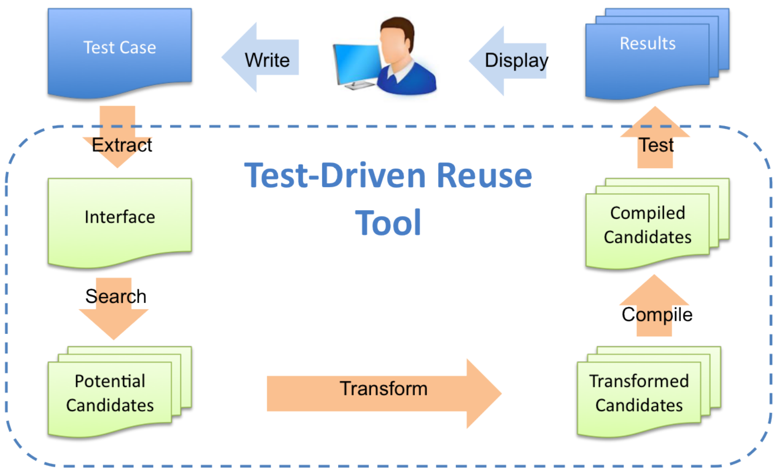


Figure 2.1: The pipe-line architecture of the test-driven reuse process as realized by existing approaches.

2.2.2 Problems in Test-Driven Reuse

We remain convinced that TDR is a promising way to enable source code reuse, however a number of difficult research problem have to be overcome before it can be adopted by industrial developers. In a qualitative study of existing TDR approaches (*Nurolahzade et al.*, 2013), we discovered that current TDR prototypes have difficulty recommending non-trivial functionality—like that needed in the daily tasks of developers. The interface-based retrieval mechanism employed by these approaches limits how relevant source code is selected. It requires the searcher to know—or correctly guess—the solution vocabulary and its design. Finding relevant source code is not the only difficult problem in test-driven reuse; automated adaptation and integration of source code is a prerequisite of running test cases. Adapting, compiling, and running arbitrary source code found on the Web are ongoing areas of research in software engineering.

Finding Relevant Source Code

Similarity of programs is evaluated in many contexts, including detecting code clones (*Gabel et al.*, 2008; *Juergens et al.*, 2010), detecting plagiarism (*Liu et al.*, 2006), finding examples (*Holmes et al.*, 2006; *Bajracharya et al.*, 2010b), and finding source to reuse. Although there is no consensus in the research community about the definition of similarity between programs, different similarity types are generally believed to belong to the dual categories of representational and semantic (or behavioural) similarity. Representational similarity refers to lexical, syntactic, and structural similarity. Semantic similarity, on the other hand, can be defined in terms of the function or behaviour of the program (*Walenstein et al.*, 2006). Unlike representational similarity for which various detection techniques has been devised, few studies have targeted semantic similarity where representational similarity may not necessarily be present.

Theoretically, a TDR system could retrieve semantically similar code by verifying its behaviour through searcher supplied test cases. However, current TDR approaches draw on the representational similarity of the function under test in the developer supplied test cases and source code in the reuse repository to select relevant source code (*Nurolahzade et al.*, 2013). Behaviourally similar code of independent origin is not guaranteed to be lexically[2] or structurally similar. In fact, it can be so syntactically different that representational similarity approaches are unlikely to identify it (*Juergens et al.*, 2010). One may argue that it is unrealistic to expect any TDR approach to not depend on the presence of specific vocabulary. If one works in a context where a domain is well-established, for instance within a specific organization or while utilizing a specific application programming interface, this dependency could even be reasonable. However, the desire to find useful functionality in the absence of known vocabulary is industrially reasonable and should not be dismissed.

Adapting and Compiling Source Code

Source code retrieved from the internet might be incomplete and miss dependencies—in the form of sources or libraries—that are required to statically or dynamically analyze it. A number of techniques have been proposed by the research community to resolve some of the external dependencies in source code (*Ossher et al.*, 2010). To optimize the size of the resulting code, resolved dependencies are sliced to a minimal set of resources that can be compiled and executed properly (*Ossher and Lopes*, 2013).

[2]The vocabulary problem also known as vocabulary mismatch problem, states that on average the chance of two domain experts using the same words to name things is only 10-15% (*Furnas et al.*, 1987).

If retrieved source code is inconsistent with the developer supplied test cases then its interface has to be adapted accordingly (*Reiss*, 2009a). Alternatively, an adapter may wrap the reusable source code with the appropriate interface in order to make it compilable and executable (*Hummel and Atkinson*, 2010). However, both techniques are rather limited and can only fix instances that require minor adaptation. In reality, such triage decisions can at best be semi-automated for non-trivial reuse tasks using current state of the art (*Holmes*, 2008).

Running Source Code

Even if source code can be compiled, there is still no guarantee that it can be run. Running programs may require provision of specific runtime environments or resources. For example, web-based and mobile components require specific containers to run. A database or network component requires those external resources to deliver its services. Even if one manages to have source code run automatically, it may not behave as expected because it requires special configuration that is described in the accompanying documentation. To improve automated analysis of source code, researchers and software professionals should look into the problem of making source code and projects more code search engine friendly. Static and dynamic analysis of large repositories of source code is not possible unless developer intervention in the process is kept to a minimum.

2.3 Design for Reuse

A type of reuse—that is not covered by this thesis—is planned reuse or design of software components so that they be reused in future projects. Component-based software engineering (CBSE) is a reuse-based approach to developing a software system as a collection of independent components. Software libraries (APIs), frameworks, and web services are examples of software components that encapsulate a set of related functions and data. One of the main objectives of CBSE is reusing software components in multiple systems. Since the 1990s, the growing availability of commercial-off-the-shelf (COTS) components in the software market has been a driver for research on location and selection of the components that best fit the requirements. Keyword searches on component descriptions (*Maarek et al.*, 1991), faceted classification (*Prieto-Diaz*, 1990), domain ontologies (*Pahl*, 2007), signature matching (*Zaremski and Wing*, 1997), formal (*Zaremski and Wing*, 1997) and behavioural specifications (*Hashemian*, 2008) are all examples of retrieval approaches proposed for component libraries (*Lucredio et al.*, 2004).

Signature matching—a technique also used in TDR approaches—has also been applied to the web service composition problem where a composite web service is built by automatically assembling a number of existing web services. In (*Hashemian*, 2008), a behavioural specification is provided along with each web service in the repository that is utilized along with the component signature in order to find and assemble the right components together. A formal specification of component behaviour along with signatures were used in (*Zaremski and Wing*, 1997) to retrieve matching components. Formal specifications were written in the form of pre and post condition predicates. A theorem prover determines wether a component specification matched query requirements.

Software product lines (SPL) (*Krueger*, 2004) are another approach to planned reuse in which a shared set of assets are used to create a collection of software systems. Unlike opportunistic approaches to reuse, software product lines only create a software asset in reusable fashion if the asset can be reused in at least another product. This thesis focuses on the topic of opportunistic reuse of source code in organizational or online repositories that is not necessarily designed in a reusable fashion. Arbitrary source code retrieved online is not accompanied by semantic behavioural models. Hence, retrieval and composition strategies utilized in reuse of reusable software components cannot be utilized by TDR approaches.

2.4 Software Recommender Systems

Software development involves dealing with large code bases and complex project artefacts. In addition, developers need to keep up-to-date with the latest changes in software development technologies through constant learning. Unless appropriate assistance is provided to them, it is easy to get bogged down in details seeking the right information (*Robillard et al.*, 2010). Software recommender systems come in handy to inexperienced developers or when the information space grows far beyond the ability of developers to consider everything. *Hipikat* (*Čubranić et al.*, 2005) is a good example of a software recommendation system that uses a broader set of project information sources including source code, documentation, bug reports, e-mail, newsgroup articles, and version information. It provides recommendations about project information a developer should consider during a modification task. (*Ying et al.*, 2004) and (*Zimmermann et al.*, 2004) apply association rule mining on software change history records extracted from version control system. The developer is warned about potentially missing changes to (based on mined association rules) when trying to commit a change-set.

There are various forms of software recommender systems. Here we discuss three of them that we find related to this dissertation.

2.4.1 Finding Example Source Code

Developers who are new to an API sometimes find themselves overwhelmed when they try to accomplish development tasks. Developers often learn to work with unfamiliar APIs by looking at existing examples. *Strathcona* (*Holmes et al.*, 2006) recommends pieces of code that demonstrate how to use an API. It uses structural context attributes (e.g., parents, invocations, and types) in the code under development to create a set of heuristics. Heuristics are then used to query the code repository, returning snippets of code with similar structural context. *MAPO* (*Xie and Pei*, 2006) clusters examples based on their API usage. Frequent API call sequences are then mined from code snippets in each cluster. Code snippets are recommended by comparing API usage in development context with that of the identified patterns. (*Bajracharya et al.*, 2010b) use structural semantic indexing (SSI) to capture word associations in order to retrieve examples from a code repository. The SSI argument is that code snippets that share APIs are likely to be functionally similar; therefore the terms used in them are likely to be related.

Unlike example retrieval systems, a TDR system is not bound by a specific API, framework, or technology platform. However, TDR systems can be extended to—optionally—include facts about the current development context in their queries. Thereby, they would be able to retrieve other test cases syntactically, structurally, or semantically similar to the current test case. Inside organizational boundaries where domain-specific software is developed, there is a chance that (partially) similar functionality is developed over time. Our test similarity model in Chapter 4 combines characteristics of examples recommenders and test-driven reuse systems. As a result, its performance is improved in the contexts that the selection of APIs is not very diverse.

2.4.2 Guiding Development

Some recommendation systems aim to improve developer productivity or detect bad usage once dealing with unfamiliar API. *CodeBroker* (*Ye and Fischer*, 2005) was one of the first proactive tools that explored this idea. It relies on developer comments and method signatures to retrieve and recommend matching components. *RASCAL* (*McCarey et al.*, 2008) uses collaborative filtering to recommend new methods to developers. It models individual API method calls as items, while method bodies are taken to be users. A program method that calls an API method is counted as a vote by a user for that item. *JADET* (*Wasylkowski et al.*, 2007) compares method call sequences with identified frequent patterns (that are taken to represent object protocols) to spot defects and code smells in developer code. Kawrykow and Robillard (*Kawrykow and Robillard*, 2009) use code similarity detection techniques to detect imitations of API method calls. The main hypothesis underlying their technique is that client code imitating the behaviour of an API method without calling it may not be using the API effectively because it could instead call the method it imitates.

These recommender systems spot similarity between current development context and that of many examples in the repository. They provide opportunities for very small-scale reuse in the form of one or more known API calls. Test-driven reuse systems, on the other hand, seek units of functionality ranging from a single method to complete components. Unless requested by the searcher, there should be no restriction on the choice of APIs used in a retrieved functionality as long as it adheres to the given behavioural specification.

2.4.3 Code Completion

Code completion systems are another form of recommender systems that are designed for integrated development environments (IDEs) and provide small-scale code reuse. *Prospector* (*Mandelin et al.*, 2005), *XSnippet* (*Sahavechaphan and Claypool*, 2006), and *ParseWeb* (*Thummalapenta and Xie*, 2007) address the situation in which a developer knows what type of object is needed, but does not know how to write the code to get the object. Code completion recommendations can disorient developers if they are irrelevant to the current development context. Recommendations can be ranked and filtered out based on the frequency of occurrence in existing code repositories and relevancy to current development context (*Bruch et al.*, 2009).

Similar to other software recommendation system, the underlying modelling techniques used by code completion can be beneficial to TDR systems. Program statement sequences and control and data flows in test cases can be analyzed to better understand the semantics of test cases. Using graphs to model code—as done by some code completion systems—is an alternative that has to be investigated by the code search and retrieval community given the latest advances in graph database technology.

2.5 Source Code Retrieval

The increasing availability of high quality open source code on the Internet is changing the way software is being developed. It has become commonplace to search the Internet for source code in the course of software development (*Gallardo-Valencia and Sim*, 2009). Developers search on-line for code to either find reusable code or reference example. A key difference between the two motivations is how much modification is expected to reuse the code. Reusable code was considered to be source code that they could slightly modify and use right away. A reference example is a piece of code that illustrates a particular solution but needs to be re-implemented or significantly modified (*Umarji et al.*, 2008).

Source code retrieval approaches can be classified based on their underlying representation schemes. Early software retrieval systems that indexed hundreds of artifacts built indexes from descriptive information provided by system administrators (*Hummel et al.*, 2013). In more recent years, with dramatic increase in the number of artifacts, automatically generated indexes have replaced bibliographic libraries of the past. Due to its inherent simplicity and better scalability and availability of various libraries and frameworks that facilitate it, representing source code as text has been and still remains the number one choice for most code search engines. Alternatively, relational databases have been used for storing structural relationships between code elements. Other more elaborate representations like representing code as graph have only been applied to small repositories. Devising a scalable approach that utilizes deeper semantics of source code remains an open problem to this date.

2.5.1 Lexical and Syntactic Approaches

Many code retrieval approaches build upon the term-based search approaches commonly used in information retrieval. Terms are generally taken from program element names (e.g., class, method, or variable), comments, and documentation. Accordingly, a search query is expressed in (or converted to) a natural language expression (*McCarey et al.*, 2008). Developers comprehend code by seamlessly processing synonyms and other word relations. Researchers have leveraged word relations in software comprehension and retrieval tools. For example, *B.A.R.T* (*Durão et al.*, 2008) enhances input queries by a domain ontology. In another research project synonyms and word relations in *WordNet* were taken into consideration in term matching (*Sridhara et al.*, 2008).

Information retrieval representation schemes such as term vector and bag of words models designed for free format text documents can also be applied to source code files. This would improve the speed of relevance matching at the cost of precision. To compensate for the loss of precision, tools might only use term-based retrieval as a first filter to shrink the candidate space considered by later, more computationally expensive, filtering stages. Unfortunately, this comes with the downside of filtering some of the relevant results in an early stage due to vocabulary mismatches. The success of term matching depends on matching the users vocabulary with that of the original programmer. To be effective, a searcher has to find terms that are unique enough to actually identify the code in mind (*Reiss*, 2009b).

Existing TDR approaches rely on underlying code search engines that represent source code files using term vector model. Relevant source files are identified through their vector distance computation from the set of terms automatically extracted from the input test cases. Hence, they are technically bound by the limitations of the term-based retrieval. If the terms are off and do not match the desired functionality, later stages down the pipeline in Figure 2.1 would not have any potentially relevant candidates to operate on.

2.5.2 Structural Approaches

Source code is different from plain text because it is structured in nature due to the fact it follows strict syntax rules specific to a programming language (*Gallardo-Valencia and Sim*, 2009). Structural-based code retrieval approaches rely on information inferred from relationships between properties (e.g. methods and objects) of a program. Such relationships include class inheritance, interface hierarchies, method invocations and dependencies, parameters and return types, object creations, and variable access within a method (*Yusof and Rana*, 2008).

Sourcerer (*Linstead et al.*, 2009) is a code search engine that provides term and fingerprint (structural metadata) searches. *Sourcerer* uses the *CodeRank* metric to rank the most frequently called/used components higher. Structure fingerprints are code metrics like number of loops and special statements (e.g., wait, notify, and **if**). Type fingerprints consist of code metrics like number of declared methods, number of fields, and number of parameters.

Structural relationships between program elements have alternatively been modelled using program dependence graph (PDG) (*Horwitz et al.*, 1988). Subgraph similarity has been tried by code clone detection community to identify source code with similar syntactic structure and data flow relationships (*Krinke*, 2001; *Liu et al.*, 2006). *Reverb* (*McIntyre and Walker*, 2007) employes a set of similarity matching heuristics on a graph built from structural facts extracted from source code to find structurally similar source code. *Jigsaw* (*Cottrell et al.*, 2008b) utilizes a similar approach on abstract syntax trees (AST) in order to identify structural similarities in source code when integrating retrieved source code with the developer's context.

Structural retrieval is most effective when the interface or the internal structure of the function sought after is known is advance. Domain-specific software follows domain vocabulary and standards. Use of domain-specific APIs positively influence the interface similarity of software (*Kratz*, 2003). Our test similarity model in our TDR approach described in Chapter 4 uses structural similarity in addition to other heuristics. Organizational developers writing domain-specific code using a bounded set of APIs will be benefited by the structural similarity heuristics in our model.

2.6 Test Similarity

Identifying similar test cases is an area of interest to a number of research communities including test suite reduction and test case prioritization. These lines of research focus on domain-specific similarities in test cases. Similarity search takes place in the domain of the same software or its release history where the APIs are limited and known in advance. Some of the underlying similarity matching heuristics in our proposed test similarity model in Chapter 4 are analogous to techniques used in other research fields that perform similarity search on tests.

2.6.1 Test Suite Reduction

Test suite reduction (or minimization) techniques attempt to minimize test maintenance costs by eliminating redundant test cases from test suites. Obviously, reducing the size of a test suite will only be justifiable if it does not reduce the its ability to reveal bugs in the software. Most test suite reduction approaches rely on coverage information collected during dynamic execution (*Harrold et al.*, 1993; *Offutt et al.*, 1995). Dynamic analysis of test cases requires availability of test input data or specialized resources (e.g., execution environment or hardware). Static approaches have been proposed for identifying potentially redundant tests cases based on similarity of instruction sequences when execution of software is not an option (*Li et al.*, 2008).

2.6.2 Test Case Prioritization

Test case prioritization (TCP) is a technique to reduce the cost of regression testing. Test cases are prioritized according to some measure of importance so that for example test cases that may detect bugs in the software are run earlier in the regression testing process. Most TCP techniques use the dynamic run behaviour (e.g., statement coverage) of the system under test (*Elbaum et al.*, 2002). Static TCP techniques try to maximize diversity by giving priority to test cases that are highly dissimilar. Similar test cases may be identified based on the static call graph of the test cases (*Zhang et al.*, 2009), string distance (*Ledru et al.*, 2012), or their linguistic features (*Thomas et al.*, 2012). While static TCP techniques do not have as much information to work with as those based on execution information, static techniques are less expensive and are lighter weight, making them applicable in many practical situations.

Chapter 3

An Assessment of TDR

The evaluation of TDR tools to-date has been performed using a collection of classic tasks commonly used in the software reuse literature. Most of these tasks involve common data structures and functions, for which the developer can be expected to use the standard domain-specific vocabulary. We claim that these tasks are not representative of the daily requirements of developers performing TDD, where a developer cannot be expected to know of a standard domain-specific vocabulary.[1]

To express our concerns about the existing evaluations of test-driven reuse approaches (*Hummel and Janjic*, 2013; *Lazzarini Lemos et al.*, 2011; *Reiss*, 2009a), we have categorized the tasks that they used according to the commonality of the domain and thus its vocabulary, and the commonality of the variation sought. The categories derive from our conjecture that the more well-defined a domain vocabulary is, the easier it should be to locate software from that domain. Likewise, common variations of a domain (e.g., an ordinary stack data structure) are easier to locate than uncommon variations of the same domain (e.g., a stack data structure retaining unique objects). It is important to note that task difficulty (in terms of implementation)

[1]This chapter is an extended version of the paper *An Assessment of Test-Driven Reuse: Promises and Pitfalls* co-authored with *Robert J. Walker* and *Frank Maurer* that appeared in *John Proceedings of 13th International Conference on Software Reuse, volume 7925 of ICSR 2013, pages 6580, Springer Berlin Heidelberg, 2013.* Permission to use the paper in this doctoral dissertation was granted to the first author by the co-authors.

is orthogonal to the difficulty in locating an existing implementation: a complicated but well-known algorithm might be very easy to find, while a simple utility that is not usually referred to by a common name might be very difficult to locate.

60 to 80 percent of the tasks from the existing evaluations fall into the most trivial cases category. For example, an implementation of the quick sort algorithm, stack data structure, or a function that prints out prime numbers are the kind of tasks given to people learning a programming language. Ironically, while the existing evaluations concentrate heavily on wellknown cases, we believe that the lessknown tasks are the ones in which a developer is most likely to need and to want to use a TDR tool. To study the effectiveness of the existing test-driven reuse tools, we conducted an experiment with tasks drawn from real developers.

3.1 Experiment

Our research question is “Do existing TDR tools recommend relevant source code for which minimal effort should be needed to integrate with an existing project?” In particular, we consider whether each tool is able to recommend known source code that solves a task, given modified forms of the existing test cases—taken from the same codebase—that exercise that source code.

To this end, we located a set of tasks that were discussed by developers on the web, suggesting that these were problematic for them. For each task, we also located an implementation that would serve as a solution; each implementation needed to exist in the TDR tools’ repositories and have automated test cases associated with them. We detail our tasks and our selection methodology in Section 3.1.1.

The test cases associated with the known solutions would not be appropriate to give to the TDR tools without modification, because (1) they would immediately identify the project and classes that would serve as the solution, and (2) they would be far too detailed to represent realistic test cases drawn from test-driven development. In TDD, the developer can at best be expected to have an approximate idea of the interfaces to be exercised. Thus, the test cases needed to be altered in as unbiased manner as possible. Furthermore, because of implementation choices and shortcomings of the prototype tools, additional translations were required in order to be fair to the tools. We detail our experimental design including the test case modification methodology in Section 3.1.2.

After obtaining recommendations from each tool for each task, we needed to assess their suitability as solutions for the task. We utilized two subjective, qualitative measures for this purpose: relevance and effort. To improve the construct validity of our measurements, we had other developers assess a sample of our suitability assessments, and compared them with our own. We detail our and the external assessments of suitability in Section 3.1.3.

3.1.1 Task Selection

Constructing a definitive reference task set that is comparable to those used in evaluation of text retrieval remains an elusive target (*Hummel*, 2010). Instead, we located the 10 programming tasks in Table 3.1 for our experiment. We explain below our methodology for obtaining these.

Sources of tasks. As the existing TDR tools all operate on *Java* source code, we focused on *Java*-oriented sources of information. We first examined material designed for developer education, including *Oracle*'s *Java* Tutorial (http://docs.oracle.

Table 3.1: Brief task descriptions.

Task	Description
1	A Base64 coder/decoder that can Base64 encode/decode simple text (String type) and binary data (**byte** array); should ignore invalid characters in the input; should return NULL when it is given invalid binary data.
2	A date utility method that computes the number of days between two dates.
3	A HTML to text converter that receives the HTML code as a String object and returns the extracted text in another String object.
4	A credit card number validator that can handle major credit card types (e.g., Visa and Amex); determines if the given credit card number and type is a valid combination.
5	A bag data structure for storing items of type String; it should provide the five major operations: add, remove, size, count, and toString.
6	An XML comparison class that compares two XML strings and verifies if they are similar (contain the same elements and attributes) or identical (contain the same elements and attributes in the same order).
7	An IP address filter that verifies if an IP address is allowed by a set of inclusion and exclusion filters; subnet masks (like 127.0.1.0/24, 127.0.1/24, 172.16.25.∗ or 127.∗.∗.∗) can be used to define ranges; it determines if an IP is allowed by the filters or not.
8	A SQL injection filter that identifies and removes possible malicious injections to simple SELECT statements; it returns the sanitized version of the supplied SQL statement; removes database comments (e.g., −−, #, and ∗) and patterns like INSERT, DROP, and ALTER.
9	A text analyzer that generates a map of unique words in a piece of text along with their frequency of appearance; allows for case-sensitive processing; it returns a Map object that maps type String to Integer where the key is the unique word and the value is the frequency of the word appearance.
10	A command line parser with short (e.g., −v) and long (e.g. −−verbose) options support; it allows for options with values (e.g. −d 2, −−debug 2, −−debug=2); data types (e.g. Boolean, Integer, String, etc.) can be explicitly defined for options; it allows for locale-specific commands.

com/javase/tutorial). Such tutorials are usually developed by experienced developers to teach other developers what they should know about a language or technology because they are likely to come across tasks that would require that knowledge. In a similar fashion, we looked in source code example catalogues including *Java2s* (http://www.java2s.com), which features thousands of code snippets demonstrating common programming tasks and usage of popular application programming interfaces (APIs). These two sources represent material designed for consumption by developers. To find what kinds of problems developers seek help to solve, we also looked at popular developer forums: *Oracle* Discussion Forums (https://forums.oracle.com/forums/category.jspa?categoryID=285) and *JavaRanch* (http://www.javaranch.com), where developers discuss their information needs with fellow developers.

Locating known solutions and their test cases. After locating descriptions of pertinent tasks, we sought existing implementations relevant to those tasks on the internet through code search engines, discarding search results that did not also come with *JUnit* test cases. After locating pertinent candidates we checked that both the solution and the test cases that exercised the solution existed in the repositories of the tools. Dissimilarity of the tools' underlying repositories made it difficult to select targets that simultaneously existed in all three repositories. Therefore, we settled for targets that exist in at least two of the three investigated repositories. Task selection and experimentation was performed incrementally over a period of three months; we found this process to be slow and laborious and thus we limited the investigated tasks to ten.

Coverage of multiple units. *JUnit* tests can cover more than one unit (despite the apparent connection between its name and "unit testing"). For example, an integration test can cover multiple classes collaborating in a single test case. Ideally, a test-driven reuse tool should be able to recommend suitable candidates for each missing piece referred to in a test scenario. Instead, current TDR prototypes have been designed around the idea that tests drive a single unit of functionality (i.e., a single method or class) at a time. We aimed our trial test cases to those targeting a single unit of functionality.[2]

3.1.2 Experimental Method

The experiment involved, for each identified task, (a) modifying the associated test case to anonymize the location of the known solution, (b) feeding the modified test case to the interface of each TDR tool, and (c) examining the resulting recommendations from each tool in order to assess their suitability to address the task (discussed in Section 3.1.3).

For simplicity of the study design, we assume that iterative assessment of recommendations and revision of the input test cases can be ignored. We further assume that a searcher would scan the ranked list of recommendations in order from the first to the last; however, we only consider the first 10 results, as there is evidence that developers do not look beyond this point in search results (*Joachims et al.*, 2005).

Anonymization. We wished to minimize experimenter bias by using (modified versions of) the existing test cases of the solutions. The query for each task then

[2]In one case, this constraint was not strictly met: the known solution for Task 4 relies on a set of **static** properties defined in a helper class CreditCardType.

consisted of the modified *JUnit* test cases—thereby defining the desired interfaces, vocabulary, and testing scenarios of the trial tasks.

The test code used in the experiment was anonymized by a four step process: (1) any **package** statement is removed; (2) any **import** statements for types in the same project are removed (by commenting them out); (3) the set of test methods is minimized, by ensuring that all required methods for the sought functionality are exercised no less than once, while removing the rest; and (4) the statements within each test method are minimized, for cases where multiple conditions are tried, by retaining the first condition and removing the rest. This process was intended to reduce the test cases to the point that would resemble the minimal test scenarios developed in a TDD setting.[3]

Tool-specific adjustments. Minor adjustments were made to some test cases in the end to make them compatible with individual tools. For instance, *Code Conjurer* does not fire a search when no object instantiation takes place in the test code, preventing *Code Conjurer* from triggering a search when the target feature is implemented through **static** methods. To get around this problem, we revised test cases for Tasks 1, 2, and 4 and replaced **static** method calls with instance method calls preceded by instantiation of the unit under test. The example in Figure 3.1 demonstrates some of the changes made to the query test class Base64UtilTest used in Task 1 for replacing **static** method calls with instance method calls.

Unlike *Code Conjurer* and Code Genie, *S6* comes with a web-based search interface; a class-level search through the *Google* or *Sourcerer* search provider was selected

[3]The complete set of known solutions and test cases, and their transformed versions used as inputs, can be retrieved from: http://tinyurl.com/icsr13.

```
@Test public void testEncodeString() {
  final String text = "This is a test";
// String base64 = Base64Util.encodeString(text);
// String restore = Base64Util.decodeString(base64);
  Base64Util util = new Base64Util();
  String base64 = util.encodeString(text);
  String restore = util.decodeString(base64);
  assertEquals(text, restore);
}
```

Figure 3.1: A sample test query illustrating the replacement of **static** calls.

for all the searches performed through the *S6* web interface. As *S6* cannot process test code directly, a conversion process was followed to transform individual test cases into a form acceptable by the *S6* interface (as a "call set"). Minor changes were made in some transformations due to the limitations imposed by the interface. In the case of Task 1, we could not provide the three test cases to *S6* all at the same time; tests were therefore provided to the *S6* search interface one at a time but *S6* did not return results for any of them. For Task 9, we were not able to exactly reproduce the test case using an *S6* call set. More specifically, we were not able to manipulate the returned java.util.Map object from the getWordFrequency() call; neither removing the assertions on the returned java.util.Map object nor using the "user code" feature produced any results. Task 10 involves the inner class CmdLineParser.Option that made *S6* complain about an unknown type; we replaced the inner class with the type Object in order for the search to be launched.

3.1.3 Suitability Assessment

Many factors can make unplanned reuse difficult (*Holmes and Walker*, 2012). Two pieces of code of similar quality might satisfy the same feature set; however, developers are inclined to reuse the one that requires less adaptation and accommodation—after all, major motivations for reuse are to save time and effort and to reduce the likelihood of bugs. Therefore, it is important to present the developer with choices that they are likely to consider as relevant to their task, suitable for integration in their code, and whose adaptation would result in a net cost savings. To assess the quality of retrieved results, we thus recorded two subjective, qualitative measurements: relevance and effort.

Relevance and effort. Relevance is a measure traditionally used in evaluation of information retrieval (IR) systems, to indicate how well a retrieved resource meets the information needs of the user. In the context of this study, relevance measures how many of the features expected by a trial task are covered by a search result. For TDR, good relevance is necessary but not sufficient.

Table 3.2: Guidelines for classifying relevance.

Level	Description
1	There is no meaningful connection between the given task and the recommended code. I would not reuse this code to finish this task.
2–4	There is a noticeable overlap between the given task and the recommended code. However, some of the required features of the described functionality are missing or implemented in a different way (the smaller the mismatch, the higher the relevance).
5	The recommended code exactly or closely matches the functionality described in the task.

Effort is a measure of the work involved to adapt the retrieved source code. Effort is a compound measurement of size and complexity of the source code to be integrated, external objects it refers to, and the amount of mismatch it has with the existing development context; for our study, the "existing development context" comprised the test suite used as input to the search.

Two recommendations with the same relevance level may have completely different effort levels. For example, one recommendation can be considerably larger, more complex, or have more external dependencies than an equally relevant one. However, relevance and effort are not orthogonal. As the relevance starts to decline, the effort tends to increase; for example, additional effort might be required to add missing features. Ultimately the developer will avoid reusing located source code if the adaptation effort is (apparently) comparable to that of reimplementation.

To account for partial suitability, we adopted 5-point scales of relevance and of effort, as shown in Tables 3.2 and 3.3 respectively. For rating relevance, 1 stands for no/minimal relevance and 5 stands for complete relevance. For rating efforts, 1 stands for little/no effort while 5 stands for excessive effort.

Table 3.3: Guidelines for classifying effort.

Level	Description
1	This code can be reused to develop the entire functionality described in the task. I may only need to do one or two very simple adjustments.
2–4	I may or may not choose to reuse the recommended code or its design ideas. However, to reuse it I would have to refactor it, make modifications to its design, or write new code for missing features (the fewer the adjustments, the lower the effort).
5	I would not reuse this code to develop the functionality described in the task. It would require too much effort to build upon this code.

Assessments of suitability. After running the 10 trial tasks against the three test-driven reuse tools we collected 109 individual recommendation results (out of 300 possible results), as the number of results was less than our cut-off of 10 for some task/tool combinations.

Figures 3.2 and 3.3 respectively show the relevance and effort scores assigned to results from individual tasks. Each subfigure represents the results for the indicated task, for up to the first 10 results, represented by three vertical bars for *Code Conjurer*, *CodeGenie*, and *S6* respectively. The absence of a bar indicates the lack of a corresponding result.

There are 25 potential combinations of relevance and effort scores; however, only some of those combinations are expected to be observed, due to the relationship between relevance and effort discussed above. Table 3.4 indicates our classification of each possible combination, as good (a solution), ok (a near-solution), bad (a non-viable recommendation), or impossible (left as blank). We deem combinations of fairly low relevance and low effort to be contradictory and hence impossible, since lack of relevance implies high effort would be needed. In the absence of a good solution, a developer might consider a near-solution, which is a relevant result that still requires non-trivial effort to use for the task.

Relevance and effort ratings were assigned to the results following a manual inspection of the retrieved source code. An attempt was made to integrate the query test cases with the retrieved code, if possible. To make the tests run, external dependencies of the source code were resolved and refactorings were performed, if necessary. Ratings were given based on the effort spent to make the tests run and an estimate of the extent of the missing features in each case.

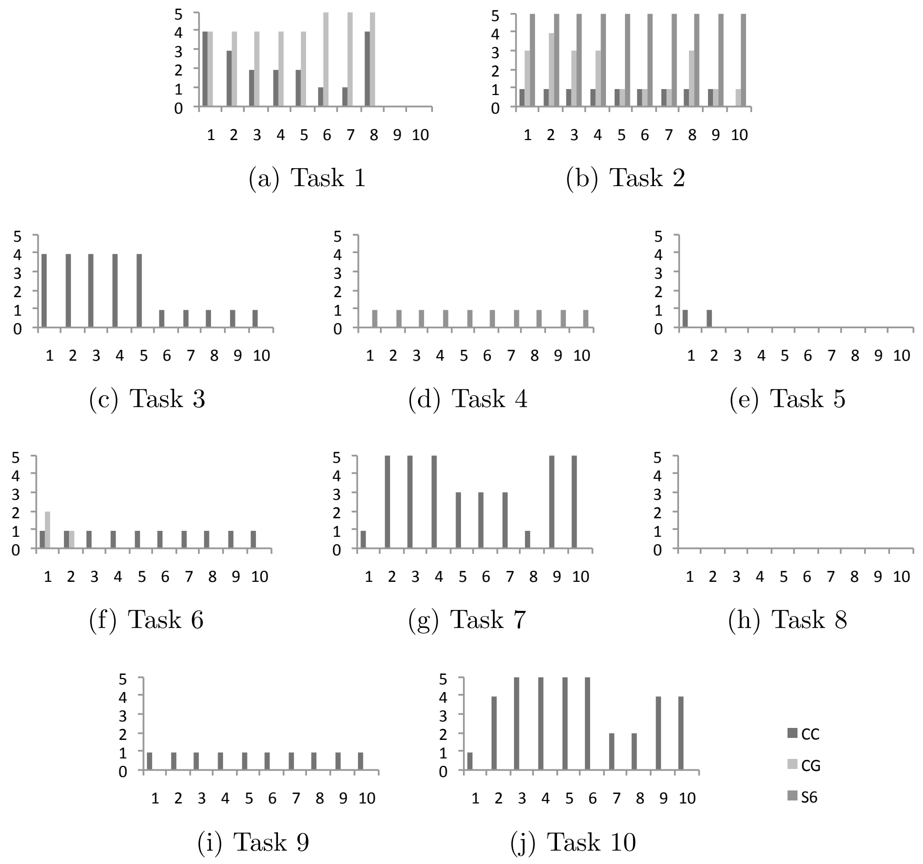


Figure 3.2: Task relevance level versus result number, for the 10 tasks.

External validation. As relevance and effort are subjective measures, different developers can disagree on the reuse suitability of a piece of code in a certain context. To improve construct validity of the experiment, we compared our relevance and effort ratings with those from five experienced *Java* developers. Participants consisted of two graduate students and three industrial developers, all with 3–5 years of industrial experience in developing *Java* software. All participants reported that being familiar with the *JUnit* framework, having developed unit tests, and having conducted code reviews in the past. A short training example was used at the start of the session

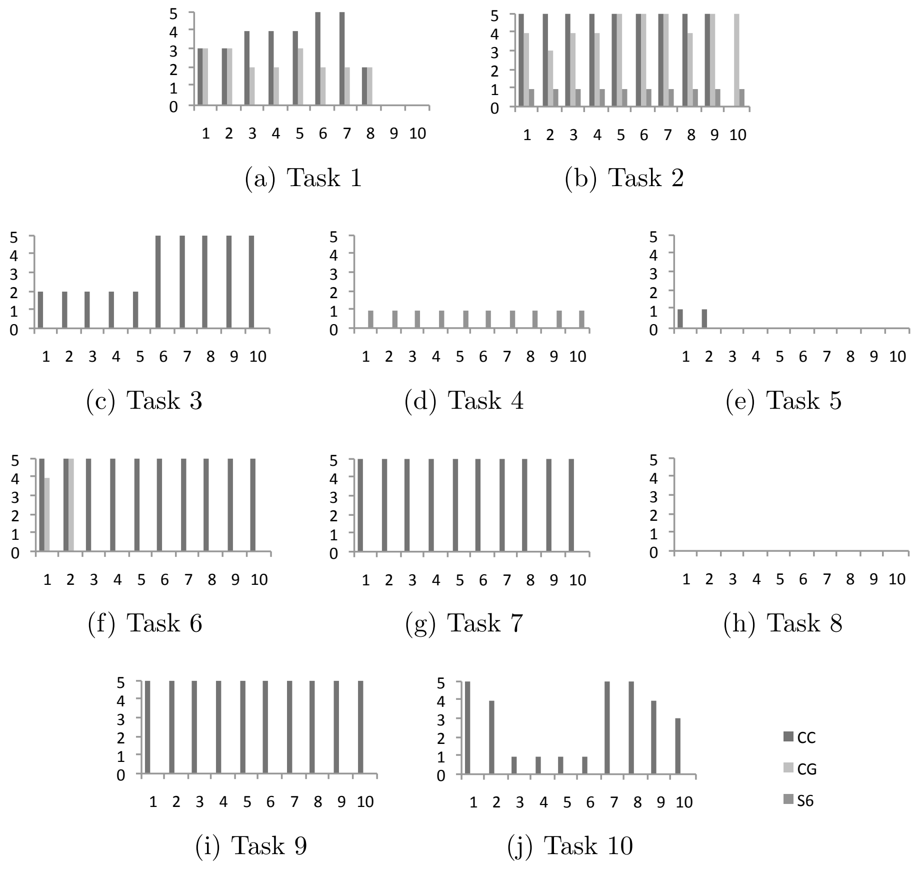


Figure 3.3: Task effort level versus result number, for the 10 tasks.

to familiarize participants with the procedures. A random sample of results (12 out of 109 recommendations) were selected, and provided to each participant for evaluation; each participant was asked to evaluate the same sample. In a short post-study questionnaire, all participants indicated that they have developed code for tasks similar to the ones they were given in the experiment, confirming that these are realistic tasks.

Participants were given a guide that described the purpose of this experiment and the rationale behind the relevance and effort scores, along with—for the results in the

Table 3.4: Quality classes.

		Relevance				
		1	2	3	4	5
Effort	1					good
	2				ok	good
	3			bad	ok	ok
	4		bad	bad	bad	bad
	5	bad	bad	bad	bad	bad

Table 3.5: Inter-rater reliability scores.

	P1	P2	P3	P4	P5
Relevance	0.86	0.89	0.93	0.84	0.81
Effort	0.72	0.75	0.82	0.74	0.72

sample to be validated—a short description of each task, the test cases, and the source code retrieved by a tool. Participants were asked to rate each result's relevance and effort according to our 5-point scales. Participants were asked to justify their choice through additional comments, which we used to check that their reasoning conformed to their numerical ratings. Spearman's rank correlation coefficient (ρ) was used to measure the inter-rater reliability of the rankings made by the first author and each participant; Spearman's ρ can measure pairwise correlation among raters who use a scale that is ordered. Table 3.5 displays the ρ values computed for participants P1 to P5. In all five cases, there is strong positive correlation between relevance and effort ratings of the first author and those of the external validators.

3.2 Results and Discussion

Table 3.6 summarizes the results of the experiment for each tool/task pairing, indicating: the number of recommendations returned; how many of these were duplicates; the ranking by the respective tool of the recommendation that we deemed the best, within the first 10 results (or fewer if fewer were recommended); and the quality of the best solution. To be clear, in some cases, the recommendation by a tool that we deemed best amongst its results, we still assessed as badly suited; the tools' rankings and our assessment of quality often did not correlate.

We can see that each of the tools did a poor job at recommending solutions.

Table 3.6: Results of the experiment for each tool/task combination. The columns for each tool indicate: the number of recommendations produced (rec); the number of these that are duplicates of other recommendations (dup); the ranking by the tool of the recommendation that we deemed the best within the results (best); and the quality of that best recommendation (qual). In cases marked with an asterisk, we were not able to verify the presence or absence of the known solution within the tool's repository.

Task	*Code Conjurer*				*CodeGenie*				*S6*			
	rec (#)	dup (#)	best (rank)	qual	rec (#)	dup (#)	best (rank)	qual	rec (#)	dup (#)	best (rank)	qual
1	8	(1)	8	ok	*8	(3)	3	good	0			
2	9	(1)	1	bad	*10		2	ok	10	(6)	1	good
3	10	(1)	3	ok	*0				0			
4	0				*0				10	(6)	1	bad
5	2	(1)	1	bad	*0				0			
6	10	(2)	1	bad	2		1	bad	0			
7	10	(5)	2	bad	0				0			
8	0				*0				0			
9	*10	(4)	1	bad	0				0			
10	10	(3)	3	good	0				0			

Table 3.7: Summary of quality classifications for all recommendations. The number of duplicate recommendations included is shown in parentheses.

Quality	*Code Conjurer*	*CodeGenie*	*S6*
good	4 (3)	3 (2)	10 (6)
ok	3	11 (2)	—
bad	62 (15)	6	10 (6)
Total	69 (18)	20 (3)	20 (12)

Code Conjurer provided a good solution for only one task, and near-solutions for two others; for five of the remaining tasks only bad recommendations were provided. *CodeGenie* provided a good solution for only one task, and a near-solution for one other; no recommendations were provided for seven out of eight of the remaining tasks, so false positives were relatively low. *S6* only provided a good solution for one task, and no near-solutions; again, no recommendations were provided for seven out of eight of the remaining tasks, so false positives were relatively low. For Task 8, none of the tools provided a recommendation. For Tasks 4–7 & 9, each tool either provided no recommendations or only bad recommendations. In fairness, for task/tool combinations in which we were unable to verify the presence of the known solution (marked with asterisks), the associated repository may not have contained a viable alternative but this only affects four of the tasks for *CodeGenie* and none for the other two tools.

Table 3.7 summarizes our classifications of all recommendations produced by the tools for the 10 tasks. *Code Conjurer* has a much larger number of false positive (i.e., bad) recommendations than the other two, but all three tools produce many bad recommendations, when they produce any recommendations at all.

The results of our experiment indicate that there is a serious problem at work with the existing TDR approaches. Could the problem simply be due to implementation weaknesses, or is there a more fundamental shortcoming with the underlying ideas? To address this question, we first examined similarities between the input test cases and the results, described below.

3.2.1 Lexical and syntactic matching

From the published literature on the TDR approaches, we recognized the importance that each places on lexical and (to a lesser extent) syntactic similarity between potential hits in the repository and the input test case. Specifically, *Code Conjurer* and *CodeGenie* both utilize similarity of type and method names plus similarity of method signatures; *S6* utilizes similarity between user-supplied keywords and potential hits plus similarity of method signatures. We manually examined each recommendation returned by the tools to determine if lexical or syntactic similarities existed with the input test case for each task. We empirically discerned four kinds of matching criteria: type name, method name, signature, and other keywords.

Table 3.8 presents the results of our similarity examination. We can see that *Code Conjurer* places great emphasis on type name similarity while *S6* ignores it. But ultimately, every recommendation could be traced to a mostly lexical similarity.

This heavy reliance on lexical/syntactic similarity to the supplied test case, in making recommendations, yields many false positive results—especially when simple functionality is sought. For example, the utility program sought in Task 2 consists of a single function with two parameters of type `java.util.Date` and a return value of type **`int`**. *Code Conjurer* retrieved 9 results all of which match this signature but none of

Table 3.8: Classification of matches between recommendations and input test cases. For a given recommendation, multiple kinds of match may exist.

Match kind	*Code Conjurer*		*CodeGenie*		*S6*	
	High similarity	Partial similarity	High similarity	Partial similarity	High similarity	Partial similarity
Type name	62		15			
Method name	5	8	1	8		6
Signature	15	8	10	8	20	
Other keyword	21		2		20	
Total recoms.	69		20		20	

which match the desired functionality.

Each approach had the greatest success when multiple kinds of similarity occurred simultaneously; again, this is not surprising since the likelihood that similarities in multiple dimensions are spurious seems much lower than in few dimensions. Unfortunately, it appears from the results that simply demanding multiple kinds of lexical/syntactic similarity simultaneously would lead to limited applicability of these tools. Others have noted the tradeoff limitations to lexical/syntactic similarity in code search (*Holmes et al.*, 2006).

3.2.2 Issues with the Approaches

We see several issues that arise not from weaknesses in tool implementation, but more fundamentally from the ideas behind the TDR approaches.

Signature matching. The existing test-driven reuse approaches make signature matching a necessary condition to the relevance and matching criteria: a component is considered only if it offers operations with sufficiently similar signatures to the test conditions specified in the original test case. However, semantic similarity neither implies nor is implied by structural similarity. This limits the applicability of the test-driven reuse to situations in which the design of the feature sought is very simple or known in advance.

A more flexible approach to signature matching could improve recall. For example, operation argument and return value types, order, or count could be ignored. Unfortunately, this would in turn make the execution of test cases for validating candidate results difficult. Automated tests can only be run if a match can be established between missing elements in the tests and those in the retrieved source code. *Code Conjurer* retrieved testable results (source code that has sufficiently close signatures for at least some of the operations) for Tasks 1, 3, and 10, while *CodeGenie* could achieve the same goal only for Tasks 1 and 2. *S6* tries to take advantage of simple transformations to generate possible candidates that can pass the tests; however, a candidate is considered for applying the transformations only if structural dissimilarities are minor. Consequently, *S6* ended up retrieving results for only two of the tasks (Tasks 2 and 4) because candidates retrieved for other tasks did not meet the input criteria of the transformations.

Filtering by lexical relevance. Filtering candidate results based on their lexical relevance before other relevance criteria are considered leaves out all potential solutions that do not match the searcher's choice of program vocabulary. For example, the date utility function in Task 2 is named getNumberOfDaysBetweenTwoDates() and is defined in a class named DateUtils. Tokenizing these two names, as is performed by *Code Conjurer* and *CodeGenie*, would give a list of generic words that can be matched with almost any date utility class. All the 9 classes retrieved by *Code Conjurer* are named DateUtils and each has at least one method matching the signature of the method sought after; however, none of them is a method that computes the distance between two dates. *CodeGenie* manages to find an instance of the function for Task 2 that is given the same name and is defined in a class with the same name. Other results are false positives arising from lexical and signature similarities.

Code Conjurer and *CodeGenie* use terms in the signature of methods as a key component of relevance. While *S6* uses a supplied keyword list to shrink the candidate space in which signature matching and transformations are to be performed, the use of keywords in the search has been limited to lexical matching that in turn results in tool performance being limited by the searcher's choice of vocabulary (*Reiss*, 2009a; *Lazzarini Lemos et al.*, 2011). We designed our experiment to favour the evaluated tools: we used test code taken from the same project in order to retrieve the feature under test. Changing the original program vocabulary instead—which would be reasonable in modelling situations where the developer does not know the needed vocabulary—would have limited lexical relevance, resulting in even worse performance of the tools.

Automatically compiling and running source code. The existing test-driven approaches all attempt to execute the supplied test case on potential matches in the repository. This has the advantage that additional semantic constraints implied by the supplied test case can be checked, eliminating false positive matches. Given the large number of false positives that we obtained from the approaches, it is clear that this idea is not working as intended; in fact, for *Code Conjurer* in particular, we believe that test case execution remains an unimplemented idea, judging from the very large number of false positives.

The retrieved source code has to be runnable in order to execute test cases, but automatically compiling and running arbitrary source code accumulated in a repository is no easy task, often because of dependencies on external source code. A number of heuristics have been proposed by the research community to resolve external dependencies without developer intervention (*Ossher et al.*, 2010). In the context of our trial, only Task 4 has an external dependency on the org.apache.commons.lang project, while Tasks 6, 7, and 8 have dependencies to other source code in the same project, and the rest of the tasks can be compiled using a standard JDK by itself.

Even if source code can be compiled, there is still no guarantee that it can also be run, as programs can rely on specific runtime environments or resources. For example a web or mobile component relies on a specific container to run. A database or network application requires those external resources to offer its services. To the benefit of the evaluated tools, none of our tasks required an external environment or resources to run.

In a similar fashion, the TDR query test cases may also rely on resources external to the *JUnit* program. For example, the original version of the XML utility for Task 6 relied on XML strings loaded from the file system. For the sake of our experiment, we modified the test[4] (Figure 3.4) to utilize XML strings embedded in the source code, but there is no a priori reason to expect that the developer will not wish to rely on external resources in this fashion.

Contextual facts. Test cases are in fact simple examples demonstrating the use of the system-under-test. They show helper types that may interact with the system-under-test in typical scenarios. The existing TDR approaches disregard the elements

[4]We repeated this experiment with the original version of the test case. *Code Conjurer* and *CodeGenie* produced the same result. *S6* does not allow using external resources.

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.io.IOException;
import org.xml.sax.SAXException;

public class TestXmlDiff {
    String xml1 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>text1
        </b><c>text2</c></a>";
    String xml2 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy
        --><a><c>text2</c><b>text1</b></a>";

    /* @Test public void testDiff() throws XmlException, IOException, SAXException {
        DomainsDocument dd1 = DomainsDocument.Factory.parse(new File("src/test/
            resources/instances/test1.xml"));
        DomainsDocument dd2 = DomainsDocument.Factory.parse(new File("src/test/
            resources/instances/test2.xml"));
        Diff myDiff = new Diff(dd1.toString(), dd2.toString());
    } */

    @Test public void testDiff() throws IOException, SAXException {
        Diff myDiff = new Diff(xml1, xml2);
        assertTrue(myDiff.similar());
        assertFalse(myDiff.identical());
    }
}
```

Figure 3.4: Test case for validating equality and similarity of XML strings (Task 6).

of this interaction like participating types and the data/control flow between them. They merely extract lexical and syntactic features of missing elements from the tests while the context in which those elements appear might also help to understand their semantics.

For example, by solely relying on names and signatures, *Code Conjurer* did not retrieve anything related to XML processing for Task 6 (the input test case is shown in Figure 3.4). Only one of the results out of the 10 retrieved happened to contain the keyword "xml" that was a statement importing the class org.allcolor.xml.parser.CStringTokenizer. The exception type org.xml.sax.SAXException thrown by the test method testDiff is part of a well-known API for processing XML documents. SAXException is not thrown by any of the resolvable methods in the test scenario; therefore, the functionality being sought should throw that exception. Incorporating this additional semantic fact could have helped to improve the relevance of retrieved results.

Searching for a specific implementation. *Code Conjurer* and *CodeGenie* managed to retrieve relevant results for Task 1 in which a Base64 encoder/decoder is sought. The class name, base64 is the name of a well-known algorithm. The method names encode and decode are common choices for a utility class offering such services. However, the Base64 encoder/decoder described in the tests extends the common variation of this algorithm and adds a few constraints. When decoding Base64 character sequences, it should detect and ignore invalid sequences and simply return NULL. Therefore, our Task 1 is slightly different from most of the Base64 encoder/decoders available on the internet. None of the recommendations by *Code Conjurer* and *CodeGenie* offers the special behaviour expected. *S6* fails to retrieve any results for the very same task. We speculate that it has retrieved various implementations of the Base64 algorithm through its initial keyword search, but not exactly the variation described in the tests; as none of them could pass the tests, they were all discarded in the end.

Most of the tasks reported by proponents of test-driven reuse approaches seek common variations of well-known algorithms and data structures. Using lexical and signature relevance criteria would yield multiple instances of such programs that can possibly pass the tests. However, similar to the example given above, if a variant were sought, relying on common terms and operation signatures would not suffice.

3.2.3 Issues with the Tools

In addition, the prototype tools themselves have a number of issues that need to be addressed.

Ranking. Table 3.6 shows the rank at which the tool placed the result for each task that we deemed to be the most suitable out of the recommendations. Only in 1 out of 6 cases in which a solution or near-solution was recommended was it ranked first by the respective tool (by *S6* for Task 2). For 3 of the 5 remaining cases, the item ranked first by the tool had the same relevance score as the solution or near-solution recommendation but rated as requiring higher effort. Hence, a better estimate of the adaptation effort should be employed in the ranking functions.

We also observed that duplicate recommendations in the result lists are not ranked consecutively. As duplicates are necessarily equally suitable, the expectation is that they should be equally treated by the ranking function.

Duplicates. There are arguments and supporting evidence that redundant results do not represent any additional value to users and can lead to disorientation and frustration (*Kazai and Lalmas*, 2006). We observed that all the tools generated duplicate recommendations in their result lists. The number of duplicate recommendations produced by tools for each task is shown in Tables 3.6 and 3.7. For example, 12 out of 20 results that *S6* retrieved for Tasks 2 and 4 were duplicates.

Most redundant recommendations originated from secondary sources, such as a

Java archive file that is replicated in multiple locations online. A code search engine crawler then retrieves and indexes the same source files multiple times. We also came across cases where the same source file (from the same origin) appeared more than once in the search results. Such duplicates could have been avoided via strategies like shingling (*Manning et al.*, 2008). Alternatively, duplicate results can be grouped together at presentation time simply because they are equally suitable.

Near-Misses. We define a near-miss as a piece of code that is structurally related to the source code sought, such as a bad method recommendation that is declared within the same class as a relevant method that should have been recommended instead. *Code Conjurer* and *CodeGenie* produced near-misses (*Kazai and Lalmas*, 2006) in their search results. For instance, *Code Conjurer* recommended a few methods that used other methods implementing the desired feature, in Tasks 1 and 3. Likewise, *CodeGenie* returned a test case in which a method implementing the desired feature was being tested, in Task 6. *CodeGenie* also recommended an interface rather than a relevant, concrete implementation of that interface, in Task 2.

Integration Support. *Code Conjurer* and *CodeGenie* provide limited automated support for integrating recommendations with an existing system. However, both tools have issues when trying to resolve dependencies, i.e., source code from the same project or external libraries that the retrieved code depends on.

When *Code Conjurer* inserts a file into the current project, it can overwrite and destroy existing code. Inserting code into a new blank project just downloads code to the developer's machine but leaves the integration task to the developer. Neither approach recovers all referenced source code or external libraries. What is missing in the end has to be manually resolved, adding to the effort required for the complete reuse task.

CodeGenie provides better integration support in comparison to *Code Conjurer*. It relies on program slicing to return self-contained pieces of code. However, the end result is often not compilable, especially when functionality in an external library is referenced. *CodeGenie* tries to slice up the source code for the library and return the exact subset that is required. The client–server connection occasionally times out when the size and number of externally referenced resources increase. Finally, the weaving process that is intended to take care of resolving the dependencies between existing and retrieved code was mostly unreliable; in most cases, we were unable to use the woven code for the tasks.

S6 does not provide IDE tooling; therefore, it does not provide automated integration support with retrieved code.

Reliability. We are not certain that *CodeGenie* fired a search for Tasks 9 and 10; the *Eclipse*-based client appears to be faulty or expecting input in an undocumented form that we were unable to figure out. We managed to get results from *S6* only for Tasks 2 and 4. *S6* timed out on Tasks 5 and 7 with the error message "No response from *S6* server". The features needed by these two tasks require having more methods, relative to some of the either tasks. To help *S6* better analyze the input, we repeated each task by reducing the number of methods; however, this still did not yield recommendations.

Composing queries. We found that translating test code into the *S6* search form was difficult. We consulted online documentation and research papers, and performed some trial and error to compose queries. It would have been much easier for us if *S6*, like *Code Conjurer* and *CodeGenie*, came with an IDE plug-in. Such a front-end plug-in could then extract all required bits of information from supplied test cases to compose the query and submit to its back-end server. However, in the case of *Code Conjurer* and *CodeGenie* it was not clear what bits of information are collected from the test code. Again, we had to rely on trial and error especially when the tools produced no or too few search results.

Presentation of results. All the tools offer views with limited syntax highlighting that make evaluating code excessively difficult. Results are limited to a single file (or excerpts of it) that matched the query; browsing support for the rest of the project is not provided. This makes evaluating suitability problematic, especially in the case of near-misses.

None of the tools highlight the logic that makes retrieved code relevant (from the tool's perspective) to the test code, known as *recommendation rationale* (*Robillard et al.*, 2010). Without relevance highlighting, discarding false positives takes longer as the size and complexity of search results increases. Likewise, we found it easy to miss the most relevant part while examining larger source files.

Studies show that, when source code is located for reuse, information about the originating project is sought or inferred to contextualize that source code (*Burkhardt and Détienne*, 1995; *Rouet et al.*, 1995). Providing such contextual information should thus be a design goal for any TDR technique.

3.2.4 Threats to Validity

The primary question regarding generalizability of our study is the representativeness of the tasks. We took task ideas from the discussions in the *Java* developer community websites, and from code example catalogues commonly used as a reference by *Java* developers. Developers evidently find these features worthwhile to discuss and learn from, and not so easy to develop or to find. In addition, our five external evaluators consisting of three industrial developers and two graduate students found the tasks familiar in the sense that they had previously developed similar functionality. The number of trial tasks is another limiting factor of our study. However, it is comparable to the average number of tasks used in the evaluations in the TDR literature (*Hummel et al.*, 2008; *Lazzarini Lemos et al.*, 2011; *Reiss*, 2009a).

The modifications we made to the trial test cases in our study also threatens the validity of our study. Anonymization was performed to ensure that the facts in the test cases, other than the identity of the target project, could still contribute to identification of the solution. Test case refinement was done to remove test cases that exercised features beyond the scope of the study tasks. Neither anonymization nor refinement should negatively affect the retrieval capacity of the tools. To find the best strategy to overcome the tools' limitations, we experimented with different alternatives and compared results in each case. The alternative that yielded better results was chosen over others.

Test-driven reuse is a repetitive process. Searchers might reformulate their queries based on the current results in a way that may result in finding better results. This brings up the question of whether having a static query set is the right way to evaluate a code search tool. We deliberately ignored this issue by giving the tools the best possible queries by providing them with the test cases developed for the same code. We considered different existing variations of the features, and chose the ones that came with a reasonable test suite. This should have biased the results in favour of the tools.

Code relevance and effort categorization are subjective, and thus may differ from one developer to the next. It is often easy to say that one source code recommendation is more suitable than another, but the quantification of this difference is somewhat arbitrary. While our categorization of the relevance and effort of each recommendation represents our best judgment, a random subset of our categorizations was independently evaluated by experienced *Java* developers. The strong positive correlation between raters suggests our categorization of the results is reasonable.

Considering only the top 10 results for evaluating a retrieval algorithm might be overly restrictive, despite evidence that developers generally do not investigate more than the first 10 results (*Joachims et al.*, 2005). However, in our experiment, the tools provided fewer than 10 recommendations in 22 out of the 30 cases. Therefore, we have considered all the results collected by the tools in more than 70% of the cases.

3.2.5 Precision versus accuracy

The measures we used in our evaluation are qualitative and imprecise. Nevertheless, the results suffice to demonstrate that the approaches work poorly for these examples, and point to the need to address their underlying designs. Thus the results do provide *accuracy*: our criteria for rating the results are sufficiently well defined that our participants' ratings agreed to a degree that is quantitatively demonstrable.

The greater precision that would be obtained by using quantitative measures is not warranted—measuring degrees of “poor performance” would not provide us with a deeper understanding of the cause of the failure of these approaches. Only with an acceptable level of performance is it worthwhile to invest in precise measurements.

Chapter 4

Test Similarity Model

In test-driven reuse, the fitness of the retrieved source code is verified through running searcher supplied test cases. However, for the tests to be effective as a means of verification, semantically relevant source code containing the functionality has to be retrieved. The interface-based search approach adopted by current TDR approaches requires the searcher to know—or correctly guess—the solution vocabulary and design. In this chapter, we propose an approximate model for representing assets in a TDR repository that improves the likelihood of retrieving semantically related source code.

4.1 Software Similarity

The problem of similarity searching, as defined in program retrieval and many other modern applications, is to find a set of objects that are more similar to a given query object. The similarity between any two objects is usually computed by a distance function. A precondition of computing similarity is to define what it means for two objects to be similar, which is not always so obvious. In the context of software, the notion of program similarity is not firmly defined by the use of a particular distance function. Different metrics have been proposed in the literature for measuring similarity along various—mostly syntactic and structural—features. A feature-based similarity measure built from static analysis of software only provides an approximation of the program behaviour. For a similarity analysis to be precise the approximate model should be close to the actual behaviour of the program. Although, a perfectly precise analysis is undecidable due to Rice's theorem (*Rice*, 1953), however even without perfect precision the results are still practical and useful (*Cesare and Xiang*, 2012).

Software is a high-dimensional data space. Modelling the various aspects of a program including its terms, syntax, structure, control and data flow, assumptions, limitations, dependencies, quality, performance, and semantics requires using complex data models. Searching in high-dimensional spaces is time-consuming (*Donoho*, 2000). Generally, as the dimensionality of a data space increases the problem of efficiently finding similar objects becomes more difficult. This effect is commonly refereed to as the *curse of dimensionality* (*Bellman*, 1961): when the dimensionality increases, the volume of the space increases so fast that the available data becomes sparse; organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data however all objects appear to be sparse and dissimilar in many ways which prevents common data organization strategies from being efficient.

4.2 Test Similarity

Test suites developed for quality assurance purpose in an organization are maintained for the purposes of regression testing. We propose a technique that utilizes this source of information for finding and reusing instances of similar functionality developed in the past. Developers practicing TDD start development of new functionality by developing minimal new test cases for it. Current TDR approaches use the interface of the function under test to find similar existing functions. However, in addition to the interfaces, tests also demonstrate examples of the control and data flow, pre- and post-conditions, exceptional cases, and input/output values. We believe that TDR can take advantage of these additional attributes of test cases to query source code repositories more accurately.

Finding an existing test case in the repository that is sufficiently similar to the test case just developed can be indicative of the fact that a function similar to the functionality about to be developed has been developed in the past. The existing function can potentially be reused or looked upon as an example. In order to find similar test cases, we developed a test similarity model based on lexical, structural context[1], and data flow attributes of test cases.

We developed similarity search heuristics for finding similar test cases according to names ($Heuristic_{Lexical}$), references ($Heuristic_{Type}$), invocations ($Heuristic_{Call}$), and data flow ($Heuristic_{DataFlow}$) facts in the query test case. Each heuristic selects a number of test cases from the system *Repository* due to similarity to a certain aspect of the query test case Q. A candidate result item is a test case that is selected by at least one of the heuristics. $Similar(Q)$ represents the set of similar test cases in the system *Repository* for a query test case Q.

[1]Our reference and call-set similarity heuristics were inspired by the *Strathcona* (*Holmes et al.*, 2006) example recommendation system.

$$Similar(Q) = Heuristic_{Lexical}(Q) \cup Heuristic_{Type}(Q) \cup Heuristic_{Call}(Q) \cup Heuristic_{DataFlow}(Q) \quad (4.1)$$

Figure 4.1 demonstrates our proposed federated search[2] strategy over multiple representations of test cases. In Section 4.4 we will describe the three representation schemes underlying our similarity heuristics. An aggregator computes a similarity score (eq 4.2) based on TC's similarity scores along lexical ($sim_{Lexical}$), type (sim_{Type}), invocation (sim_{Call}), and data flow ($sim_{DataFlow}$) features. In other words, if a candidate test case is returned by more than one heuristic, its similarity scores are added up.

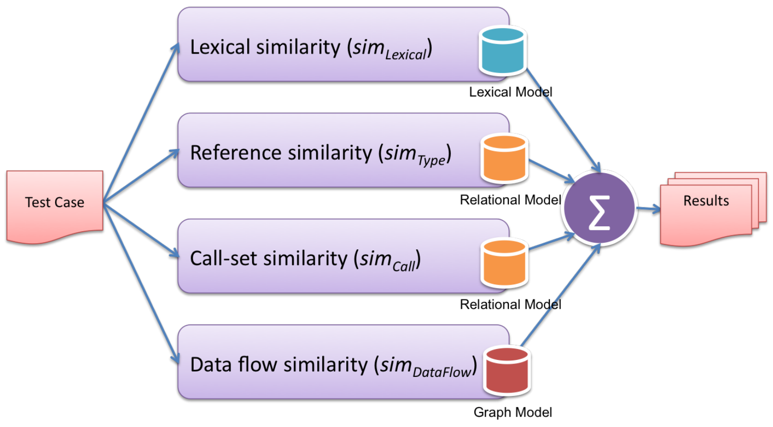

Test Case
Lexical similarity ($sim_{Lexical}$)
Lexical Model
Reference similarity (sim_{Type})
Relational Model
Call-set similarity (sim_{Call})
Relational Model
Data flow similarity ($sim_{DataFlow}$)
Graph Model
Σ
Results

Figure 4.1: A federated search platform utilizing multiple similarity search heuristics.

[2]Federated search is an information retrieval technology that allows the simultaneous search of multiple searchable resources. A user makes a single query request which is distributed to the search engines participating in the federation. The federated search then aggregates the results that are received from the search engines for presentation to the user.

Retrieved test cases TC in $Similar(Q)$ for a query test case Q are ranked in descending order of aggregated similarity score. A test case that has higher aggregated similarity can potentially have similarity along more dimensions with the query test case; hence it might be a better fit as a recommendation. Similarity scores of the four dimensions were given equal weights; i.e., $w_{Lexical} = w_{Type} = w_{Call} = w_{DataFlow} = 1$. Heuristic similarity scores are normalized by the maximum score assigned to any test case in the repository for that heuristic. If more than one test case is provided in the search query the facts from all test cases are aggregated into a unified fact set. Hence, the test cases are considered as if they were one big test case. Likewise, when trying to find a match in the repository, we find matching test classes with one or more test cases containing facts that match those in the search query.

$$\begin{aligned} AggregatedSim(Q,TC) = \\ w_{Lexical} \times \frac{sim_{Lexical}(Q,TC)}{max\{sim_{Lexical}(Q,TC') \mid TC' \in Heuristic_{Lexical}(Q)\}} + \\ w_{Type} \times \frac{sim_{Type}(Q,TC)}{max\{sim_{Type}(Q,TC') \mid TC' \in Heuristic_{Type}(TC)\}} + \\ w_{Call} \times \frac{sim_{Call}(Q,TC)}{max\{sim_{Call}(Q,TC') \mid TC' \in Heuristic_{Call}(TC)\}} + \\ w_{DataFlow} \times \frac{sim_{DataFlow}(Q,TC)}{max\{sim_{DataFlow}(Q,TC') \mid TC' \in Heuristic_{DataFlow}(TC)\}} \end{aligned} \tag{4.2}$$

In the following, we describe our heuristics and similarity functions for retrieving similar test cases based on different features of a query test case.

4.2.1 Lexical Similarity

Two test cases have lexical similarity if they use similar terms for naming programming constructs. Terms are selected from program elements including test class, test method, accessed references, exceptions, and invoked methods. In addition to the names of elements the fully qualified name of some constructs are utilized in similarity comparisons. Lexical similarity is also utilized in the interface-based retrieval approach underlying current TDR approaches. However, our lexical similarity heuristic goes beyond the terms in the interface of function under test and also considers the terms in its context. The name of the test class, test method, and references can provide additional semantic information in identifying the function under test. Table 4.1 displays the complete list of lexical features extracted from test cases. Appendix A provides an example of how terms from test cases are represented as bag of words data structures in *Apache Solr*.

Table 4.1: Lexical features used in similarity comparison of test cases.

Test Element	Property	Bag of Words
Test class	Name	Names
Test method	Name	Names
	Name	Names
Method	FQN	FQNs
	Return type FQN	FQNs
	Name	Names
Reference	FQN	FQNs
	Owner's FQN	FQNs
Exception	FQN	FQNs

Two bags of words are compiled for each test case; one for the names and a second one for the fully qualified names (FQNs) in the test case. Using *Apache Solr* each test case is represented as a vector with two dimensions; the *Names* and *FQNs* bags of words being its dimensions. To measure the similarity of two test cases one now has to measure the similarity of their term vectors. The function $sim_{Lexical}(TC, TC')$ that combines tf-idf and cosine similarity (*Manning et al.*, 2008) (as implemented by *Apache Lucene*[3]) represent the lexical similarity between query test case TC and a test case TC'. Term frequency-inverse document frequency (tf-idf), is a numerical statistic which reflects how important a word is to a document in a collection. Cosine similarity, on the other hand, is a measure of similarity between two vectors in a vector space that measures the cosine of the angle between them. The similarity function gives equal weights to *Names* and *FQNs* bags of words. The lexical similarity heuristic returns the top 1000 test cases TC' in *Repository* that are lexically similar to the query test case TC.

4.2.2 Reference Similarity

The types that a function is related to in the context of a test case can potentially help to understand its semantics. For example, the runtime exceptions it throws or the helper types associated with it help with understanding the behaviour of the function. These helper types that might produce (or consume) a service provided by the function under test may not necessarily be part of its interface. The interface-based retrieval technique only considers the types in the public interface of the function under test. Our reference similarity heuristic, on the other hand, also considers the

[3]http://lucene.apache.org/core/4_5_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

types in the context in which the function is tested. Two test cases are taken to have reference similarity if they use or operate on a similar set of data structures. Reference types can be primitive like **char** and **int**, standard classes defined in the JDK like String and Date, or general classes defined by 3rd party libraries and user code.

Let $Types_{TC} = \{t_1, t_2, ..., t_n\}$ be the set of types referenced in the test case TC. The function $sim_{Type}(TC, TC')$ represents the reference similarity between the query test case TC and a test case TC'.

$$0 \leq sim_{Type}(TC, TC') = \frac{|Types_{TC} \cap Types_{TC'}|}{|Types_{TC}|} \leq 1 \tag{4.3}$$

The reference similarity heuristic (eq 4.4) returns the test cases TC' in *Repository* that have references similar to the query test case TC. In our evaluations, we used all retrieved test cases in the repository (i.e., $\theta_{Type} = 0$). For a larger repository the similarity cut off value needs to be empirically adjusted.

$$Heuristic_{Type}(TC) = \{TC' \in Repository \mid sim_{Type}(TC, TC') \geq \theta_{Type}\} \tag{4.4}$$

4.2.3 Call-Set Similarity

The invocations made in the context of a test case are semantically related to the test scenario. Such invocations are not only limited to the function under test but also include helper types. Although the reference similarity heuristic considers helper types, the details of what role they play in the scenario is better captured by considering the invocations on them. Two test cases have call-set similarity if they call a similar set of methods. Let $Calls_{TC} = \{m_1, m_2, ..., m_n\}$ be the set of method invocations in a test case TC. The function $sim_{Call}(TC, TC')$ represents the call-set similarity between the query test case TC and a test case TC'.

$$0 \leq sim_{Call}(TC, TC') = \frac{|Calls_{TC} \cap Calls_{TC'}|}{|Calls_{TC}|} \leq 1 \tag{4.5}$$

The call-set similarity heuristic (eq 4.6) returns the test cases TC' in *Repository* that have invocations similar to the query test case TC. In our evaluations, we used all retrieved test cases in the repository (i.e., $\theta_{Call} = 0$). For a larger repository the similarity cut off value needs to be empirically adjusted.

$$Heuristic_{Call}(TC) = \{TC' \in \mathit{Repository} \mid sim_{Call}(TC, TC') \geq \theta_{Call}\} \tag{4.6}$$

4.2.4 Data Flow Similarity

The similarity of the dependencies between program statements in test cases (e.g., ordering of statements) is not included in the metrics discussed so far. Data flow relationships are a model for representing the coupling between program statements. In this section we introduce our static light-weight data flow similarity model customized for modelling data flow dependencies in test code. Using a combination of relational database and text representations, we have designed a naive—yet scalable—graph matching algorithm for finding similar data flow patterns.

Call Dependence Graph (CDG)

Program dependence graphs (PDGs)—program representations that encode data and control dependencies between statements and predicates—have been used in the past for finding similar code (*Krinke*, 2001), plagiarism detection (*Liu et al.*, 2006), and mining semantic clones (*Gabel et al.*, 2008). Similarity between program fragments is established based on the isomorphism of their associated PDGs. To the best of our knowledge, PDGs have never been applied to tests. We introduce call dependence graph (CDG), a customized variation of the program dependence graphs, for representing data flow dependencies between method invocations in test code[4]. A call dependence graph is in fact a model composed of the data interactions between the invocations and references in a test case. Hence, it overlaps two of our existing test similarity heuristics (i.e., call-set and reference heuristics). In Chapter 6 we will discuss if the data flow heuristic can indeed replace these two disjoint heuristics.

Definition 1. *A call dependence is a data flow relationship between two method calls m_1 and m_2 such that the data produced and returned by m_1 is passed to and consumed by m_2 as an input parameter. A* call dependence *relationship implies a temporal relationship between the two invocations; as for the data flow to be possible, m_1 should have been invoked prior to m_2.*

Definition 2. *A call dependence graph is a directed graph such that:*

1. *Each node is a method of the system under test or one of the helper types in the test scenario. Nodes have a number of attributes including the type owning the method and the method's name, return type, and parameters.*

[4]Control dependencies are not included in this version but the model can easily be extended to include them. Appendix D provides further discussion on providing support for control structures in a test data flow model.

2. *Each edge is a call dependence between two method invocations in the test scenario.*

Definition 3. *The call dependence graph of a test case TC is denoted as CDG_{TC}. The set of nodes and edges in CDG_{TC} are denoted by $V(CDG_{TC})$ and $E(CDG_{TC})$ respectively.*

Building a Call Dependence Graph

The query test case and the test cases in the repository are processed for data flow dependencies. Figure 4.2 shows a naive static analyzer for building CDG_{TC}. Existing static/dynamic data flow analyzers are designed for source code that can be compiled. A TDD test case, on the other hand, is source code that cannot be compiled. Furthermore, due to unavailability of the source code of the system under test intra-procedural data flow analysis is impossible. Therefore, we ended up designing our own naive static analyzer that extracts intra-procedural data flow dependencies. Call dependencies extracted by the analyzer are recorded to a relational database model that allows fast retrieval of call dependency graph edges. The attributes of each node are stored in an *Apache Solr* text representation that allows quick discovery of nodes with similar attributes.

Limitations Our naive static data flow analyzer is designed for efficiency. Hence it can only detect a subset of data dependencies; more specifically, it can only detect direct data flow dependencies between methods. Let **void** f(**int**) and **int** g(**void**) be two methods invoked in a test case; the call dependency between the two methods is considered direct if *(a)* the return value of g() flows into f() as an input parameter (e.g., f(g());) or *(b)* the same effect is mediated through a third-party reference (e.g., **int**

ref = g(); f(ref);). A data dependency that involves more than one reference (e.g., **int** ref1 = f(); **int** ref2 = ref1; g(ref2);) is not detected by the current algorithm. However, the algorithm can be extended to cover these scenarios as well.

```
for all $statement$ in $TC$ do
  if $statement$ is a method call $f()$ then
    if $f()$ is nested inside another method call $f'()$ then
      generate a data flow edge from $f()$ to $f'()$
    else if $f()$ is part of an expression that is assigned to a reference $x$ then
      remember the data flow dependency from $f()$ to $x$
    else if a reference $y$ is passed to $f()$ as parameter then
      if there is a data flow dependency from a method $f''()$ to $y$ then
        generate a data flow edge from $f''()$ to $f()$
      end if
    end if
  end if
end for
```

Figure 4.2: A naive static analysis algorithm for building a call dependence graph from a test case TC.

A sample test case from *Apache Commons Net*[5] project is demonstrated in Figure 4.3. The system under test in the given scenario is org.apache.commons.net.SocketClient. The two helper types java.net.Proxy and java.net.InetSocketAddress assist the system under test for the purpose of this test case. Figure 4.4 demonstrates the call dependence graph generated by our naive static algorithm for this test case.

Finding Test Cases with Similar Call Dependencies

To find test cases that have data flow similarity to a query test case TC the subgraph isomorphism problem needs to be solved. However, as this problem is NP-complete

[5]http://commons.apache.org/proper/commons-net

```
public class SocketClientTest extends TestCase
{
    private static final String PROXY_HOST = "127.0.0.1";
    private static final int PROXY_PORT = 1080;

    /**
     * A simple test to verify that the Proxy is being set.
     */
    public void testProxySettings()
    {
        SocketClient socketClient = new FTPClient();
        assertNull(socketClient.getProxy());
        Proxy proxy = new Proxy(Proxy.Type.SOCKS, new
      InetSocketAddress(PROXY_HOST, PROXY_PORT));
        socketClient.setProxy(proxy);
        assertEquals(proxy, socketClient.getProxy());
        assertFalse(socketClient.isConnected());
    }
}
```

Figure 4.3: A sample test case taken from *Apache Commons Net* project. SocketClient is the system under test and Proxy and InetSocketAddress are helper types.

in general (*Cook*, 1971) we propose a naive—yet scalable—approximate solution. Our proposed algorithm relies on a combined metric of node and edge similarity for measuring the similarity of subgraphs.

Definition 4. *The data flow similarity of a query test case Q and a candidate test case TC is measured by the degree of similarity of their call dependence graphs CDG_Q and CDG_{TC} that is denoted by the function $sim_{DataFlow}(Q, TC)$.*

The data flow heuristic $Heuristic_{DataFlow}(Q)$ returns the set of test cases TC in *Repository* that have similar data flow dependencies to a given query test case Q. The data flow similarity of the query test case Q and candidate test cases TC is measured by the function $sim_{DataFlow}(Q, TC)$. The data flow heuristic is designed for scalabil-

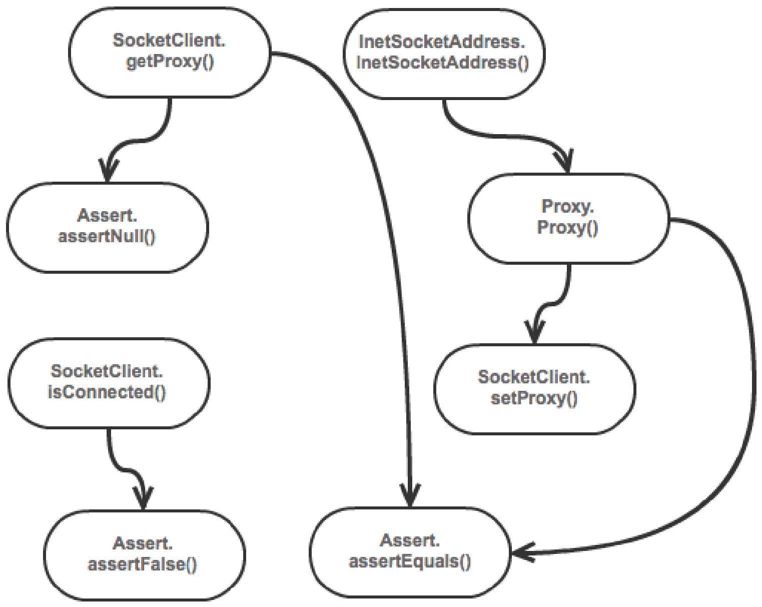


Figure 4.4: Call dependence graph generated by our naive static algorithm for the *Apache Commons Net* project test case in Figure 4.3.

ity. To allow approximate graph matching[6] in a repository of thousands/millions of graphs we made the trade-off of making our algorithm naive; instead of looking for approximate structural similarity we settled for maximal edge similarity. Hence, the algorithm returns the set of graphs in the repository that have the highest number of common edges (i.e., call dependence relationships) with a given query graph.

Our naive data flow similarity function is based on a two phase algorithm. We used a hybrid indexing method that incorporates node attributes and graph structure using text and relational representations to support the searches performed. In phase one, the algorithm finds the sets of matching nodes $Match_{e_{start}}$ and $Match_{e_{end}}$ for every edge (e_{start}, e_{end}) of the query graph CDG_Q in the graph repository. To do so, it performs lexical matching on node attributes using the text index built for all nodes

[6]Approximate graph matching is the problem of finding subgraphs in a database of graphs that are similar to the a given query graph, allowing for node mismatches and graph structural differences.

in the repository. In phase two, the similarity score of the graphs that have an edge with one node in $Match_{e_{start}}$ and another in $Match_{e_{end}}$ is increased. The relational database index is used for verifying the structure of the graph and connectivity of the nodes.

The algorithm for computing $Heuristic_{DataFlow}(Q)$ is given in Figure 4.5. In our evaluations, we used all retrieved test cases in the repository (i.e., $\theta_{DataFlow} = 0$). For a larger repository the similarity cut off value needs to be empirically adjusted.

Definition 5. *A node v in a call dependence graph constitutes of an ordered set of attributes $A(v)$ from the domain A. $A_i(v)$ is the ith attribute of v where $1 \leq i \leq |A|$.*

Definition 6. *If the similarity of two attributes a_1 and a_2 of two methods is defined by the function $0 \leq sim_{Attribute}(a_1, a_2) \leq 1$. Then the similarity of two methods v_1 and v_2 in a call dependence graph is defined as:*

$$sim_{Node}(v_1, v_2) = \frac{\sum_{i=1}^{|A|} w_i sim_{Attribute}(A_i(v_1), A_i(v_2))}{\sum_{i=1}^{|A|} w_i} \tag{4.7}$$

where w_i is the weight associated with the ith attribute.

Given a query test case Q and a candidate test case TC, the $sim_{Node}(v_1, v_2)$ similarity function is used to find the best match v_2 in CDG_{TC} for every method v_1 in call dependence graph CDG_Q. To be the best matching node, v_2 needs to have a set of attribute values that best match that of v_1. We used *Apache Solr* term matching for $sim_{Attribute}(a_1, a_2)$ on method attributes. Attributes were assigned to two bags of words for the sake of similarity comparisons. The *Names* bag of words is seeded with method name while *FQNs* is populated with method, return type, and parameter fully qualified names (see Section 4.2.1 for more details). Appendix A provides an example of how terms from methods are represented as bag of words data structures in *Apache Solr*.

```
for all test case $TC \in Repository$ do
    $sim_{Graph}(CDG_Q, CDG_{TC}) \leftarrow 0$
end for
for all data flow edge $(e_{start}, e_{end}) \in E(CDG_Q)$ do
    $Match_{e_{start}} \leftarrow \{v \in V(CDG_{TC}), TC \in Repository \mid sim_{Node}(v, e_{start}) \geq \theta_{Node}\}$
    $Match_{e_{end}} \leftarrow \{v \in V(CDG_{TC}), TC \in Repository \mid sim_{Node}(v, e_{end}) \geq \theta_{Node}\}$
    $HasSimilarNodes \leftarrow \{TC \in Repository \mid (v_1, v_2) \in E(CDG_{TC}) \wedge v_1 \in Match_{e_{start}} \wedge v_2 \in Match_{e_{end}}\}$
    for all test case $TC \in HasSimilarNodes$ do
        $maxScore \leftarrow max(sim_{Node}(v_1, e_{start}) + sim_{Node}(v_2, e_{end}))$
        $sim_{Graph}(CDG_Q, CDG_{TC}) \leftarrow sim_{Graph}(CDG_Q, CDG_{TC}) + maxScore$
    end for
end for
$Heuristic_{DataFlow}(Q) \leftarrow \{TC \in Repository \mid sim_{Graph}(CDG_Q, CDG_{TC}) \geq \theta_{DataFlow}\}$
```

Figure 4.5: A naive algorithm for finding the test cases with the most data flow similarity to a query test case Q. A cut off value of 0.5 was used for call dependence graph node similarity (θ_{Node}).

Possible Extensions

The data flow similarity function rewards test cases based on the number of call dependence relationships that they share with the query test case. The unwanted side effect of such a rewarding policy would be that larger test cases with lots of call dependence relationships get scored higher. The data flow similarity score can be normalized by the size of the candidate call dependence graph CDG_{TC} to account for the size of its test case.

$$AdjustedSim_{Graph}(Q, TC) = \frac{Sim_{Graph}(Q, TC)}{|E(CDG_{TC})|} \tag{4.8}$$

Another possible extension to the data flow similarity function can be inclusion of longer data flow paths. Currently, only the similarity of direct data dependencies is taken into consideration. A more generalized version of the data flow similarity function may also take the path $(v_1, ..., v_2)$ to be similar to (v_1, v_2). The similarity score can be discounted by the length of the distance from v_1 to v_2.

$$AdjustedSim_{Node}(v_1, v_2) = \frac{Sim_{Node}(v_1, v_2)}{d(v_1, v_2)} \tag{4.9}$$

A graph database representation can be used in order to make computation of shortest path more efficient[7].

4.3 Software Representation and Indexing

A common technique to managing high dimensional data is to reduce the dimensionality of the data using techniques such as principal component analysis (PCA). Multidimensional index structure are then utilized to support searches in the resulting low-dimensional space (*Böhm et al.*, 2001). Current software modelling techniques, including those used by the TDR approaches, rely on underlying representation schemes that are suitable for querying artifacts based on either lexical or structural features (*Hummel et al.*, 2013). Document search engines and relational databases allow lexical and some structural similarity comparisons to be efficiently performed on source code—as is done in other research on marked-up text.

Some of the most accurate software representation models created over the past decade include complex data structures involving graphs and trees (*Krinke*, 2001;

[7]For example, Neo4j graph database provides the query function shortestPath for finding the shortest path between two nodes

Mandelin et al., 2005; *Sager et al.*, 2006; *Sahavechaphan and Claypool*, 2006; *Thummalapenta and Xie*, 2007; *Gabel et al.*, 2008). However, due to the lack of an efficient large-scale graph storage and retrieval platform, in-memory representations have been used in most software recommendation and search and retrieval research. Consequently, the resulting approaches are not scalable and are limited to a single project or a small repository that can be loaded into the system memory. The alternative strategy adopted by some research—including existing TDR approaches—is to reduce the size of the search space by filtering artifacts based on their lexical and structural features. Consequently, similarity comparisons based on other features—not included in the filter—might require a sequential scan of the retrieved results if not efficiently represented. If the cost of these similarity operations is high, the sequential scan has to be limited to a subset of the results to ensure timeliness of the response when excessive number of results are retrieved.

The filtering approach to similarity search comes with a major drawback. It places a high burden on the developer to provide precise input for the features included in the filter; it will fail to retrieve pertinent source code if exact but inappropriate lexical and structural details are provided. For example, the choice of names or structures might be an arbitrary means to express the sought after semantics. Putting too much emphasis on these aspects of the query can be misleading for finding a viable match. The empirical study results indicate that behaviourally similar code of independent origin is highly unlikely to be syntactically similar (*Juergens et al.*, 2010).

4.4 Multiple Representations of Source Code

To overcome the limitations of current test-driven reuse systems, we propose a multi-representation approach to building a source code repository. According to (*Frakes and Pole*, 1994) reuse systems should employ multiple representation techniques in order to improve their efficiency:

> *Represent your collection in as many ways as you can afford. None of the methods is sufficient for finding all relevant components for a given search. Having more representations will increase the probability that relevant items will be found.*

The rationale behind using multiple representations can also be explained in terms of the implementation requirements of the similarity model. A complex model may have to be compromised—if at all possible—in order to be represented through a single representation. Breaking down a complex model into sub-models results in redundancy but at the same time it allows each sub-model to be represented by a platform and backed up by indexes that are optimized for the underlying data structures. Representing source code as text allows indexing and thus querying its lexical feature efficiently; program structures would be best represented in a relational database; and graph databases are a more suitable representation for modelling program dependencies.

In general, any representation might allow building efficient index structures for only a subset of the source code features. Hence, the strategy we propose (similar to work done in other high-dimensional spaces like multimedia (*Ishimaru and Uemura*, 1996) and geospatial (*Rigaux and Scholl*, 1994) data) is to use multiple representations of source code to facilitate different types of similarity comparisons. As demonstrated in Figure 4.6, we used three interconnected representations of source code based on two storage and retrieval platforms to model its lexical, structural, and data flow dependency features. By separating representations we made the data set more efficiently indexed and searchable according to criteria related to different feature sets. Additionally, separating feature representations improves extensibility of the model; features can be added or removed from the model independent of the existing features in the model.

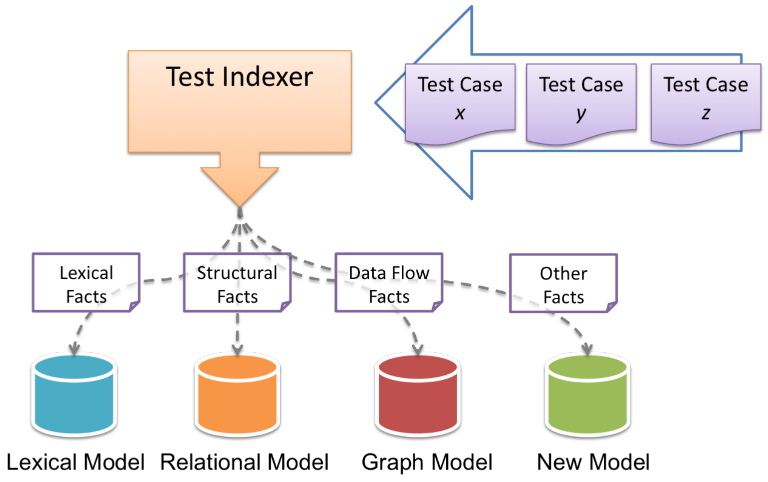


Figure 4.6: A test-driven reuse library utilizing multiple representation schemes. The model can be extended by adding new representation schemes for additional facts.

4.4.1 Terms

Representing software through a controlled vocabulary, faceted classification, or free text has been a common practice with many early software retrieval systems (*Frakes and Pole*, 1994). Most modern software search engines utilize a key-value pair storage model in which fields are populated by content and metadata extracted from the software component, its documentation, and potentially its source code. Many code search engines automatically identify and parse different programming languages. When parsing source code, terms extracted from different programming constructs can be tagged accordingly. The search query can then specify the scope of the result that a term should be present in. For example, a term can be required to match a class or method name.

We used the bag of words model that is a simplifying representation used in natural language processing and information retrieval. The terms collected from each test case are represented as an unordered collection of words. The following process is repeated at index time for each test case in the repository and at query time for the supplied query test case to generate the bag of words. First, to facilitate partial comparisons, words are split using camel case and fully qualified name (FQN) tokenizers. For example, the expression java.io.FileInputStream is split into the set of words: java, io, File, Input, and Stream. Next, to allow case-insensitive comparisons, all words are converted to lower case. Furthermore, a stemmer algorithm[8] for the English language reduces words to their stems. For example, the words *retrieval*, *retrieves*, *retrieving*, and *retrieved* are all reduced to their root word *retrieve*. Finally, duplicate words are removed from the resulting bag of words. Hence, both the order and frequency of terms is ignored in our lexical representation.

4.4.2 Structure

The relational model has traditionally been the storage model of choice when it comes to representing the structure of software. Recent code search work (*Hummel et al.*,

[8]We used the Snowball (*Porter*, 2001) stemmer algorithm.

2013; *Bajracharya et al.*, 2012; *Holmes et al.*, 2006) has suggested a few relational data models for storing source code artifacts. Our structural representation combines ideas from existing models and adds a few missing concepts (e.g., ControlFlow, DataFlow , TestMethod, and Assertion). Figure 4.7 displays the entity relationship diagram of our relational database schema.

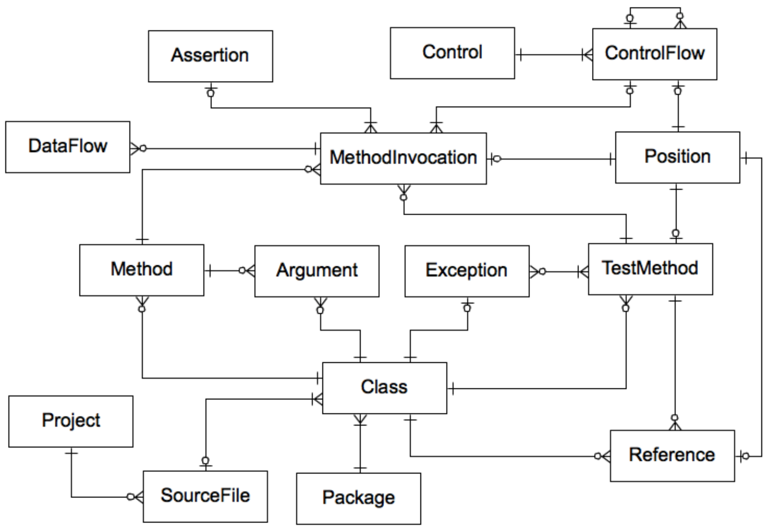


Figure 4.7: Entity relationship diagram of the structural representation.

4.4.3 Control and Data Flow

Program control and data flow dependencies are usually modelled as graph structures. Relational databases can be used for representing graphs and simple querying requirements, e.g., finding all nodes adjacent to a query node. However, querying structural properties that are essential for similarity comparisons between subgraphs (e.g., the shortest path between two nodes) becomes less efficient in a relational representation as the size of the database grows (*Vicknair et al.*, 2010). In recent years, storage and retrieval mechanisms with support for SQL-like query languages (NoSQL) have been offered for handling big data. Specifically, the popularity of online social networks have led to the emergence of graph database storage systems.

Data and control flow relationships, design patterns, object specifications and protocols, code clones, and program dependencies are examples of the structural relationships that can be modelled and queried efficiently through a graph database. Existing work in the area of software reuse recommendation systems that relies on in-memory graph representations such as (*Mandelin et al.*, 2005; *McIntyre and Walker*, 2007; *Sahavechaphan and Claypool*, 2006)—and hence are usually bound to a single project—can be scaled to mine cross project patterns in a repository of graphs. Exploring the possibility of utilizing graph databases for representing structural and semantic relationships in/between software artifacts remains a topic for future research. We used a light-weight data flow model design using the relational representation to model data dependencies. The Control, ControlFlow, and DataFlow entities in Figure 4.7 entity relationship diagram were added to represent control and data flow dependencies between method invocations in test cases.

Chapter 5

Reviver: The TDR Prototype

We built *Reviver*, a proof of concept TDR prototype, based on the ideas introduced in Chapter 4. *Reviver* is designed as an IDE-based code search infrastructure for finding existing functionality in an organizational source code repository. Developers practicing test-driven development can use this infrastructure in their development process to look up similar functionality developed in the past for reuse or as reference examples. We populated the system repository from an existing repository of open source code and evaluated its ability to identify and retrieve existing functions given their test cases (see Chapter 6).

5.1 System Architecture

Reviver is built completely on top of open-source platform-independent technology stack. It is composed of a client-side component that sits inside the developer IDE and a server-side component that hosts an indexed repository of organizational source code. The client-side was implemented as an *Eclipse*[1] IDE plug-in. It relies on the active *JUnit*[2] test case under development as the search query. Once the test case is parsed by *Fact Extractor* a search request is sent over the network to the

[1]http://www.eclipse.org
[2]http://junit.org

Query Manager on the server-side. *Query Manager* then generates multiple search queries based on different heuristics that are executed by the *Fact Manager*. Search results are collected and aggregated by *Query Manager* and sent over the network to the *View Manager* for presentation. A UML deployment diagram of the system in Figure 5.1 displays the packaging of the system components and the communication protocols used. The system repository consists of two different representations of organizational source code. The lexical representation is managed by *Apache Solr* search platform and the relational representation is hosted by *MySQL*[3] relational database management system. Access to the system repository is provided through enterprise *Java* beans (EJB) deployed inside *JBoss Application Server*[4].

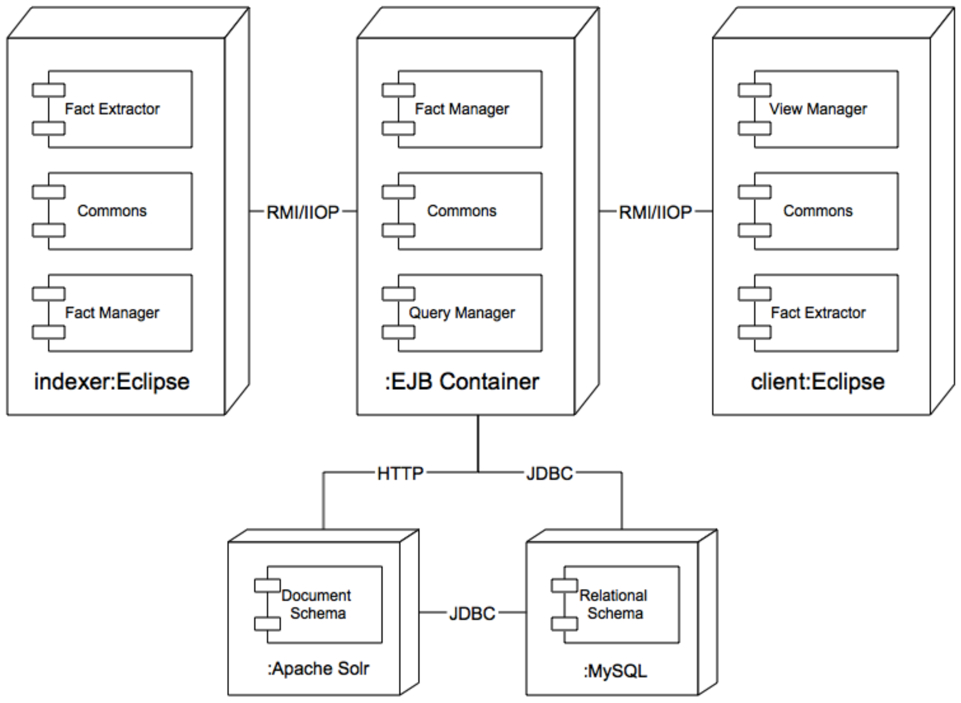


Figure 5.1: UML deployment diagram of *Reviver*, the proof of concept TDR prototype

[3]http://www.mysql.com
[4]https://www.jboss.org/jbossas

5.1.1 Fact Extractor

Fact Extractor extracts lexical, structural, and data flow facts from test cases. Fact extraction takes place both at index time when representing a test case in the repository and at query time for finding similar test cases in the repository. *Fact Manager* is implemented as an *Eclipse* JDT plug-in that parses the abstract syntax tree (AST) representation of *JUnit* test code. To improve overall performance of the indexing process, *Fact Manager* was built as a multi-threaded headless *Eclipse* plug-in that handles multiple test cases in parallel. The indexer walks through the files in a specified directory on the file system. If the *Java* class in the source file is a *JUnit* 3.x or 4.x test file then it is further processed, otherwise it is ignored. For each processed test class, all occurrences of references, invocations, exceptions, and assertions in all test methods are recorded. Furthermore, data flow dependencies between methods invoked in test cases are detected as described in Section 4.2.4.

5.1.2 Fact Manager

Fact Manager provides access to different representations of facts in the system. It is designed following the service architecture and implemented as a number of EJB components[5] deployed inside *JBoss Application Server*. Services in the *Fact Manager* are divided into two layers. The bottom layer EJB services provide access to the relational database and text representations of software artifacts in the repository discussed in Section 4.4. The *Hibernate*[6] object-relational mapping framework was used for persisting and retrieving artifact objects from the relational database. Com-

[5]EJB technology enables development of scalable, distributed, and portable server components based on *Java* platform.

[6]http://www.hibernate.org

munication with the *Apache Solr* server is facilitated through a custom built *SolrJ*[7] connector. Similarity comparisons and score computations are conducted by the top layer heuristic services (see Section 4.2 for a descriptions of test similarity heuristics).

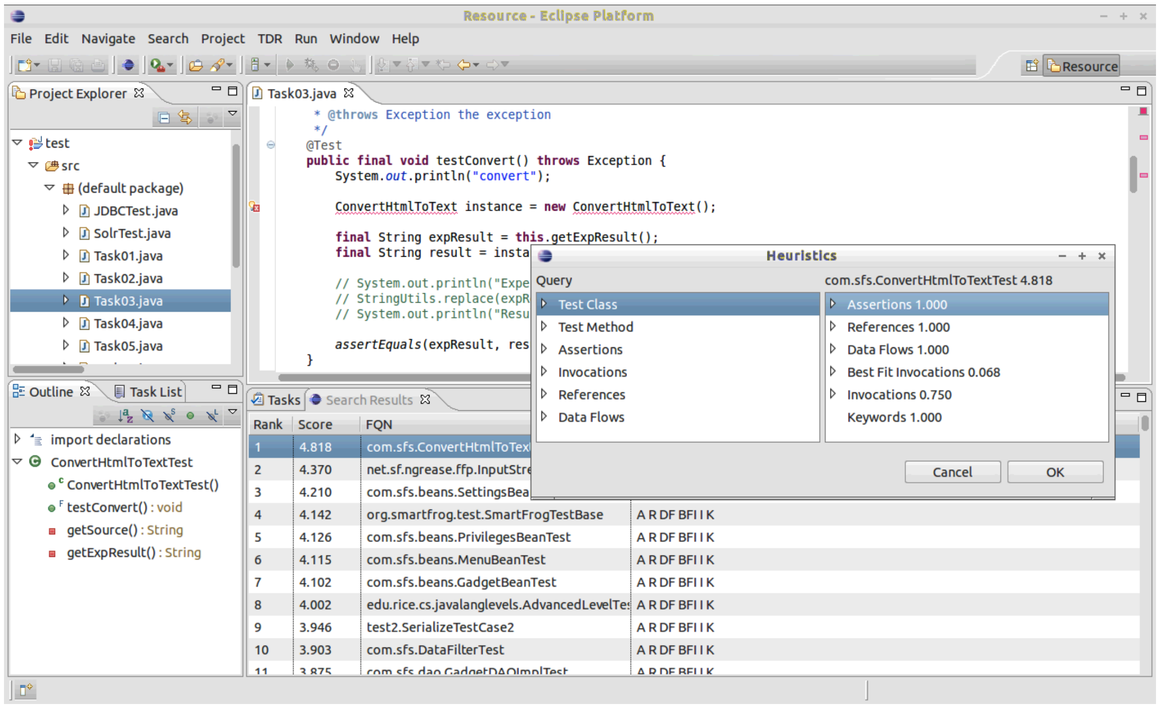
5.1.3 Query Manager

The client-side of the *Reviver* is implemented as an *Eclipse* plug-in that fires a search based on the current active *JUnit* test case. Figure 5.2 demonstrates a screen shot of the user interface of the prototype. The search query generated by the *Fact Extractor* is forwarded to the remote search service proxy inside the EJB container. Searches are performed on indexed data in relational database and text storage providers. After search results are collected for all similarity heuristics they are aggregated and returned to the *View Manager* for display. Retrieved artifacts are ranked in descending order of their aggregated similarity score. The searcher can then browse through artifacts and see the list of relevant items that contributed to the selection of each.

5.2 Repository

The structure and graph representations of the assets in our repository is managed by a *MySQL* database instance. Use of the *Hibernate* object-relational mapping framework in the *Fact Manager* however makes the system portable and independent of any particular database engine. The text representation of the assets is managed by an *Apache Solr* instance hosting two schemas. The main schema defines the lexical elements of test cases utilized by the lexical similarity heuristic (see Section 4.2.1). A second schema defines attributes of method signatures for the sake of finding similar

[7]http://wiki.apache.org/solr/Solrj



```
     * @throws Exception the exception
     */
    @Test
    public final void testConvert() throws Exception {
        System.out.println("convert");

        ConvertHtmlToText instance = new ConvertHtmlToText();

        final String expResult = this.getExpResult();
        final String result = insta

        // System.out.println("Expe
        // StringUtils.replace(expR
        // System.out.println("Resu

        assertEquals(expResult, res
    }
```

Query	com.sfs.ConvertHtmlToTextTest 4.818
Test Class	Assertions 1.000
Test Method	References 1.000
Assertions	Data Flows 1.000
Invocations	Best Fit Invocations 0.068
References	Invocations 0.750
Data Flows	Keywords 1.000

Rank	Score	FQN	
1	4.818	com.sfs.ConvertHtmlToTex	
2	4.370	net.sf.ngrease.ffp.InputStre	
3	4.210	com.sfs.beans.SettingsBea	
4	4.142	org.smartfrog.test.SmartFrogTestBase	A R DF BFI I K
5	4.126	com.sfs.beans.PrivilegesBeanTest	A R DF BFI I K
6	4.115	com.sfs.beans.MenuBeanTest	A R DF BFI I K
7	4.102	com.sfs.beans.GadgetBeanTest	A R DF BFI I K
8	4.002	edu.rice.cs.javalanglevels.AdvancedLevelTes	A R DF BFI I K
9	3.946	test2.SerializeTestCase2	A R DF BFI I K
10	3.903	com.sfs.DataFilterTest	A R DF BFI I K
11	3.875	com.sfs.dao.GadgetDAOImplTest	A R DF BFI I K

Figure 5.2: The client user interface of *Reviver*, the proof of concept TDR prototype

nodes in call dependence graphs in the data flow heuristic (see Section 4.2.4). Populating the *Apache Solr* index takes place once all source code is parsed and indexed in the *MySQL* database. A custom built importer links up *Apache Solr* with *MySQL* through *JDBC*.

5.2.1 Content

To help with evaluating the prototype the repository was populated with a research data set collected from open source projects. We used a subset of the *Merobase* code repository (*Janjic et al.*, 2013) in our experiments. The *Eclipse* JDT cannot build a complete AST of *Java* source code unless it is compilable. Previous research has come up with approaches for resolving missing dependencies in source code retrieved from the Internet (*Ossher et al.*, 2010). We relied on an alternative approach by using the *Eclipse* plug-in provided by the partial program analysis (PPA)[8] project (*Dagenais and Hendren*, 2008). PPA cannot detect the fully qualified name of all types referenced in the indexed source code and hence introduces some level of error. However, the effort required for parsing the source files was considerably reduced this way.

[8]Partial Program Analysis for *Java* is a static analysis framework that transforms the source code of an incomplete *Java* program into a typed abstract syntax tree.

Chapter 6

Evaluation

The goal of our evaluation is to validate the hypothesis that our semantic heuristics utilizing multiple representations of source code enhance the effectiveness of test-driven reuse by retrieving better results. To validate this hypothesis, we conducted a controlled experiment using the ten test cases from Chapter 3, each of which had 16 variations. We evaluated the effectiveness of the approach in retrieving approximations of input test cases by providing it with transformations of the tests. To compare the efficiency of our approach to that of existing TDR approaches we made an ad hoc implementation of the interface-based retrieval technique—the combination of signature matching and keyword-based retrieval generally used by these approaches. Test cases were provided as search queries to both systems; the rank of the correct result was recorded in each case. We experimented with different configurations of *Reviver* to see what combination of heuristics makes it more effective. We found that overall a configuration of *Reviver* using lexical and data flow similarity heuristics is more likely to find an existing implementation of the function under test. An analysis of evaluation results and discussion of limitations of the experiment is provided in the conclusion.

6.1 Experiment

Given an input test case, the interface-based retrieval approach tries to find an instance of the system under test only using the facts related to its interface; while the domain specific TDR approach also considers additional facts related to the test case that exercises the system under test. We decided to design an experiment to compare the ability of the two approaches in identifying the associated functions under test by providing a set of input test cases exercising those functions. However, due to unavailability of two of the three TDR prototypes[1], we ended up building an ad hoc implementation of the interface-based retrieval technique based on the ideas used in these approaches.

We used the same of set of tasks we used in Section 3.1.1 for evaluating existing TDR approaches. Each task is composed of a minimized and anonymized *JUnit* test case. Table 6.1 shows the interface of the function under test in the ten trial test cases. A short description of the tasks is provided in Table 3.1. For each task, the unmodified version of each test case and all classes under test were added to the system repository. The baseline objective of the evaluation is to see if the system is able to identify and recommend the function in Table 6.1 once given the anonymized version of the original test case. Furthermore, to evaluate the ability of the approaches in retrieving approximations of the search query we also provided each system with transformations

[1]All tools are offered as online services. *CodeGenie* was offered as a search client for the *Sourcerer* code search engine. The recent developments in the *Sourcerer* project has forced the *CodeGenie* sub-project into retirement. Although the latest source code for the *Sourcerer* project is available however it cannot be used to set up an experiment locally because it no longer supports *CodeGenie*. Likewise, the *Code Conjurer* search client remotely connects to the *Merobase* code search engine. The new version of *Code Conjurer* is under development while due to changes to the search gateway the previous version is no longer functional. Due to unavailability of the source code and configuration of the *Merobase* code search engine, setting up a local instance of *Merobase* for the sake of experiments is not an option.

of the original test cases. Test cases were transformed along name, type, scenario, and protocol dimensions according to the rules described in Section 6.1.2.

Our repository contains 43,826 test methods collected from 4,768 *JUnit* test classes. Tests were automatically selected from the SVN repository of the *Merobase* project dataset (*Janjic et al.*, 2013). Table 6.2 gives an overview of the statistical properties of the content loaded into repository for the evaluation study. The *Apache Solr* index that supported the interface-based retrieval prototype was populated with the terms from the interfaces of 15,119 classes that appeared in test cases as a reference, exception, method argument, or method return type.

In Section 5.2.1 we described the process of statically processing test code in the *Reviver* repository. Due to limitations of partial program analysis (*Dagenais and Hendren*, 2008) that we relied upon for this purpose, many types were not identified and hence listed as the default UKNOWN type. Likewise, when the package that a type belongs to is not identified it is listed as UNKNOWNP. The fully qualified name of a type that is completely unclassified (i.e., neither the class nor the package is identified) is listed as UKNOWNP.UNKNOWN. Table 6.3 lists the percentage of program elements that could not be classified at index time. For reference, method, and argument entities roughly 10 percent or less of the types were not classified. The PPA parser runs into trouble resolving method return value types more than anything else; the return value type of slightly more than half of the methods invoked in the test cases were not classified.

Table 6.1: The interfaces of the function under test extracted from the ten trial test cases used in the evaluation. The name of the function under test and its method signatures are indicated in *Class Name* and *Methods* columns, respectively.

Task	Class Name	Methods

1	Base64Util	static String encodeString(String);
		static String decodeString(String);
		static String encode(byte[]);
		static byte[] decode(String);

2	DateUtils	static int getNumberOfDaysBetweenTwoDates(Date,Date);

3	ConvertHtmlToText	ConvertHtmlToText();
		String convert(String);

4	TypeValidatorUtil	TypeValidatorUtil();
		boolean isCreditcardNumber(String, CreditcardType);

5	StringBag	StringBag();
		void add(String);
		boolean remove(String);
		int occurencesOf(String);
		int size();
		String toString();

6	Diff	Diff(String, String);
		boolean similar();
		boolean identical();

7	IpFilter	IpFilter();
		void setFilters(String, String);
		boolean isIpAllowed(String);

8	SQLInjectionFilterManager	static SQLInjectionFilterManager getInstance();
		String filter(String);

9	Utilities	Utilities();
		Map getWordFrequency(String);
		Map getWordFrequency(String, boolean);

10	CmdLineParser	CmdLineParser();
		CmdLineParser.Option addBooleanOption(char, String);
		CmdLineParser.Option addIntegerOption(char, String);
		CmdLineParser.Option addStringOption(char, String);
		CmdLineParser.Option addDoubleOption(char, String);
		CmdLineParser.Option addBooleanOption(char, String);
		CmdLineParser.Option addBooleanOption(String);
		CmdLineParser.Option addLongOption(char, String);
		parse(String[], Locale);
		Object getOptionValue(CmdLineParser.Option);
		String[] getRemainingArgs();

Table 6.2: An overview of the statistical properties of the repository. For each parameter the total, the minimum, maximum, average, and standard deviation are indicated in *Total*, *Min*, *Max*, *Avg*, and *Stdev* columns, respectively.

Parameter	Total	Min	Max	Avg	Stdev
# methods per *JUnit* test class	43,826	1	180	9.21	12.91
# references per test method	149,129	1	159	4.09	3.73
# invocations per test method	364,701	1	642	9.16	12.67
# assertions per test method	88,623	1	106	3.52	4.82
# exceptions per test method	25,367	0	8	1.16	0.50
# data flow dependencies per test method	157,060	0	420	4.86	6.21

Table 6.3: Unclassified entities in the repository. For each entity type the number of unclassified instances, the total number of occurrences, and the percentage of unclassified instances are indicated in *Unclassified*, *Total*, and *Percentage* columns, respectively.

Entity	Unclassified	Total	Percentage
Reference	15,438	149,129	10.35%
Method invocation	38,629	364,701	10.59%
Method return type	31,044	58,347	53.21%
Method argument	4,980	57,344	8.68%

6.1.1 Interface-based Retrieval Prototype

Existing TDR approaches rely on keyword searches enhanced by signature matching, a technique also known as interface-based or interface-driven retrieval. Unfortunately, bugs in the prototypes and unavailability due to on-going development or retirement of the project makes conducting a comparative study on these approaches very difficult, if not impossible. We decided to build an ad hoc prototype of the interface-based retrieval technique based on the description given in TDR literature (*Hummel*, 2008; *Bajracharya*, 2010; *Reiss*, 2009a). Our simple ad hoc implementation of interface-based retrieval does not come with the method parameter matching feature proposed in (*Hummel et al.*, 2013) but would otherwise measure up to what is offered by the existing prototypes.

We used text representation, as is used by existing TDR approaches, for representing classes under test. Bag of words data structures, similar to the ones we used in our lexical and data flow similarity heuristics, hold attributes of the interface of the indexed classes. Tokenization, stemming, and duplicate removal are applied to all words, as was described in Section 4.2.1. Table 6.4 gives an overview of the interface attributes and the bag of words data structure they are assigned to in our interface-based retrieval prototype. All attributes are weighted equally when computing similarity scores. We processed the test cases in the repository and indexed the interface of the class under test. More specifically, the class names, method names, parameter types, and return types were indexed in each case. Appendix A provides an example of how terms in a class interface are represented as bag of words data structures in *Apache Solr*.

Table 6.4: The attributes of the classes under test indexed by the interface-based retrieval prototype. The *JUnit* test case element and the property from which the attribute was collected from are indicated in *Test Element* and *Property* columns, respectively. The *Apache Solr* bag of words structure utilized for indexing each attribute is indicated in the *Bag of Words* column.

Test Element	Property	Bag of Words
Class under test	Name	ClassFQNs
	Method name	MethodNames
	Method argument FQNs	ArgumentFQNs
	Method return type FQNs	ReturnTypeFQNs

Apache Solr allows logical operators like AND and OR in input queries. Existing TDR prototypes relying on interface-based retrieval either connect all terms in the query through the AND operator or just do this initially and if nothing is retrieved try to relax the query by replacing the AND operators with OR. For example, an interface-based search for a Calculator class with add, subtract, multiply, and divide methods will result in the search query Calculator AND add AND subtract AND multiply AND divide. However, if not such Calculator class can be found the search query is relaxed to Calculator OR add OR subtract OR multiply OR divide.

Our initial experiments indicated that although the use of the AND operator serves best for retrieving exact matches but at the same time it is going to make retrieving any other variation impossible. As soon as a new term is introduced to the input query as a result of one of the transformations the resulting query will not match anything in our repository. In the example above, the query using AND operators will not match a class named Computer that would have the same four methods add, subtract, multiply, and divide. We use the OR operator between all query terms in our prototype implementation of the interface-based retrieval. Despite the fact that it negatively impacts the rank of the correct result once no transformation is applied, it makes retrieving transformed versions of the input query possible.

6.1.2 Test Case Transformations

We devised a technique for generating approximations of test cases by modifying their underlying features, in order to compare the ability of *Reviver* with that of the interface-based retrieval in matching approximations of an input query. Transformations refactor and restructure code but do not modify the semantics of the function under test. In other words, the idea of transforming a test case is to generate semantically equivalent functions under test with alternative designs and testing scenarios. In real world situations a searcher might formulate a search query in many different ways. Transformations help with evaluating and comparing retrieval algorithms. We can study if a code search mechanism can retrieve a target function given the transformed version of the query. Ideally, differences in program vocabulary, types, and design should not exclude an asset from search results if it semantically satisfies the search query.

The number of modifications that can be made during each transformation is virtually unlimited. We identified the attributes of the input test cases that each algorithm relied on for retrieving similar assets. Attributes were classified into four general categories: names, types, scenario, and protocol. A transformation was assigned to each category to modify the attributes, hence resulting in four transformations. The four transformation types are orthogonal and can be combined. We considered all possible combinations of the four transformation types resulting in a total of 15 transformations. To allow more fine tuned evaluation of code search and retrieval system, future research should look into automating the process of generating transformed variations of input queries. For the sake of our evaluation we performed transformations manually.

Given the 10 original test cases, we came up with 160 (150 transformed test cases plus 10 original test cases) input queries for our evaluation study. Examples of application of the four transformation types to a test case are provided in Appendix B. In the rest of this section we will describe the four transformation types.

Name Transformation

Name transformation generates a variant of a given test case that utilizes alternative terminology. It modifies the choice of names for the function under test and that of elements of the test case. Element names can be changed to their synonyms, random words taken from a repository of programming element names, or randomly generated based on a dictionary. The names of class, methods, and public fields of the function under test are modified during this transformation. Similarly, the names of the class, test methods, fields, local variables, and literals of the test class exercising the function under test are modified. Table 6.5 illustrates the set of rules for modifying name and literal attributes of a test case. Appendix B provides an example of applying name transformation rules to a test case.

Table 6.5: Test case name transformation (N) rules. The element to which the rule is applied and the set of modifications that take place on the element are indicated in *Element* and *Transformation Rule* columns, respectively.

Element	Transformation Rule
Class under test	Rename to `C`
Instance of the class under test	Rename to `c`
Methods of class under test	Rename to `m1`, `m2`, etc. in order of appearance
Test class	Rename to `CTest`
Test methods	Rename to `tm1`, `tm2`, etc. in order of appearance
Fields, local variables, and method arguments	Rename to `var`, `var1`, `var2`, etc. in order of appearance (variable names can be reused if the contexts in which they appear does not overlap)
Literal	Change to arbitrary values that do not break test logic

Type Transformation

Type transformation generates a variant of a given test case that utilizes alternative types and references. It modifies the choice of types and APIs for the function under test and the test case exercising it. Types can be refactored to compatible types from *JDK* or other APIs or arbitrary types as long as it does not break the test logic. Parameterized types need to have both the main type and the parameter to be refactored. The types of public fields, method arguments, and method return values of the function under test are modified by this transformation. Similarly, the types of fields, local variables, helper classes, and helper methods of the test class exercising the function under test are modified. Table 6.6 illustrates the set of rules for modifying the types in test cases. Appendix B provides an example of applying type transformation rules to a test case.

Table 6.6: Test case type transformation (T) rules. The element to which the rule is applied and the set of modifications that take place on the element are indicated in *Element* and *Transformation Rule* columns, respectively.

Element	Transformation Rule
Test class	Add new fields/variables with arbitrary types
Test class fields and local variables	Refactor to other compatible types/APIs
Method parameters of the function under test	Refactor types, count, and order
Method return type of the function under test	Refactor type
Public fields of the function under test	Refactor type

Scenario Transformation

Scenario transformation generates a variant of a given test case that utilizes an alternative scenario for testing the function under test. It modifies how the function under test is exercised in a test case. A unit test class relies on a number of architectural elements like fixture setup and test helpers for supporting operations in the test cases. The factoring of the test scenario to setup, helper, and test methods is modified by the scenario transformation. In addition, the number of scenarios and the conditions that they depend on is altered. Table 6.7 gives an overview of the rules for modifying the testing scenario in test cases. Appendix B provides an example of applying scenario transformation rules to a test case.

Table 6.7: Test case scenario transformation (S) rules. The element to which the rule is applied and the set of modifications that take place on the element are indicated in *Element* and *Transformation Rule* columns, respectively.

Element	Transformation Rule
Fixture setup method	Combine with test method
Helper method	Combine with test method
Test method	Refactor into fixture setup and helper methods
Test case inside test method	Add a new scenario, remove an existing scenario, change data/conditions, or modify assertions of an existing scenario

Protocol Transformation

Protocol transformation generates a variant of a given function under test that has alternative design. It modifies the protocol design and design patterns (*Gamma et al.*, 1994) associated with the function under test in a test case. The behaviour of a function can be altered by changing the selection of methods offered and ordering of the calls. For example, a dependency between two methods `m1` and `m2` that makes `m1` a precondition of `m2` can be replaced by a new method `m3` that combines `m1` and `m2`. Creational, structural, and behavioural design patterns can be taken advantage of in creating semantically equivalent functions. For example, using the creational design patterns singleton and factory method we modified how a function is instantiated. Alternatively, by making the method calls static we removed the need for instantiation in some scenarios. Table 6.8 gives an overview of the rules for modifying the protocol of the function under test in test cases. Appendix B provides an example of applying protocol transformation rules to a test case.

Table 6.8: Test case protocol transformation (P) rules. The element to which the rule is applied and the set of modifications that take place on the element are indicated in *Element* and *Transformation Rule* columns, respectively.

Element	Transformation Rule
Constructor of the function under test	Replace with singleton or factory method
Method of the class under test	Make static if instance method; or make instance method if static
	Split into two or more methods where possible
	Combine with another method where possible
Class under test	Add new methods

6.2 Results

Tables 6.9 and 6.10 summarize the results of the evaluation experiment for the interface-based retrieval prototype and *Reviver*, respectively. For each of the 10 evaluation tasks we supplied the original and 15 transformed test cases as input queries to both prototypes and recorded the rank of the correct result in the recommendations. Tables indicate the rank of the correct result for every possible combination of name, type, scenario, and protocol transformations in addition to the original test case. Only the first one thousand results produced by each prototype were retrieved in every session. If the correct result was not found in the first one thousand results the rank is left blank in the tables. The bottom row in Table 6.10 lists the time in seconds for extracting the top one thousand results for all 16 variations of a task from the repository. The results were collected from an *Ubuntu* 13.10 running on a 8 core *Intel Xeon* 3.20GHz box with 8GB of RAM.

In order to facilitate the analysis and comparison of the results we decided to focus on cases in which the correct result was ranked among the top ten items retrieved. There is evidence that developers do not look beyond this point in search results (*Joachims et al.*, 2005). Table 6.11 provides a side-by-side comparison of the experiment results for the two approaches. For each pair of task and transformation combination, the rank of the correct result for the interface-based retrieval prototype (left) and *Reviver* (right) are given. Only ranks among the top ten are indicated; others are marked with a dash sign. The total column and row indicate the count of instances that the correct result was ranked among the top ten for each transformation combination and task, respectively. In some cases the rank of correct result has been improved despite application of transformations. We verified that if a frequently

Table 6.9: The result of the experiment for 10 tasks and 16 transformation combinations for the interface-based retrieval prototype. The numbers in columns indicate the rank of the correct result for each task. If the correct result was not found in the first 1000 items retrieved the rank is marked by a dash sign. In the *Transformations* column, *O*, *N*, *T*, *S*, and *P* stand for original, name, type, scenario, and protocol transformations respectively.

	Tasks									
Transformations	1	2	3	4	5	6	7	8	9	10
O	3	1	1	1	13	30	2	2	1	1
N	58	5	403	-	685	-	621	455	11	2
T	3	1	1	1	26	7	13	2	1	1
S	3	1	2	1	17	45	1	1	1	1
P	3	1	2	1	20	1	2	93	5	1
NT	-	317	-	-	-	-	-	-	177	26
NS	46	5	649	-	685	-	686	-	27	2
NP	70	5	425	-	743	-	621	779	906	56
TS	3	1	1	1	26	7	2	1	1	1
TP	2	1	2	1	27	1	4	40	3	1
SP	2	1	5	1	22	1	1	5	3	1
NTS	-	317	-	-	-	-	-	-	625	26
NTP	-	-	523	-	563	-	-	-	-	56
NSP	38	27	468	-	693	-	686	-	908	56
TSP	2	1	2	1	30	1	2	11	3	1
NTSP	-	-	688	-	-	-	-	-	-	20

occurring fact is eliminated from the query as a result of the application of a transformation then it can positively impact the rank of the correct result in the result set. This effect can be attributed to the the inverse document frequency (*Manning et al.*, 2008) component in the computation of the lexical similarity metric. It results in reduced lexical similarity score when the intersection of the search query and the document representation of the correct test case is composed of frequent terms.

Table 6.10: The result of the experiment for 10 tasks and 16 transformation combinations for *Reviver*. The numbers in columns indicate the rank of the correct result for each task. If the correct result was not found in the first 1000 items retrieved the rank is left blank. In the *Transformations* column, *O*, *N*, *T*, *S*, and *P* stand for original, name, type, scenario, and protocol transformations respectively. The *Time* row at the bottom indicates the total time in seconds for extracting the first 1000 results for all 16 variations of a task from the repository.

	Tasks									
Transformations	1	2	3	4	5	6	7	8	9	10
O	1	1	1	1	1	1	1	1	1	1
N	1	1	18	5	709	-	1	105	2	1
T	3	1	1	1	1	1	16	4	28	1
S	1	1	1	1	1	1	1	1	1	1
P	1	1	1	1	1	832	1	4	78	1
NT	-	917	353	-	-	-	-	-	-	60
NS	3	1	18	114	709	-	-	885	3	1
NP	5	5	18	4	498	-	1	105	223	1
TS	1	1	1	319	1	10	12	1	23	1
TP	2	1	1	1	1	137	13	21	486	9
SP	1	1	1	1	1	837	1	1	78	1
NTS	-	-	802	-	-	-	-	-	-	77
NTP	-	-	297	-	-	-	-	-	-	78
NSP	46	6	18	113	340	-	-	885	223	1
TSP	1	1	1	314	1	140	13	74	486	1
NTSP	-	-	806	-	-	-	-	-	-	34
Time (sec)	1:42	1:48	3:34	1:20	1:50	0:32	0:28	2:16	2:34	3:30

Table 6.11: The result of the experiment for each task and transformation combination. The numbers in columns indicate the rank of the correct result for each task. For each task, the left column represents the interface-based retrieval prototype and the right column represents *Reviver*. If the correct result was not placed in the top 10 items retrieved the rank is marked by a dash sign. In the *Transformations* column, O, N, T, S, and P stand for original, name, type, scenario, and protocol transformations respectively. The *Total* column and row show the number of cases in which the correct result was retrieved in the top 10 results for each transformation combination and task, respectively.

	Tasks																					
Transformations	1		2		3		4		5		6		7		8		9		10		Total	
O	3	1	1	1	1	1	1	1	-	1	-	1	2	1	2	1	1	1	1	1	8	10
N	-	1	5	1	-	-	-	5	-	-	-	-	-	1	-	-	-	2	2	1	2	6
T	3	3	1	1	1	1	1	1	-	1	7	1	-	-	2	4	1	-	1	1	8	8
S	3	1	1	1	2	1	1	1	-	1	-	1	1	1	1	1	1	1	1	1	8	10
P	3	1	1	1	2	1	1	1	-	1	1	-	2	1	-	4	5	-	1	1	8	8
NT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
NS	-	3	5	1	-	-	-	-	-	-	-	-	-	-	-	-	-	3	2	1	2	4
NP	-	5	5	5	-	-	-	4	-	-	-	-	-	1	-	-	-	-	-	1	1	5
TS	3	1	1	1	1	1	1	-	-	1	7	10	2	-	1	1	1	-	1	1	9	7
TP	2	2	1	1	2	1	1	1	-	1	1	-	4	-	-	-	3	-	1	9	8	6
SP	2	1	1	1	5	1	1	1	-	1	1	-	1	1	5	1	3	-	1	1	9	8
NTS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
NTP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
NSP	-	-	-	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	2
TSP	2	1	1	1	2	1	1	-	-	1	1	-	2	-	-	-	3	-	1	1	8	5
NTSP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
Total	8	11	11	12	8	8	8	8	0	8	6	4	7	6	5	6	8	4	10	12	71	79

6.2.1 Heuristic Selection

Feature selection (*Blum and Langley*, 1997) is the process in which the redundant and irrelevant features are removed from the model. Our test similarity model might contain redundant heuristics. Redundant heuristics are those that provide no more information that the currently selected ones. If a heuristic selects irrelevant results then the overall performance of the model can be improved by removing it from the model. Furthermore, the aggregated similarity function in Equation 4.2 can be optimized by properly adjusting the relative importance (i.e., weights) of heuristics based on the data in the repository. Machine learning techniques like reinforcement learning, Bayesian methods, or Genetic Algorithms (GA) can be used to optimize the weights in the similarity function given the data in the repository (*Richter and Weber*, 2013).

To select heuristics and adjust weights, data in the repository is split into training and validation sets. The training data set is first used for heuristic selection and weight learning. The error rate in the validation set is then used to measure the fitness of the model. Alternatively, a cross-validation technique like k-fold cross validation[2] can be used.

Manual generation of test case transformations poses a major limitation for applying feature selection and weight learning techniques to our data set. To train and cross-validate the test similarity model the 15 combinations of the four test case transformations have to be generated for a large (if not the entire) subset of the repository;

[2]In k-fold cross-validation, the repository is randomly split into k equal size subsets. Of the k subsets, a single subset is retained as the validation data for testing the model, and the remaining k-1 subsets are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsets used exactly once as the validation data. The average error rate of the folds is then used as a single estimation.

a task that would not be possible unless the process of generating transformations is automated. Unfortunately, automating the transformation process is beyond the scope of this thesis. Therefore, we decided to experiment with the small manually generated data set of the ten trial tasks we originally used to evaluate the existing TDR approaches.

We evaluated and compared the performance of *Reviver* on the ten trial tasks and their transformations with all 15 possible combinations of the four heuristics. Tables 6.12 and 6.13 provide the result of this evaluation. The numbers in columns indicate the number of tasks (Table 6.12) and transformations (Table 6.13) for which the correct result was placed in the top 10 items retrieved by each heuristic combination. The *Total* row shows the number of cases in the ten evaluation tasks and their 15 transformations (between 0 and 160) that the correct result was ranked in the top 10 items by each heuristic combination. As it can be seen in both tables the combination of the data flow and lexical similarity heuristics offers the best performance over the ten trial tasks.

6.3 Analysis

Figure 6.1 summarizes the number of correct recommendations made by the two prototypes[3]. For every combination of test case transformations the number of tasks (i.e., between 0 and 10) that made it to the top ten recommendations are shown as columns in the bar chart. The bar chart in Figure 6.2 provides an alternative view of the same data; the number of transformations (i.e., between 0 and 16) for which the

[3]Numbers in the two charts for *Reviver* belong to our best performing configuration—the combination of data flow and lexical similarity heuristics.

Table 6.12: Comparing the performance of all possible combinations of the four heuristics on the ten trial tasks and their transformations. The numbers in the columns indicate the number of tasks (between 0 and 10) for which the correct result was placed in the top 10 items retrieved by each heuristic combination. In the *Heuristics* columns, *R*, *I*, *D*, and *K* stand for reference, call-set, data flow, and lexical similarity heuristics respectively. In the *Transformations* column, *O*, *N*, *T*, *S*, and *P* stand for original, name, type, scenario, and protocol transformations respectively. The *Total* row shows the number of cases in the ten evaluation tasks and their 15 transformations (between 0 and 160) that the correct result was ranked in the top 10 items by each heuristic combination. An alternative view of this data is provided in Table 6.13.

	Heuristics														
Transformations	RIDK	RID	RIK	RDK	IDK	RI	RD	RK	ID	IK	DK	R	I	D	K
O	10	10	8	10	10	8	10	9	10	9	10	9	6	10	10
N	6	9	4	6	5	8	9	3	7	3	5	9	5	7	2
T	8	4	4	9	10	1	6	5	7	7	10	0	1	7	10
S	10	8	8	10	10	7	8	9	9	10	10	6	6	9	10
P	8	8	8	8	9	8	8	9	8	9	9	9	5	8	10
NT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NS	4	5	3	4	4	6	5	1	5	3	4	5	5	5	2
NP	5	8	3	5	5	7	8	3	6	2	5	8	4	6	2
TS	7	4	4	8	8	2	5	6	6	5	10	0	2	6	10
TP	6	3	4	8	8	1	4	6	4	7	9	1	1	4	10
SP	8	6	8	8	9	7	6	9	7	9	9	6	5	5	10
NTS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NTP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NSP	2	4	2	1	2	5	4	2	4	2	2	6	4	3	2
TSP	5	2	3	7	6	1	3	6	3	5	9	1	1	3	10
NTSP	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1
Total	79	71	59	84	86	61	76	68	76	72	92	60	46	73	89

Table 6.13: Comparing the performance of all possible combinations of the four heuristics on the ten trial tasks and their transformations. The numbers in the columns indicate the number of transformation combinations (between 0 and 16) for which the correct result was placed in the top 10 items retrieved by each heuristic combination. In the *Heuristics* columns, *R*, *I*, *D*, and *K* stand for reference, call-set, data flow, and lexical similarity heuristics respectively. In the *Transformations* column, *O*, *N*, *T*, *S*, and *P* stand for original, name, type, scenario, and protocol transformations respectively. The *Total* row shows the number of cases in the ten evaluation tasks and their 15 transformations (between 0 and 160) that the correct result was ranked in the top 10 items by each heuristic combination. An alternative view of this data is provided in Table 6.12.

						Transformations									
Tasks	RIDK	RID	RIK	RDK	IDK	RI	RD	RK	ID	IK	DK	R	I	D	K
Task01	11	11	4	11	11	8	12	5	12	7	11	8	8	11	8
Task02	12	8	12	12	12	8	8	12	8	12	12	8	8	8	12
Task03	8	7	5	8	8	6	12	8	12	5	8	8	8	12	8
Task04	8	8	8	8	7	8	8	8	6	6	9	8	0	6	8
Task05	8	10	8	8	8	10	10	8	8	8	8	8	8	8	8
Task06	4	3	0	4	7	0	3	0	4	6	8	0	0	4	8
Task07	6	4	4	10	8	4	4	6	4	4	10	4	0	4	8
Task08	6	4	0	8	8	0	6	6	8	1	8	4	0	8	8
Task09	4	6	6	4	6	8	6	4	4	10	6	7	4	4	8
Task10	12	10	12	11	11	9	7	11	10	13	12	5	10	8	13
Total	79	71	59	84	86	61	76	68	76	72	92	60	46	73	89

correct result made it to the top ten recommendations is shown for each task.

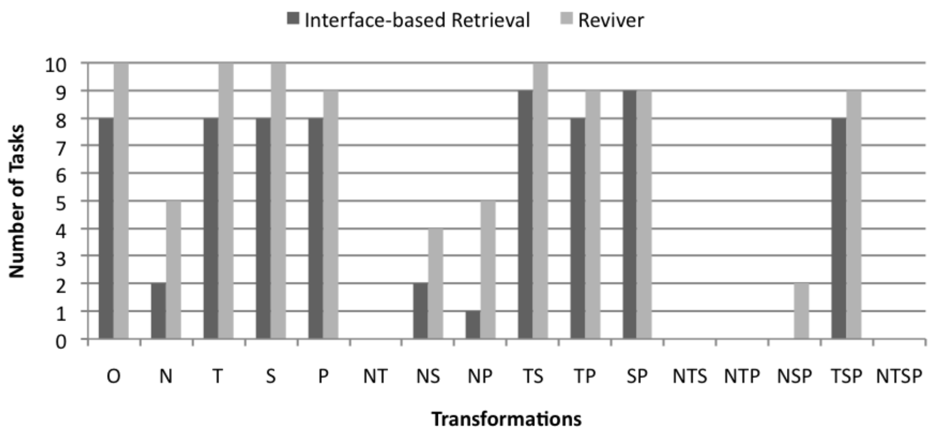


Figure 6.1: The number of tasks successfully retrieved for every transformation combination.

Looking at results of evaluation for different combinations of heuristics, one can see the lexical similarity heuristic—with 89 correct results—is the second best combination amongst all. A plausible question that therefore arises is whether the data flow similarity heuristic in the combination of the two is playing any significant role. Another related question is whether either of the dismissed reference and call-set similarity heuristics retrieved any correct results that data flow and lexical similarity heuristics did not cover. Table 6.14 provides a side-by-side comparison of the four similarity heuristics. Comparing the results of data flow similarity with those of the lexical similarity revealed that heuristics share 59 correct results. In addition, the data flow and lexical similarity heuristics separately returned 14 and 30 correct results respectively. Hence, the data flow similarity heuristic provides utility to the model by retrieving correct results that the lexical similarity heuristic could not retrieve.

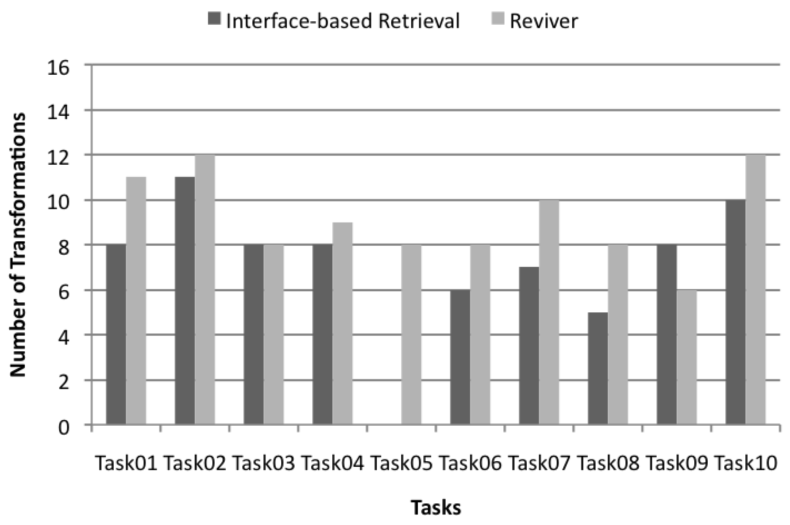


Figure 6.2: The number of transformation combinations successfully retrieved for every task.

Interestingly, the call-set similarity heuristic did not return any correct results that were not already covered by the lexical and data flow similarity heuristics but the reference similarity heuristic produced 8 correct results that neither of the two heuristics identified. Therefore, the reference similarity heuristic has to be given perhaps a discounted weight but shall not be completely dismissed from the overall model.

Reviver does a great job when exact input is provided as the search query (i.e., no transformation is applied) by always retrieving the correct result at rank one. While the interface-based retrieval approach could achieve the same goal for only five tasks. In the case of Tasks 5 and 6 the correct result did not even make it to the top ten. We suspected that this might be the result of using the OR operator by default in the interface-based retrieval prototype. Therefore, we repeated the experiment for this prototype with the AND operator—instead of OR—connecting all the terms in the *Apache Solr* search. This strategy improved the ranking of the

Table 6.14: Comparing the performance of the four similarity heuristics. The entries in the columns mark the heuristics that ranked the correct result among top 10 retrieved results for each transformation combination. For example, a *R,I,D,K* entry denotes that the correct result was successfully retrieved by all four heuristics. In the *Tasks* columns, *R*, *I*, *D*, and *K* stand for reference, call-set, data flow, and lexical similarity heuristics respectively. In the *Transformations* column, *O*, *N*, *T*, *S*, and *P* stand for original, name, type, scenario, and protocol transformations respectively.

	Tasks									
Transformations	1	2	3	4	5	6	7	8	9	10
O	R,I,D,K	R,I,D,K	R,I,D,K	R,D,K	R,I,D,K	D,K	R,D,K	R,D,K	R,I,D,K	R,I,D,K
N	R,I,D,K	R,I,D,K	R,I,D	R	R		R,D	R,D	R,I,D	R,I,D,K
T	D,K	K	D,K	D,K	I,D,K	D,K	K	D,K	K	D,K
S	R,I,D,K	R,I,D,K	R,I,D,K	R,D,K	R,I,D,K	D,K	K	D,K	R,I,D,K	I,D,K
P	R,I,D,K	R,I,D,K	R,I,D,K	R,D,K	R,I,D,K	K	R,D,K	R,D,K	R,K	R,I,D,K
NT										
NS	R,I,D	R,I,D,K	R,I,D	R	R		R	R	I,D	I,D,K
NP	R,I,D	R,I,D,K	R,I,D	R			R,D,K	R,D	R	R,I,D,K
TS	D,K	K	D,K	K	I,D,K	D,K	K	D,K	K	I,D,K
TP	D,K	K	D,K	D,K	R,I,D,K	K	K	K	K	K
SP	R,K	R,I,D,K	R,I,D,K	R,D,K	R,I,D,K	K	K	D,K	R,K	I,K
NTS										
NTP										
NSP	R,I,D	R,I,D,K	R,K,D	R					R	R,I,K
TSP	D,K	K	I,K	D,K	R,I,D,K	K	K	K	K	K
NTSP										I,K

original test cases by placing seven correct recommendations at the top position—it still did not manage to place the correct result for two tasks among the top ten because the interfaces are composed of very common terms. On the downside, this strategy reduced retrievability for transformed test cases by retrieving a total of 27 correct results—instead of the original 71 correct results with the OR operator.

If the searcher's choice of names is the same as that of the existing functionality (i.e., in the case of T, S, P, TS, TP, SP, and TSP transformations) both approaches perform quite well—with *Reviver* being noticeably better. The *Reviver* approach only failed to retrieve 4 out of 80 results in this category while the interface-based retrieval failed in 15 cases. Exactness of names often provides enough lexical similarity for both approaches to correctly identify the target function. However, if the combination of terms in the input query is very common then lexical similarity might not suffice.

If the searcher uses the correct types but her choice of names is not the same as that of the existing functionality (i.e., in the case of N, NS, NP, and NSP transformations) both approaches struggle to retrieve relevant results—with *Reviver* again being noticeably better. *Reviver* successfully retrieved 16 out of 40 results in this category while the interface-based retrieval only did it for 5 cases. In other words, if a searcher is uncertain about naming of a function or its interface elements (e.g., methods) a keyword-based search would not be the best approach for finding it. This could happen every time a searcher is looking for a piece of code that is not described by a designated name or none that she is aware of. Type similarity still induces some level of lexical similarity that the interface-based retrieval approach can benefit from. The searcher might still have a chance to retrieve some relevant results if she can get the types in the interface of function right—that is its public attributes, method arguments, and return value types. Similarity of types in method invocations—despite dissimilarity of their names—still gives the data flow similarity heuristic the chance to find similar data flow paths, hence making *Reviver* more efficient.

The greatest challenge to both approaches is when the searcher's choice of both names and types is not the same as that of the existing functionality (i.e., in the case of NT, NTS, NTP, and NTSP transformations). Out of the 40 cases in this category neither of the approaches managed to place any results in the top ten. The combination of name and type dissimilarities leaves no lexical similarity between the input query and the target function. Hence, the interface-based retrieval would have no chance to retrieve the correct result. Likewise, both lexical and data flow similarity heuristics in *Reviver* are rendered ineffectual in the absence of lexical and type similarity.

At the task level, *Reviver* outperforms the interface-based retrieval approach by retrieving one to five more correct results except for Tasks 3 and 9; approaches tie in the number of correct results retrieved for Task 3 and the interface-based retrieval approach performs better in the case of Task 9 by retrieving two more correct results. The function under test in both tasks has a very simple design and is composed of one method—that limits the applicability of our data flow similarity heuristic. In other words, the data flow similarity heuristic is returning rather irrelevant results for these two tasks hence reducing the quality of the overall result set. However, the combination of the terms in the name of the class and its method (i.e., *convert*, *html*, and *text* in Task 3 and *util*, *word*, and *frequenc* in Task 9) are unique enough[4] in our repository to help identify the class once the names are not masked[5] through name

[4]A list of top 30 frequent terms in the *Apache Solr* index is provided in Appendix C.

[5]The correct results in both tasks are from cases that name transformation was not applied, i.e.,

transformation.

A good example of the term uniqueness phenomenon is observed in the case of Task 4. *Reviver* outperforms the interface-based retrieval approach by retrieving one more correct results but the interface-based retrieval approach does a better job by placing eight correct result at the top—compared to *Reviver* that only did it for 6 cases. Further investigation uncovered that the term *creditcard* appears in no other class than that of the function under test in Task 4. Hence, similar to Tasks 3 and 9, the interface-based retrieval approach manages to successfully identify the function under test for the 8 transformations that do not include name transformation due to the fact that the combination of terms in the test cases are quite unique.

On the contrary, when it comes to Task 5 the interface-based retrieval prototype cannot retrieve any of the transformations while *Reviver* came up with 8 out of the 16 possible transformation combinations. The set of terms that make up the interface of Task 5 (i.e., *string*, *bag*, *add*, *remov*, *occur*, and *size*) are very general and the combination of them appears in our repository quite frequently. As a result, none of the transformations of Task 5 made it to the top ten results retrieved by the interface-based retrieval prototype.

6.4 Limitations

The choices and assumptions made in the evaluation could have limited the validity of the experiment.

The repository used in the evaluation study is relatively small compared to the size of the open source code available online. However, it is well comparable to a

O, T, S, P, TS, TP, SP, and TSP transformations.

private software repository of a medium to large organization. In terms of repository content, a domain-specific repository like that of an organization would consist of more tightly-related projects exercising standardized vocabulary, APIs, and technologies. Our source code repository was populated with arbitrary projects taken from the *Subversion* servers of the *Sourceforge*[6] open source software directory. The uniqueness of terms related to some tasks might have given the lexical similarity heuristic an unfair advantage in identifying them where name transformation is not applied, hence elevating the relative importance of the lexical similarity heuristic. However, its impact on the comparison of the two approaches would be small because they both utilize lexical similarity.

Another factor that influences the evaluation results is the choice of tasks that were employed to generate input queries. Tasks were originally selected for the evaluation study in Chapter 3. As we argued previously, task ideas were taken from the discussions in the *Java* developer community websites, and from code example catalogues commonly used as a reference by *Java* developers. Developers evidently found these functions worthwhile to discuss and learn from, and not so easy to develop or to find. The tasks were selected to be complex enough so that a developer would rather prefer to reuse, if given a chance. However, for a quantitative evaluation—to be statistically representative of the kind of tasks developers perform during their daily activities—the size of the task pool needs to be much larger than ten.

The four types of transformation discussed in Section 6.1.2 can only generate a subset of the possible semantically equivalent variations of a given function. Future research has to look into extending and adding other potential transformation

[6]sourceforge.org

types. Besides, manual application of transformation rules to test cases makes generation of transformed variation a tedious task prone to human error. Automating the process would allow for application of additional, more complex rules resulting in the expansion of the pool of the possible variations. Selecting an optimal subset of heuristics and their respective weights in the aggregated similarity function requires cross-validation with a large subset of the data set. Unless the application of test case transformations is automated, such a validation will not be possible. Furthermore, we are aware of the fact that utilizing such a small data set increases the risk of overfitting. Therefore, the results hereby reported may not be generalizable to the entire repository—until a future experiment validates them.

The four transformations were manually applied in the following order: name, type, scenario, and protocol. For example, to derive the NTP variation, first the name transformation was applied to the original test case. Then, the type and protocol transformation were applied to the outcome of the previous transformation respectively. The order in which transformations are applied can have an impact on the final product. Due to manual application of transformation we did not consider all possible ordering of transformations. When automating the generation of transformations in the future, altering the order of transformations may yield more variations.

In our evaluation combinations of transformations are assumed to have equal impact. As a result, the likelihood of the occurrence of semantically equivalent variations of any given function is expected to be the same. Furthermore, to limit the number of resulting combinations, partial transformations are not considered. Consequently, all the rules in a transformation category are administered together (resulting in binary combinations). In reality, however, none of these assumptions may be true; certain variations may statistically occur more frequently than others. For example, semantically equivalent instances of a function that provide alternative protocols in their design might be far more frequent than instances that utilize alternative terminology—or vice versa. Empirical studies in the future have to investigate characteristics of semantically similar source code and verify these assumptions.

One may argue that human written test cases can be very different from manually or automatically generated test cases based on a ruleset. Searchers in the real world might write test cases that would have little resemblance to those of the relevant functions in the repository. However, the same argument can also be made against the interface-based retrieval technique. The contrasts between the terminology and signature design equally limits the performance of the interface-based retrieval: the effect of such human generated test cases on the performance of the approaches has to be further studied in a future controlled user experiment.

The quality of the interface-based similarity metric has an impact on the effectiveness of the resulting interface-based retrieval approach. A better crafted interface-based similarity implementation can improve the performance of the TDR approach utilizing it. Our interface-based retrieval prototype does not support signature-based searches—where the names are left blank but the types are filled in—that can be useful for some searches. However, no matter how perfect the interface-based similarity gets it would still fail once name and type transformations are applied. Our study has been successful in demonstrating this major limitation of the interface-based retrieval approach.

Our interface-based retrieval prototype is roughly equivalent to the retrieval component of the existing TDR prototypes; it does not come with the automated transformation and testing features offered by some of the existing TDR approaches. Although post-retrieval processes—like those offered by *S6*—help with selection of more relevant results, they do not impact what is or is not retrieved. The question we tried to answer by our evaluation is which of the two approaches is more effective in retrieving relevant results. The effectiveness of the post-retrieval processes is best evaluated by another controlled experiment or a user study that is beyond the scope of our work.

Chapter 7

Extending Test-Driven Reuse

Our proposed approach to representing software semantics requires a test suite in order to index the semantics of the functionality in the system under test. Unit testing is a practice on the rise and it would not pose a problem to a organization that already practices unit testing. However, for the time being, the number of open source projects that come with unit tests is limited. As a result, the repository built by processing test cases will only be a fraction of the publicly available source code. To overcome this limitation, we propose ideas for two alternative techniques for retrieving relevant source code using test cases without requiring the projects to have been delivered with a test suite. The ideas here are presented in a preliminary form and have not been empirically evaluated. Future research in the area of software search and retrieval has to further solidify these propositions before they can be put into practice. Our first proposed approach seeks to improve interface-based retrieval as it is the underlying technique in existing TDR approaches. Next, we describe a new approach for finding existing functionality based on structural and behavioural patterns identified in test cases and the function under test.

7.1 Extended Interface-based Retrieval

The interface-based retrieval approach is based on the assumption that the design of the interface of a function and its choice of vocabulary can be known—or at least partially guessed—by the searcher. Although generally not true, this approach can help with retrieving software under circumstances that the interface design and vocabulary is preset or can easily be guessed. However, we found that the interface-based retrieval technique employed by existing TDR approaches can be made more flexible by changing its underlying representation model. To take advantage of high-speed token matching capabilities of text search engines, *Code Conjurer* and *CodeGenie* represent operation signatures as text. We propose an interface-based matching technique based on hybrid text and relational representation methods. Our approach builds matching flexibility into interface-based retrieval by allowing use of wildcards in the search query.

When program interfaces are represented as text then type similarity is limited to type name similarity. For example, all numeric reference types in the `java.lang` package (e.g., `Integer`, `Byte`, `Short`, and `Double`) extend the abstract class `java.lang.Number`. As a result, the search query specifying a single method specialized operation like `String toString(Integer)` can technically be matched with a more generalized implementation that comes with the operation signature `String toString(Number)`. The `toString(Number)` operation can be passed an argument of any type extending `Number` including `java.lang.Integer`. However, as type hierarchies are not modelled in the text representation of program interfaces this operation cannot be performed. One may argue that this problem can be resolved by defining the terms *number* and *integer* as synonyms. Although it makes sense in this example, it would not work for any general type and sub-type relationship. For example, although java.util.Stack<E> extends java.util.Vector<E>, defining the terms *stack* and *vector* as synonyms would do far more harm than good to the term matching overall.

7.1.1 Wildcards

Let's imagine a searcher is looking for a simple *Logger* feature that she initially expresses as:

```
class Logger {
  void setDefaultLevel(Integer);
  Integer getDefaultLevel();
  void log(String);
  void log(Integer, String);
}
```

The logging level (e.g., *info*, *warning*, and *error*) are given as Integer numbers and log messages will be supplied as String objects. The searcher has composed the input query based on her idea of what a logger might look like but would consider other similar designs as long as the functionality she is looking for is provided. For example, an alternative design in which logging level is defined by the enumeration type Level consisting of values INFO, WARN, and ERROR. Hence, the suggested interface for such a logger would be:

```
class Logger {
  void setDefaultLevel(Level);
  Level getDefaultLevel();
  void log(String);
  void log(Level, String);
}
```

The current interface-based similarity techniques—for instance that one offered by the *Merobase* code search engine—are equipped with a procedure called query relaxation (*Hummel and Janjic*, 2013). Interface-based retrieval query relaxation ignores the name of an operation or the types in its signatures if the query does not match any or enough number of results. However, the current procedure is not flexible to distinguish between what is already known and preset and what can potentially be relaxed. We propose to use wildcards to create additional flexibility for matching type relationships inside and between operation signatures. Using wildcards in interface-based search queries is like using parameters when defining programming types, for instance *Java* parameterized types. Any type that can be relaxed in a search query can be replaced by a wildcard. Constraints can be specified to restrict the types that can be replaced for the wildcard. Using wildcards, the search query for the Logger can be expressed as:

```
class Logger {
  void setDefaultLevel(W_1);
  W_1 getDefaultLevel();
  void log(String);
  void log(W_1, String);
}
```

W_1 in the above code snippet is a wildcard that be replaced by the type Integer, a local type that is part of the same API, or simply any other type that even the searcher does not know of. The same idea can be applied to the log methods as well. In their current form they expect to receive an input argument of type String to write to the output. However, it could well be that an existing implementation of the Logger is so designed that it expect an Exception, Throwable, or a generic Object as the input argument. A second wildcard W_2 can be defined to replace the String argument in the two log methods. The resulting input query then looks like the following:

```
class Logger {
  void setDefaultLevel(W1);
  W1 getDefaultLevel();
  void log(W2);
  void log(W1, W2);
}
```

7.1.2 Constraints

A searcher may associate a set of constraints with a wildcard. Constraints are meant to provide the searcher with a means of restricting or specifying preference for the types that can be replaced for the wildcard. Wildcard constraint expressions are composed according to the following protocol:

- A wildcard W can only be replaced with a type T is denoted by the clause $W = T$. For example, $W =$ String states that all occurrences of W in the retrieved program interface have to be of type String.
- A wildcard W can only be replaced with a sub-type of T is denoted by the clause $W < T$. W can be replaced either by T or one of its sub-types is denoted by $W <= T$. For example, $W <$ java.lang.Number states that W can be replaced by sub-types of the abstract class Number like Integer or Double.
- A wildcard W can only be replaced with a super-type of T is denoted by the clause $W > T$. W can be replaced either by T or one of its super-types is denoted by $W >= T$. For example, $W >=$ java.net.ConnectException states that W can be replaced by ConnectException or one of its super-types like java.net. SocketException or java.io.IOException.
- The wildcard $Self$ can only be replaced with the type sought after. It can be used to demonstrate creational design patterns like Singleton (*Gamma et al.*,

1994) or for referencing inner-types defined inside the main type sought after. For example, a singleton builder for Logger can be stated as $Self$ getInstance().

- AND, OR, and NOT logical operators can be used to join wildcard clauses.

7.1.3 Representation

Our proposed interface-based retrieval approach relies on text and relational representations of program interfaces. A bag of words representation based on *Apache Solr* similar to the one we used for the interface-based retrieval prototype in Chapter 6. Each class in the repository is represented as a document. Each document is a bag of words consisting of the names in the public interface of the class, i.e., the name of the class, its public methods, and attributes. For example, in the case of the Logger discussed before the test representation would consist of the terms *log*, *set*, *default*, *level*, and *get*. The words are reduced to their stem and duplicates are removed in the pre-processing step as described in Section 4.4.1.

The relational model stores the types in the program interfaces and their relationships. Type hierarchies are explored for all types in the repository and the **extends** and **implements** relationships are stored in the relational model. Figure 7.1 shows the entity relationship diagram of the relational model. Alternatively, the relational model can be replaced by a graph model that facilitates querying paths between nodes. For instance, verifying **implements** and **extends** relationships between two types using the graph model would be much easier than a relational model.

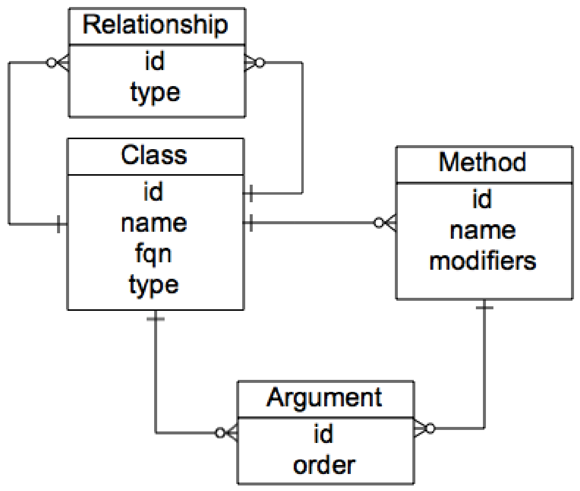


Figure 7.1: The entity relationship diagram of the relational representation of the extended interface-based retrieval approach.

7.1.4 Similarity

Matching an input query with wildcards and a set of constraints with interfaces of existing assets in the repository requires solving the assignment problem (*Burkard et al.*, 2009) over program interface elements.

Definition 7. *Let M_Q and M_C be the set of methods in the input query Q and a class C in the repository. The method interface similarity function $sim : M_Q \times M_C \rightarrow [0, 1]$ defines the degree of similarity between two method signatures based on their properties (i.e., name, arguments, return type, exceptions, and qualifiers).*

Definition 8. *Given the two sets M_Q and M_C and the method interface similarity function sim, the similarity score of the input query Q and a class C in the repository is computed by finding an injective function $f : M_Q \rightarrow M_C$ such that all the query constraints are met and the assignment score $\sum_{m \in M_Q} sim(m, f(m))$ is maximized.*

Unfortunately, solving such an assignment problem for every asset in a large repository is not computationally viable. Therefore, we might rely on a lexical pre-filtering stage to reduce the number of potential candidates. The test lexical representation is used for filtering candidates. Potential candidates are then delivered down the pipeline to an interface matcher that computes the assignment score between the query and each candidate. Finally, candidates are ranked in descending order of their scores. To take the idea to the next level, special wildcards can be allowed for certain words in the query. The searcher my further constrain the word wildcards by providing a small vocabulary.

7.2 Pattern-based Retrieval

When a developer reads through source code and documentation they form a mental model and make speculations about source code elements, their role, and potential relationships. In other words, the relationship between source code elements affects their semantics. Unit tests provide a living documentation of the system by highlighting appropriate and inappropriate use of a unit. Developers can gain a basic understanding of the system API and how to use it by looking at its unit tests (*Nasehi and Maurer*, 2010). A developer reads test code, in the same way they read any other code, and tries to understand its semantics by figuring out the relationship between its elements. We observed that certain structural and behavioural patterns get repeated in test code. Such patterns relate the elements of the test case and the function under test through pre and post conditions, constraints, and assumptions. For example, a method might be inferred to be a pre-condition of another method judging by the test scenarios that exercise that function. We propose to take advantage of structural and behavioural patterns observed in test cases to search and retrieve the software under test.

Identifying recurring patterns in API usage is the subject of specification mining research (*Zeller*, 2011). The functionality exercised in TDD tests might—at best—have been partially developed. This creates a major hurdle for application of existing specification mining techniques on TDD test cases. We thought of an alternative approach for identifying structural and behavioural patterns in the test cases based on the interface of the function under test. Using this approach we have collected a number of these structural and behavioural relationships in a pattern catalogue. Each pattern in our catalogue is a dependency between two or more methods in the function under test. In addition, we describe how each pattern is mined from existing source code. A pattern representation of source code can be used by a code search and retrieval approach—in conjunction with other representations—to potentially improve retrieval of relevant results.

Future software search and retrieval approaches can improve upon and utilize our pattern catalogue. In a pattern-based retrieval system, the search query test cases are scanned for any instances of the patterns in the catalogue. Identified patterns are then matched with existing assets in the repository that possess similar patterns. Although such a pattern-based approach cannot provide enough precision by itself, however it can be beneficial in conjunction with other heuristics in the test similarity model. Future empirical studies will determine the effectiveness of the proposed pattern-based retrieval approach.

7.2.1 Setter-Getter Pattern

The setter-getter pattern is a form of data flow dependency between two methods.

Identifying in a query test case An instance of setter-getter pattern exists in a test case if a literal or a reference is passed to the system under test through an invocation and later retrieved through another invocation. An assertion (e.g., *assertEquals* or assertSame) verifies that the retrieved value is equal to the value originally passed to the system under test. This indicates an implicit dependency between the two methods that is mediated through common underlying state. For example, in the following unit test excerpt m1 passes the same reference value to the system under test that is later retrieved and returned by m2.

```
SystemUnderTest sut = new SystemUnderTest();
int x = 10;
sut.m1(x);
assertEquals(x, sut.m2());
```

Identifying in an existing unit An instance of setter-getter pattern can potentially be present in a unit if a method m1 modifies the internal state (e.g., a data structure) based on an input parameter. The internal state is later queried and returned by a second method m2. For example, in the following code snippet, a call to m1 followed by a call to m2 will result in a data flow relationship from m1 to m2 that is mediated by the attribute f. Such a call sequence might not be an acceptable usage scenario for the unit and might be disallowed.

```
public class SystemUnderTest {
  private int f;
  public void m1(int v) {
    f = v;
  }
  public int m2() {
    return f;
  }
}
```

7.2.2 Collection Pattern

The collection pattern is a variation of the setter-getter pattern that requires special structural design in the system under test.

Identifying in a query test case An instance of collection pattern exists in a test case if multiple values (i.e., literals or references) are passed to the system under test through one method and later recovered through another method. Assertions (e.g., *assertEquals* or assertSame) are used to verify that the retrieved values are the same as the values originally passed to the system under test. This indicates an implicit data flow dependency between the two methods that is mediated through an underlying collection data structure. For example, in the following unit test excerpt m1 passes three different values to the system under test that are later retrieved and returned by m2.

```
SystemUnderTest sut = new SystemUnderTest();
int x = 1, y = 2, z = 3;
sut.m1(x);
sut.m1(y);
sut.m1(z);
assertEquals(z, sut.m2());
assertEquals(y, sut.m2());
assertEquals(z, sut.m2());
```

Alternatively, the getter method may retrieve and return the entire collection instead of individual items. Depending on the type of the collection returned the verification is going to be different. For example, the following test code snippet demonstrate the collection pattern when a Map collection is returned by the getter method.

```
SystemUnderTest sut = new SystemUnderTest();
```

```
int xValue = 10, yValue = 20, zValue = 30;
int xKey = sut.m1(xKey, xValue);
int yKey = sut.m1(yKey, yValue);
int zKey = sut.m1(zKey, zValue);
Map<Integer, Integer> map = sut.m2();
assertEquals(xValue, map.get(xKey));
assertEquals(yValue, map.get(yKey));
assertEquals(zValue, map.get(zKey));
```

Identifying in an existing unit An instance of collection pattern can potentially be present in a unit if a method m1 modifies the internal state (e.g., a collection data structure) based on an input parameter. The internal state (e.g., the collection object) is later queried and an individual item or a subset of the collection is returned by a second method m2. For example, in the following code snippet, a call to m1 followed by a call to m2 will result in a data flow relationship from m1 to m2 that is mediated by the collection f.

```
public class SystemUnderTest {
  private Collection<Item> f;
  public void m1(Item item) {
    f.add(item);
  }
  public Item m2(Integer p) {
    for (Item item : f)
      if (item.getSomeAttribute() == p)
        return item;
    return null;
  }
}
```

7.2.3 Inspector-Actor Pattern

The inspector-actor pattern is a form of control flow dependency between two methods.

Identifying in a query test case An instance of inspector-actor pattern exists in a test case if the return value of the call to m1 controls the condition of a control structure while m2 is invoked inside that control structure. m2 is therefore dependent upon the service provided by m1. The type of the control structure has an effect on the nature of the dependencies. For example, an **if/else** or **switch** control structure would make it a precondition dependency. In the following test excerpt the call to m1 is the precondition of the call to m2.

```
int v = 0;
if (m1()) {
  v=m2();
}
assertNotEquals(0, v);
```

Identifying in an existing unit An instance of the inspector-actor pattern can potentially be present in a unit if a method m1 returns a projection or attribute of a field f that a method m2 makes updates to. m1 can be used to verify a potential condition that is modified by m2. For example, in the following code snippet m1 returns a property of the collection f that is modified through m2 and has an effect on what is returned by m1.

```
public class SystemUnderTest {
  private Collection f;
  public int m1() {
    return f.size();
  }
  public void m2(Item item) {
    f.add(item);
  }
}
```

7.2.4 Precondition Pattern

The precondition pattern is a behavioural pattern observed in the sequence of invocations of two methods.

Identifying in a query test case If invocations of a method always precede the invocations of another method then they might have pre-condition dependency. A first method invocation m1 might initialize the internal state so that a call to m2 can successfully go through. However, coming across a couple of scenarios in which m1 appears before m2 cannot be a strong indication that a precondition relationship exists. For example, the searcher might have simply missed scenarios that the invocation order of m1 and m2 is interchanged. However, the process can still be semi-automated with the searcher being prompted to confirm existence of a precondition dependency at query time.

```
SystemUnderTest sut = new SystemUnderTest();
sut.m1();
Object v = sut.m2();
assertNotNull(v);
```

Identifying in an existing unit An instance of the precondition pattern can potentially be present in a unit if a method m1 initializes internal/external structures that the method m2 has to acquire in order to accomplish its task. Initialization can either be a one-time thing or it might have to be repeated each time m2 is called. If a one-time thing then the initialization usually takes place during construction time. However, the object might be lazily initialized once a call to m2 is made. For example, in the following code snippet m1 initializes the internal field f that is used by the method m2.

```
public class SystemUnderTest {
  private List<Integer> f;
  public SystemUnderTest() {}
  public void m1() {
    f = new ArrayList<Integer>();
  }
  public void m2(Integer i) {
    f.add(i);
  }
}
```

7.2.5 State Dependency Pattern

The state dependency pattern is a behavioural pattern that takes place between pairs of methods that are tested together.

Identifying in a query test case If the effect of a method m1 is tested by another method m2 then the two methods are in a state dependency. In other words, m1 affects the internal state of the system in such a way that is visible and verifiable through m2. Precondition pattern is a specialized form of the state dependency pattern. The effect of initialization performed by the first method is visible and can be tested by the second method. Hence, a state dependency exists between the two methods. Unlike the precondition pattern, in the general state dependency pattern the call m2 does not need to always preceded by a call to m1. In the following test case excerpt m2 is used for verifying the effect of a previous invocation of m1.

```
SystemUnderTest sut = new SystemUnderTest();
sut.m1();
assertTrue(sut.m2());
```

Identifying in an existing unit An instance of the state dependency pattern can be present in a unit if a method m1 modifies state that is used by a second method m2 for generating the return value. For example, in the following code snippet the call to m1 has an effect on the field f that can be verified by a call to m2.

```
public class SystemUnderTest {
  private List<Integer> f;
  public SystemUnderTest() {
    f = new ArrayList<Integer>();
  }
  public void m1(Integer i) {
    f.add(i);
  }
  public int m2() {
    return f.size();
  }
}
```

7.2.6 Expected Exception Pattern

The expected exception pattern is observed in unintended or exceptional usage scenarios of a unit.

Identifying in a query test case If a usage scenario is expected to cause a runtime or checked exception during testing then one or more of the methods invoked in the scenario should be able to throw that exception or one of its sub-types. The treatment of the exceptional behaviour can be indicative wether it is an intended or unintended usage scenario for the unit under test. For example, handling an exception takes place when it is intended. Failing the test through a fail call or re-throwing the exception is the sign of unintended behaviour. In the following test scenario either m2 or m3 might throw IOException or one of its sub-types. The failed outcome of the test case indicates that the exceptional behaviour is not intended.

```
SystemUnderTest sut = new SystemUnderTest();
sut.m1();
try {
  sut.m2();
  sut.m3();
} catch (IOException e) {
  fail();
}
```

Alternatively, in *JUnit4* the expected exception pattern might appear as an annotation on the test case. An exception expected this way is intended and does not mark inappropriate usage of the unit under test.

```
@Test(expected = IllegalStateException.class)
public void test() {
  SystemUnderTest sut = new SystemUnderTest();
  sut.m1();
  sut.m2();
}
```

Identifying in an existing unit Checked exceptions thrown by each method in a unit are easily identifiable by looking at the **throws** statement of each method definition. Other runtime exceptions can be traced by verifying the **throw** in the body of the method and other methods invoked by it. For example, in the following code snippet m1 throws the checked exception type IOException because it calls on m3. m2 can throw the runtime exception type IllegalStateException and m1 that invokes m2 is not handling this exception. Consequently m1 has the potential to throw IllegalStateException as well.

```
public class SystemUnderTest {
  public void m1() throws IOException {
        m2();
        m3();
    }
    private void m2() {
        if (!precondition)
            throw new IllegalStateException();
        }
    }
    public void m3() throws IOException {
        // something that can result in IOException
    }
}
```

Chapter 8

Discussion

The result of the evaluation study in Chapter 6 confirms our initial hypothesis that modelling functionality by indexing test cases that exercise them—in addition to their interfaces—provides a better means of capturing their semantics. We compared the performance of our *Reviver* prototype that we built based on our proposed idea with a prototype that we built to represent the interface-based retrieval approach. *Reviver* performed better in retrieving semantically relevant results by comparing query test case with test cases in the repository modelled by multiple representations. In this chapter, we provide an analysis of the contributions, implications, limitations, and applications of our work and a discussion of possible research directions that can follow it.

8.1 Analysis of Findings

Private organizational repositories: Not all source code available publicly online comes with a test suite. Therefore, building a catalogue of functions using test cases would only be limited to a fraction of open source code available online. However, the popularity of unit testing and test-driven development is increasing between industrial developers every year (*Hein*, 2012). For the time being, our proposed alternative to TDR perhaps better suites private repositories like that of an organization in which test-driven development is practiced. In such settings, the commonality among applications in terms of services, operating environments, technologies, and implementation techniques is high. Use of domain specific APIs increase similarity (*Kratz*, 2003) and as a result increases the opportunity to reuse functionality developed in the past. Our proposed TDR approach can take advantage of this opportunity and provides automated support for retrieving relevant source code from organizational repository when developing new functionality.

Pattern-based similarity: The evaluation study in Chapter 6 indicated that *Reviver*—similar to existing TDR approaches—is likely to fail retrieving relevant results if the query names and types do not match with that in the sought after function. The lexical similarity heuristic cannot be helpful in these situations for obvious reasons. The data flow similarity heuristic, on the other hand, requires similarity of method call nodes in the data flow graph in order to determine the similarity of data flow edges. The similarity of method call nodes is determined through lexical and type similarity which is why it also fails once lexical and type similarity is not present. To overcome this limitation of our approach we need test similarity heuristics that do not require lexical and type similarity.

The *Sourcerer* source code search engine creates fingerprints of code entities in the repository to support structural searches (*Linstead et al.*, 2009). Such structural fingerprints support retrieval of code with specific syntax irrespective of the its semantic aspects. One of the kinds of fingerprints supported by *Sourcerer* is micro patterns (*Gil and Maman*, 2005). Micro patterns are lower level structural design patterns that are defined based on the interface of a *Java* class. The patterns in our pattern catalog presented in Chapter 7 focus on behavioural attributes of methods.

Such a behavioural pattern catalog can be used to fingerprint source code in the repository to enable retrieval based on behavioural traits of source. An implementation of the pattern-based similarity heuristic can potentially improve the overall performance of *Reviver* by identifying similar assets in absence of lexical and type similarity.

Graph-based similarity: Our call dependence graph introduced in Chapter 4 is a model for describing the behaviour of a function in the context of a test case. Different behavioural models have been proposed in the literature for describing and matching software components. Petri nets, finite state machines, and work flow languages have all been used for modelling the behaviour of software components in the past. The problem with all these approaches is that the generation of such models cannot be mostly automated. Therefore, applying them to a repository the size of today's code search engines is practically out of the question. Our call dependence graph model built from component test cases—although not as precise as developer generated behavioural models—has the advantage of being automatically generated from test code.

Complex similarity model: Lexical similarity, structural fingerprinting, and data flow models have all been used for representing and matching software components in the literature. However, to the best of our knowledge, these techniques have never been applied to test cases before. We demonstrated how a similarity model can be built based on multiple interconnected sub-models each having their own representation. Using indexing techniques best suited for each representation, our similarity model is well-equipped to handle large-scale source code repositories. Our model is extensible and can include additional sub-models and representations that facilitate new similarity heuristics. The empirical evaluation study in Chapter 6 uncovered that the lexical and data flow similarity sub-models contribute the most overall in the similarity model.

Similarity heuristics: When building a model to allow similarity comparison of test cases (or code in general) one can assemble a group of rather disjoint naive heuristics or combine them together and form fewer more complex heuristics. Naive heuristics offer better scalability[1] while complex heuristics can potentially offer better precision. The reference and call-set similarity heuristics in our work—inspired by the *Strathcona* (*Holmes et al.*, 2006) example recommendation system—are rather naive heuristics that specialize on one aspect of test cases. During the evaluation study we observed that the naive heuristics returned more false-positives, resulting in degradation of the aggregated results. Therefore, we hypothesize that a more complex heuristic like data flow similarity—that combines lexical, reference, and invocation features—has the potential to be more effective.

Protocol similarity: Neither lexical nor type similarity is a prerequisite to semantic similarity. A similar function might have been designed using different vocabulary and types. A key question however is if the types used in a similar function are different then would such a function provide any utility for reuse? Obviously, if a searcher is flexible in their choice of types or the effort required for refactoring types in the function is justified then such a function can well be source for reuse. Otherwise, it might serve as an example if the searcher finds the function design or its algorithm

[1]We had to make our data flow similarity heuristic more naive in order to make it computationally efficient.

reusable.

Another key question regarding semantic similarity in test cases is whether it can be detected if lexical and type similarity is inexistent or insignificant. One way to approach this problem is to observe similarity in the behavioural patterns of the functions under test. In a controlled experiment, independent participants implemented a given functionality based on a written description. The chance of having similar protocol was observed to be much higher than having similar interface (*Kratz*, 2003). Analysis and extraction of object protocols is the objective of both typestate analysis (*DeLine and Fähndrich*, 2004) and specification mining (*Zeller*, 2011) research fields. However, the techniques devised in these fields require the source code or binary of the participating objects to be available for some variation of static or dynamic analysis. Therefore, these approaches cannot be applied to TDD test cases in which the code for the function under test is not yet available. Our pattern-based retrieval idea introduced in Chapter 7 is an attempt to find structure and behaviour similarity in software components based on the static analysis of their test cases.

Performance evaluation: A software reuse retrieval system—similar to any information retrieval system—has to be assessed on how well it meets the information needs of its users. Currently, there is no reference repository for evaluating and comparing the performance of reuse systems (*Hummel*, 2010). The traditional metrics like precision and recall [2] are based on the underlying assumption that the number of relevant items to the query in the repository is known. In absence of a reference repository with a set of queries to which the number of relevant items is known

[2]Precision is the fraction of retrieved items that are relevant to the query; recall is the fraction of items relevant to the query that are retrieved (*Manning et al.*, 2008).

many reuse systems rely on user studies in order to evaluate the utility of their systems. A user study better serves as a technique for evaluating the toolset associated with a retrieval system and should not be solely used as a means of evaluating its performance. In Chapter 6 we presented a technique based on code refactoring for evaluating the performance of *Reviver*. Although we generated approximations manually for a small task set but future research can look into automating generation of approximations. A fully automated and improved approximate generator can be used to measure approximate match retrieval performance of software reuse systems over large repositories.

Multiple representations: Our multi-representation reuse system is—to the best of our knowledge—the only one other than *Proteus* (*Frakes and Pole*, 1994). A comparative study on document collections in information retrieval discovered that although the differences in recall and precision between representation methods may not be significant, different methods tend to retrieve different documents (*Das-Gupta and Katzer*, 1983). Frakes later suggested that reuse systems should also seek to employ as many different representations as cost will permit (*Frakes and Gandel*, 1989). Findings of an empirical study with *Proteus* reuse system using four different representation methods were consistent with that of document collections (*Frakes and Pole*, 1994). We came up with the idea of using multiple representations of source code independently without prior knowledge of *Proteus* research. Our evaluation results in Chapter 6 provide further evidence that different similarity heuristics utilizing different representations of source code can find different items. The data flow and lexical similarity heuristics separately found 14 and 30 correct results respectively that the other heuristic did not find. Furthermore, the reference similarity heuristic produced 8 correct results that neither of the other two heuristics identified.

8.2 Future Work

Realizing the complete CBR cycle: Case-based reasoning as a cyclic problem solving process—as is test-driven reuse that is based on test-driven development. The classic case-based reasoning cycle is composed of four steps: *retrieve*, *reuse*, *revise*, and *retain* (*Richter and Weber*, 2013). Of these four steps, only the first three have been covered by the test-driven reuse approaches. It would be interesting to see how *retain*—the final step in CBR cycle—can also be integrated into the test-driven reuse process. In the *retain* step of CBR, the problem and the adapted solution are added to the case base. In the context of TDR, if the function a searcher is seeking is successfully located in the repository then the input test cases and the adapted function can both be added to the repository as a new variation. We hypothesize that the overall utility of the search service can be improved by allowing searchers to contribute new test cases and variations of existing functions.

Building a database of word relationships for software: When analyzing the semantics of a software the use of vocabulary should never be overlooked. After all, our most successful similarity heuristic proved to be the lexical similarity heuristic. However, this should not be interpreted as if the lexical attributes of source code is the most important of all; truth is that lexical similarity techniques and tools have been around much longer than other software similarity approaches—and hence are more mature. Researchers in recent years have tried to further adapt lexical similarity techniques and tools to the domain of software search and retrieval.

The main factor that limits the application of lexical similarity techniques is the vocabulary problem (*Furnas et al.*, 1987). A number of techniques have been devised in text analysis to get around the vocabulary problem; for example word synonyms can be used to expand the input query or similarity relationships can be established between words that appear together through latent semantic indexing (LSI). We could have had our lexical similarity heuristic utilize a database of word relationships like *WordNet* (*Fellbaum*, 1998) in conjunction to *Apache Solr*. However, application of *WordNet* to software has not been generally successful. Such a setup has been observed to lead matching the input query with many rather irrelevant artifacts (*Sridhara et al.*, 2008) simply because *WordNet* is populated with word relationships in general English. In order to improve the existing lexical similarity techniques for the software domain, future research has to come up a *WordNet*-like database of word relationships for software. In such a lexical database the words *write*, *persist*, and *dump* might be considered synonyms—a relationship that does not exist between their English counterparts in *WordNet*.

Measuring type similarity: Another limiting factor for our similarity heuristics is type-based similarity. For example, the reference similarity requires coexistence of references of the same types in two contexts to classify them as similar. This leads to what we call the type mismatch problem: different developers might use different types or components to implement the same function. It remain a topic for future research to determine the statistical significance of type mismatch in the software domain. We can think of two approaches to get around this problem and improve type-based similarity heuristics. A lexical and structural component similarity measure can be developed using the anti-unification theory similar to that used in *Jigsaw* (*Cottrell et al.*, 2008a). The degree of similarity of any pair of types in the repository can then be determined at index-time. Alternatively, following the latent semantic indexing technique for words, one can hypothesize that two software components that are used similarly (e.g., in similar contexts) might in fact be similar. A usage-based type similarity measure can therefore be defined based on the contexts in which two types appear as is done in (*Bajracharya et al.*, 2010b). Test similarity heuristics then can rely on either of these type similarity measures to provide cross-type matching.

Utilizing graph databases: Over the past decade various semantic and structural similarity approaches based on abstract syntax tree and graph model have appeared in the literature. Program dependence graph models have been used to identify semantically similar code (*Krinke*, 2001) and semantic code clones (*Gabel et al.*, 2008). Abstract syntax trees have been utilized for finding structural similarity in source code (*Sager et al.*, 2006). Call graphs have been utilized for finding frequent API call sequences (*Xie and Pei*, 2006) and sequences that start and end in two known types (*Mandelin et al.*, 2005; *Thummalapenta and Xie*, 2007). A graph model of the active development context has been used to offer code snippets to developers as examples (*Sahavechaphan and Claypool*, 2006). Behavioural patterns modelled as graphs have been used to retrieve and compose web services (*Grigori et al.*, 2008; *Corrales et al.*, 2008). However, the lack of technologies that facilitate mass storage and querying of graph structures has restricted past research to in-memory representations and matching of graph models. We believe exploring how graph-based models can be utilized in large-scale source code search and reuse would be an interesting future line of research. As a first step, we explored how a data flow heuristic—part of our test similarity model in Chapter 4—based on a graph model of dependencies between method invocations can find test cases with similar data flow patterns.

Improving data flow analysis: Our data flow similarity model is naive in its design and implementation. First of all, our static data flow analysis engine—as described in Chapter 4—is very simple and does only detect a subset of potential data flow dependencies. Furthermore, we did not model control flow dependencies. Our analysis of the control structures in our repository indicated that a large portion of test cases utilize a variety of control structures (see Appendix D for more details). Therefore, the impact of control structures on data flow dependencies in test cases cannot be dismissed. Future research has to look into ways of enhancing the data flow similarity model in test cases in order to improve the overall performance of the test similarity model.

Semi-Automated search queries: Automated generation of search queries from test cases in TDR makes it more convenient to the searcher. Unit tests are written in test-driven development regardless, therefore TDR does not generate an additional burden to developers. A plausible question that hereby arises is whether auto-generated queries provide enough expressive power for all type of searches. For example, when the searcher is not sure how to name or what types to use for certain elements of the function sought after writing a test case can be difficult. In these type of situations perhaps a mechanism for expressing the level of certainty can be provided to the searcher. Alternatively, a searcher can be allowed to specify the importance of individual elements in the test cases. For example, while certain behaviours or structures might be deemed essential to a function, others might only be preferred.

Ideally, a TDR prototype should allow the searcher to revise the auto-generated query and supply additional meta data.

Automated suggestions for query composition: As we discussed in Chapter 4 some software retrieval approaches—including existing TDR approaches—perform an initial filtering of search results based on the lexical and structural attributes of candidate assets to reduce turnaround time. As a result, if the searcher cannot come up with the correct choice for the interface terms and design then relevant results cannot be retrieved. If the searcher is unaware of the content of the repository then they have to resort to making arbitrary modifications to the search query. The *Design Prompter* (*Hummel et al.*, 2010) provides insight about repository content in the form of automated suggestions by computing the average interface of the assets that match an input query. These type of automated suggestions can help searchers pick terms, types, and signatures that would lead to matches. However, we found that more often the combination of the choices made by the searcher are so unique that cause the query not to match any assets. An alternative strategy that hence can be taken is to provide suggestions about choices that can be removed or altered to make the query match more results. In other words, the retrieval system should make automated query relaxation or modification suggestions in order to help adjust the query based on repository content.

Creating search engine friendly software projects: The variety and complexity of project configurations utilized by open source and proprietary projects makes them difficult to be setup automatically. Setting up a project and then building it requires developer knowledge that is now mostly described in documentation pages of a project. As a result, not all open source code retrieved on the Web can be processed and analyzed by automated tools. To make processing of projects more effective, the software community has to look into ways for supplying meta data that makes software projects more code search engine friendly—as web masters do in order to improve the visibility of their respective web sites. The *Apache Maven*[3] build automation tool can be used as a model for describing the project dependencies, including binaries, containers, and operating systems. Furthermore, the individual configurations required for building, running, and testing software projects can be outlined.

Measuring test case and implementation code similarity: We used the concept of test similarity as the basis of our approach for retrieving relevant source code in a test-driven reuse system. This approach can also extended to be applied to existing implementation code in the reuse library. When a new test case is written, the facts in the test case can be directly compared to those in existing source code in the library. If a close match is found, it is recommended as relevant source code. The Strathcona Example recommendation system (*Holmes et al.*, 2006) uses the same technique for retrieving examples exercising similar APIs from a reuse library. It would be interesting to see if a similarity measure between test and implementation code retrieve relevant source code from a reuse library that can be adapted and tested or looked upon as an example.

[3]http://maven.apache.org

Chapter 9

Conclusion

As test-driven development has gained in industrial popularity, the prospect of utilizing test cases as the basis for software reuse has become tantalizing: test cases can express a rich variety of structure and semantics that automated reuse tools can potentially utilize. However, the imprecise interface-based retrieval technique utilized by existing TDR approaches defeats the original purpose of utilizing the more precise instrument of test cases. A TDR candidate selection approach needs to be more flexible in recommending solutions, recognizing the inability of the developer to know exact vocabulary and that such vocabulary will often fail to suffice in locating a desired variation on common functionality.

The thesis of this dissertation is by modelling tests—in addition to function interfaces—the odds of finding semantically relevant source code is improved. We built a similarity model for comparing test cases along lexical, structural, and data flow dimensions. To provide the flexibility required by our complex similarity model, our similarity heuristics utilize multiple interconnected representations of test cases.

We built a proof of concept prototype, called *Reviver*, that indexes test cases and represents them using text search and relational database platforms[1]. We populated the repository of *Reviver* with a selection of *Java* open source projects with *JUnit*

[1]A graph database could have been an alternative platform for implementing the graph-based portion of our test similarity model.

tests. The result of a controlled experiment produce evidence that *Reviver* is more efficient identifying the function under test in the supplied test cases compared to an interface-based retrieval prototype that we built to represent existing TDR approaches. We provide further evidence that different similarity heuristics utilizing different representations of test cases can find different items.

In the end, detecting semantic similarity in source code in absence of lexical and type similarity still remains an open question. Today, despite being able to build large libraries of open source code we are constrained in what we can retrieve and reuse in such libraries due to our narrow definition of similarity. However, we believe that the idea of multi-representation reuse libraries is promising and has to be further pursued by the software search and retrieval community.

9.1 Contributions

This thesis has made five main contributions:

Evidence that current TDR approaches fall short of retrieving non-trivial or uncommon variants of functionality: We performed an experiment on the three existing tools for test-driven reuse, in which we found realistic, non-trivial tasks in developer forums, and for which a known solution existed in the tools repositories. We used existing test cases that exercised the known solution as the basis of the input to the tools. All the tools failed in most cases to locate relevant source code that would be simple to reuse, and often recommended irrelevant source code.

A similarity model for finding relevant source code in a TDR approach: Existing TDR approaches rely on interface-based retrieval for selecting the candidate code that goes through adaptation and testing. We propose the novel idea of indexing tests instead of the interface of the function under test. We designed a similarity model that uses heuristics that operate on the lexical, structural, and data flow features of test cases in order to find existing tests that exercise similar functions.

A technique for building a multi-representation reuse library from test code: There is empirical evidence that representing reusable artifacts through multiple representations improves the performance of reuse systems. Our test similarity model is implemented using three different representations. Our evaluation results provide further evidence that different similarity heuristics utilizing different representations of source code can find different items. Furthermore, we provide a federated search technique that aggregates the results retrieved by multiple heuristics.

The *Reviver* TDR prototype: We built a TDR prototype utilizing our similarity model using the open source stack. *Reviver* is designed as an IDE-based code search provider for finding existing functionality in a source code repository. Developers practicing test-driven development can use this toolset in their development process to look up similar functionality developed in the past for reuse or as reference examples.

A technique for evaluating performance of TDR approaches: We devised a technique for generating approximations of test cases by modifying their attributes. Transformations refactor code but do not modify the semantic of the function under test. We came up with a ruleset for transformations that generates semantically equivalent functions under test with alternative designs and testing scenarios. Using these transformations, we compared the performance of our similarity model with that of the interface-based retrieval in retrieving approximate matches.

[Bibliography]Bibliography

Klaus-Dieter Althoff, Andreas Birk, Christiane Gresse von Wangenheim, and Carsten Tautz. CBR for experimental software engineering. In Mario Lenz, Hans-Dieter Burkhard, Brigitte Bartsch-Spörl, and Stefan Wess, editors, *Case-Based Reasoning Technology: From Foundations to Applications*, volume 1400 of *Lect. Notes Comp. Sci.*, pages 235–254. 1998.

Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Searching api usage examples in code repositories with sourcerer api search. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, SUITE '10, pages 5–8, New York, NY, USA, 2010a. ACM.

Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, (0):–, 2012.

Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010b. ACM.

Sushil Krishna Bajracharya. *Facilitating Internet-Scale Code Retrieval.* PhD thesis, Donald Bren School of Information and Computer Sciences, University of California, Irvine, 2010.

Anton Barua, StephenW. Thomas, and AhmedE. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, pages 1–36, 2012.

Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.

Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

R. Bellman. *Adaptive Control Processes: A Guided Tour.* 'Rand Corporation. Research studies. Princeton University Press, 1961.

Avrim L. Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artif. Intell.*, 97(1-2):245–271, December 1997.

Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, September 2001.

Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proc. Europ. Softw. Eng. Conf./ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, pages 213–222, 2009.

Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

Jean-Marie Burkhardt and Françoise Détienne. An empirical study of software reuse by experts in object-oriented design. In *Proc. IFIP TC13 Int. Conf. Human–Comput. Interact.*, pages 133–138, 1995.

Silvio Cesare and Yang Xiang. Formal methods of program analysis. In *Software Similarity and Classification*, SpringerBriefs in Computer Science, pages 29–39. Springer London, 2012.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings* *of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

Juan Carlos Corrales, Daniela Grigori, Mokrane Bouzeghoub, and Javier Ernesto Burbano. Bematch: A platform for matchmaking service behavior models. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 695–699, New York, NY, USA, 2008. ACM.

Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, pages 214–225, 2008a.

Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Jigsaw: A tool for the small-scale reuse of source code. In *Companion Int. Conf. Softw. Eng.*, pages 933–934, 2008b.

Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.

Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial Java programs. In *Proc. ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, pages 313–328, 2008.

Padima Das-Gupta and Jeffrey Katzer. A study of the overlap among document representations. *SIGIR Forum*, 17(4):106–114, June 1983.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin Heidelberg, 2004.

D. L. Donoho. High-dimensional data analysis: the curses and blessings of dimensionality. In *American Mathematical Society Conf. Math Challenges of the 21st Century*. 2000.

Frederico A. Durão, Taciana A. Vanderlei, Eduardo S. Almeida, and Silvio R. de L. Meira. Applying a semantic layer in a source code search tool. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1151–1157, New York, NY, USA, 2008. ACM.

Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, February 2002.

Christiane Fellbaum. *WordNet: An Electronic Lexical Database.* Bradford Books, 1998.

Martin Fowler. Xunit, 2013. http://www.martinfowler.com/bliki/Xunit.html.

W. B. Frakes and P. B. Gandel. Representation methods for software reuse. In *Proceedings of the Conference on Tri-Ada '89: Ada Technology in Context: Application, Development, and Deployment*, TRI-Ada '89, pages 302–314, New York, NY, USA, 1989. ACM.

W.B. Frakes and T.P. Pole. An empirical study of representation methods for reusable software components. *Software Engineering, IEEE Transactions on*, 20(8):617–630, 1994.

William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Commun. ACM*, 38(6):75–88, 1995.

G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30:964–971, 1987.

Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

Rosalva E. Gallardo-Valencia and Susan Elliott Sim. Internet-scale code search. In *Proc. Wkshp. Search-Driven Devel. Users Infrastr. Tools Eval.*, pages 49–52, 2009.

Rosalva E. Gallardo-Valencia, Phitchayaphong Tantikul, and Susan Elliott Sim. Searching for reputable source code on the web. In *Proceedings of the 16th ACM international conference on Supporting group work*, GROUP '10, pages 183–186, New York, NY, USA, 2010. ACM.

RosalvaE. Gallardo-Valencia and Susan Elliott Sim. Software problems that motivate web searches. In Susan Elliott Sim and Rosalva E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 253–270. Springer New York, 2013.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 97–116, New York, NY, USA, 2005. ACM.

Daniela Grigori, Juan Carlos Corrales, and Mokrane Bouzeghoub. Behavioral matchmaking for service retrieval: Application to conversation protocols. *Information Systems*, 33(78):681–698, 2008. Advances in Data and Service Integration.

Florian S. Gysin. Improved social trustability of code search results. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pages 513–514, 2010.

M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

Seyyed Vahid Hashemian. *Automatic Signature Matching in Component Composition.* PhD thesis, School of Computer Science, University of Waterloo, 2008.

Avi Hein. Top 3 trends in developer testing, November 2012. http://esj.com/articles/2012/11/26/3-trends-developer-testing.aspx.

Reid Holmes. *Pragmatic Software Reuse.* PhD thesis, University of Calgary, November 2008.

Reid Holmes and Robert J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 2012. In press.

Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proc. Europ. Softw. Eng. Conf./ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, pages 237–240, 2005.

Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

Oliver Hummel. *Semantic Component Retrieval in Software Engineering.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Mannheim, 2008.

Oliver Hummel. Facilitating the comparison of software retrieval systems through a reference reuse collection. In *Proc. Wkshp. Search-Driven Devel. Users Infrastr. Tools Eval.*, pages 17–20, 2010.

Oliver Hummel and Colin Atkinson. Extreme harvesting: Test driven discovery and reuse of software components. In *Proc. IEEE Int. Conf. Info. Reuse Integr.*, pages 66–72, 2004.

Oliver Hummel and Colin Atkinson. The Managed Adapter Pattern: Facilitating glue code generation for component reuse. In *Proc. Int. Conf. Softw. Reuse*, pages 211–224, 2009.

Oliver Hummel and Colin Atkinson. Automated creation and assessment of component adapters with test cases. In Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 166–181. Springer Berlin Heidelberg, 2010.

Oliver Hummel and Werner Janjic. Test-driven reuse: Key to improving precision of search engines for software reuse. In Susan Elliott Sim and Rosalva E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 227–250. Springer New York, 2013.

Oliver Hummel, Werner Janjic, and Colin Atkinson. Evaluating the efficiency of retrieval methods for component repositories. In *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, pages 404–409, 2007.

Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52, 2008.

Oliver Hummel, Werner Janjic, and Colin Atkinson. Proposing software design recommendations based on component interface intersecting. In *Proc. Int. Wkshp. Recommend. Syst. Softw. Eng.*, pages 64–68, 2010.

Oliver Hummel, Colin Atkinson, and Marcus Schumacher. Artifact representation techniques for large-scale software search engines. In Susan Elliott Sim and Rosalva E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 81–101. Springer New York, 2013.

Tomoyuki Ishimaru and Shunsuki Uemura. An object-oriented data model for multiple representation of object semantics. *Systems and Computers in Japan*, 27(9): 23–32, 1996.

Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An unabridged source code dataset for research in software reuse. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 339–342, Piscataway, NJ, USA, 2013. IEEE Press.

Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay. Accurately interpreting clickthrough data as implicit feedback. In *Proc. ACM SIGIR Int. Conf. R&D Info. Retrieval*, pages 154–161, 2005.

Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Code similarities beyond copy & paste. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 78–87, Washington, DC, USA, 2010. IEEE Computer Society.

David Kawrykow and Martin P. Robillard. Improving api usage through automatic detection of redundant code. In *ASE*, pages 111–122, 2009.

Gabriella Kazai and Mounia Lalmas. eXtended Cumulated Gain measures for the evaluation of content-oriented XML retrieval. *ACM Trans. Inf. Syst.*, 24(4):503–542, 2006.

Benedikt Kratz. Empirical research on the relationship between functionality and interfaces of software components. Master's thesis, Tilburg University, 2003.

J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309, 2001.

Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

CharlesW. Krueger. Towards a taxonomy for software product lines. In FrankJ. Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 323–331. Springer Berlin Heidelberg, 2004.

Beth M. Lange and Thomas G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, pages 69–73, 1989.

Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53(4):294–306, 2011.

Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

Nuo Li, P. Francis, and B. Robinson. Static detection of redundant test cases: An initial study. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 303–304, 2008.

Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.

Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

D. Lucredio, A.F. Prado, and E.S. de Almeida. A survey on software components search and retrieval. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 152–159, 2004.

Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, August 1991.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proc. ACM SIGPLAN Conf. Progr. Lang. Des. Impl.*, pages 48–61, 2005.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie Williams. *Extreme Programming Perspectives.* Addison-Wesley, 2003.

Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick. Knowledge reuse for software reuse. *Web Intelli. and Agent Sys.*, 6(1):59–81, January 2008.

Mark M. McIntyre and Robert J. Walker. Assisting potentially-repetitive small-scale changes via semi-automated heuristic search. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 497–500, New York, NY, USA, 2007. ACM.

Seyed Mehdi Nasehi and Frank Maurer. Unit tests as api usage examples. In *ICSM*, pages 1–10, 2010.

Mehrdad Nurolahzade, Robert J. Walker, and Frank Maurer. An assessment of test-driven reuse: Promises and pitfalls. In John Favaro and Maurizio Morisio, editors, *Proceedings of 13th International Conference on Software Reuse*, volume 7925 of *ICSR 2013*, pages 65–80. Springer Berlin Heidelberg, 2013.

Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123, 1995.

Joel Ossher and Cristina Lopes. Applying program analysis to code retrieval. In Susan Elliott Sim and Rosalva E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 205–225. Springer New York, 2013.

Joel Ossher, Sushil Bajracharya, and Cristina Lopes. Automated dependency resolution for open source software. In *Proc. Int. Wkshp. Mining Softw. Repositories*, pages 130–140, 2010.

Claus Pahl. An ontology for software component matching. *International Journal on Software Tools for Technology Transfer*, 9(2):169–178, 2007.

D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 228–232, 2009.

Andy Podgurski and Lynn Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pages 349–361, 1992.

Martin F. Porter. Snowball: A language for stemming algorithms, October 2001. http://snowball.tartarus.org/texts/introduction.html.

R. Prieto-Diaz. Implementing faceted classification for software reuse. In *Software Engineering, 1990. Proceedings., 12th International Conference on*, pages 300–304, 1990.

Rubén Prieto-Díaz. Status report: Software reusability. *IEEE Softw.*, 10(3):61–66, 1993.

Steven P. Reiss. Semantics-based code search. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pages 243–253, 2009a.

Steven P. Reiss. Specifying what to search for. In *Proc. Wkshp. Search-Driven Devel. Users Infrastr. Tools Eval.*, pages 41–44, 2009b.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358366, 1953.

Michael M. Richter and Rosina O. Weber. *Case-Based Reasoning: A Textbook*. Springer, 2013.

Philippe Rigaux and Michel Scholl. Multiple representation modelling and querying. In *Proceedings of the International Workshop on Advanced Information Systems: Geographic Information Systems*, IGIS '94, pages 59–69, London, UK, UK, 1994. Springer-Verlag.

Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Softw.*, 27(4):80–86, 2010.

Mary Beth Rosson and John M. Carroll. Active programming strategies in reuse. In *Proc. Europ. Conf. Object-Oriented Progr.*, pages 4–20, 1993.

Jean-François Rouet, Catherine Deleuze-Dordron, and André Bisseret. Documentation as part of design: Exploratory field studies. In *Proc. IFIP TC13 Int. Conf. Human–Comput. Interact.*, pages 213–216, 1995.

Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *Proc. Int. Wkshp. Mining Software. Repos.*, pages 65–71, 2006.

Naiyana Sahavechaphan and Kajal T. Claypool. XSnippet: Mining for sample code. In *Proc. ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, pages 413–430, 2006.

Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often code is cloned across repositories. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1289–1292, Piscataway, NJ, USA, 2012. IEEE Press.

Martin Shepperd. Case-based reasoning and software engineering. In Aybüke Aurum, Ross Jeffery, Claes Wohlin, and Meliha Handzic, editors, *Managing Software Engineering Knowledge*, chapter 9. Springer, 2003.

Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1):4:1–4:25, December 2011.

Giriprasad Sridhara, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proc. IEEE Int. Conf. Program Comprehen.*, pages 123–132, 2008.

Peri Tarr and Harold Ossher. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 729–730, 2001.

StephenW. Thomas, Hadi Hemmati, AhmedE. Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, pages 1–31, 2012.

Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 204–213, 2007.

Medha Umarji, Susan Elliott Sim, and Crista Lopes. Archetypal internet-scale source code searching. In Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP The International Federation for Information Processing*, pages 257–263. Springer US, 2008.

Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, 2006.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proc. Int. Wkshp. Mining Softw. Repositories*, pages 54–57, 2006.

Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Au-*

tomated Software Engg., 12(2):199–235, April 2005.

Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

Yuhanis Yusof and Omer F. Rana. Combining structure and function-based descriptors for component retrieval in software digital libraries. *Integr. Comput.-Aided Eng.*, 15(4):279–296, December 2008.

Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, October 1997.

Andreas Zeller. Mining specifications: A roadmap. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 173–182. Springer Berlin Heidelberg, 2011.

Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. Prioritizing junit test cases in absence of coverage information. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 19–28, 2009.

Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pages 563–572, 2004.

Appendix A

Apache Solr Document Structure

We use *Apache Solr* to support lexical searches for the lexical and data flow heuristics in our *Reviver* prototype. Additionally, the interface-based retrieval prototype we built for the sake of evaluation study in Chapter 6 is also built on top of the *Apache Solr* text search engine. The *JUnit* test cases for Task 1 of our evaluation study is given in Figure A.1. The XML document in Listing A.1 demonstrates how the lexical elements in the test cases are represented as bag of words data structures. The lexical similarity heuristic described in Chapter 4 creates a similar document out of the searcher supplied use cases and finds existing documents in the *Apache Solr* index that best match it.

```
<doc>
  <str name="test_class_fqn">com.javaeedev.util.Base64UtilTest</str>
  <str name="test_class_name">Base64UtilTest</str>
  <arr name="method_name">
    <str>encodeString</str>
    <str>decodeString</str>
    <str>getBytes</str>
    <str>encode</str>
    <str>decode</str>
  </arr>
  <arr name="reference_name">
    <str>text</str>
    <str>base64</str>
    <str>restore</str>
    <str>data</str>
    <str>length</str>
    <str>i</str>
  </arr>
  <arr name="test_method_name">
    <str>testEncodeString</str>
    <str>testEncode</str>
    <str>testDecodeBadBase64</str>
  </arr>
  <arr name="method_return_type_fqn">
    <str>java.lang.String</str>
    <str>byte[]</str>
  </arr>
  <arr name="parameters">
    <str>java.lang.String</str>
    <str>byte[]</str>
  </arr>
  <arr name="reference_fqn">
    <str>java.lang.String</str>
    <str>byte[]</str>
    <str>int</str>
  </arr>
  <arr name="fqns">
    <str>com.javaeedev.util.Base64UtilTest</str>
    <str>java.lang.String</str>
    <str>byte[]</str>
    <str>int</str>
  </arr>
  <arr name="names">
    <str>Base64UtilTest</str>
    <str>testEncodeString</str>
    <str>testEncode</str>
    <str>testDecodeBadBase64</str>
    <str>text</str>
    <str>base64</str>
    <str>restore</str>
    <str>data</str>
    <str>length</str>
    <str>i</str>
    <str>encodeString</str>
    <str>decodeString</str>
    <str>getBytes</str>
    <str>encode</str>
    <str>decode</str>
  </arr>
</doc>
```

Listing A.1: A sample *Apache Solr* document illustrating how test classes are represented.

Our data flow similarity heuristic described in Chapter 4 uses bag of words data structures to represent the methods invoked in the test cases. The XML document in Listing A.2 demonstrates how the lexical elements in Task 1's Base64Util.encode() method are represented in *Apache Solr*. The data flow similarity heuristic creates a similar document out of each invocation in the searcher supplied use cases and finds existing documents in the *Apache Solr* index that best match it.

```
<doc>
  <str name="fqn">com.javaeedev.util.Base64Util</str>
  <str name="name">encode</str>
  <arr name="parameters">
    <str>byte[]</str>
  </arr>
  <str name="return_type_fqn">java.lang.String</str>
</doc>
```

Listing A.2: A sample *Apache Solr* document illustrating how methods are represented.

The interface-based retrieval prototype in Chapter 6—that we used as a baseline to compare the performance of *Reviver* against—uses bag of words data structures to represent the interface of the class under test in test cases. The XML document in Listing A.3 demonstrates how the lexical elements in the interface of Task 1's Base64Util are represented in *Apache Solr*. The prototype create a similar document out of the searcher supplied test cases and finds existing document in the *Apache Solr* index that best match it.

```
<doc>
  <str name="class_fqn">com.javaeedev.util.Base64Util</str>
  <arr name="method_argument_fqns">
    <str>java.lang.String</str>
    <str>byte[]</str>
  </arr>
  <arr name="method_names">
    <str>encodeString</str>
    <str>decodeString</str>
    <str>encode</str>
    <str>decode</str>
  </arr>
  <arr name="method_return_type_fqns">
    <str>java.lang.String</str>
    <str>byte[]</str>
  </arr>
</doc>
```

Listing A.3: A sample *Apache Solr* document illustrating how classes under test are represented.

```
package com.javaeedev.util;

import static org.junit.Assert.*;
import org.junit.Test;

public class Base64UtilTest {
    @Test
    public void testEncodeString() {
        final String text = "--- abcdefg \r\n hijklmn \t opqrst \u3000 uvwxyz ---";
        String base64 = Base64Util.encodeString(text);
        String restore = Base64Util.decodeString(base64);
        assertEquals(text, restore);
    }

    @Test
    public void testEncode() {
        final byte[] data = "abcdefg \r\n hijklmn \t opqrst \u3000 uvwxyz".getBytes();
        String base64 = Base64Util.encode(data);
        byte[] restore = Base64Util.decode(base64);
        assertEquals(data.length, restore.length);
        for(int i=0; i<data.length; i++) {
            assertEquals(data[i], restore[i]);
        }
    }

    @Test
    public void testDecodeBadBase64() {
        final String base64 = "ABCDEFG@@@\u3000\n\n@@..=";
        assertNull(Base64Util.decode(base64));
    }
}
```

Figure A.1: A sample test class retrieved from the source code repository.

Appendix B

Example of Transformations

An example of applying the four transformations to Task 6 of our evaluation study is provided here. The original test case for Task 6 is given in Figure 3.4. The result of applying name, type, scenario, and prototype transformations to the same test case can be seen in Figures B.1, B.2, B.3, and B.4 respectively.

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.io.IOException;
import org.xml.sax.SAXException;

public class CTest {
  String var1 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>text1</
      b><c>text2</c></a>";
  String var2 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy
      --><a><c>text2</c><b>text1</b></a>";

  @Test
  public void tm1() throws IOException, SAXException {
    C c = new C(var1, var2);
    assertTrue(c.m1());
    assertFalse(c.m2());
  }
}
```

Figure B.1: The name transformation applied to Task 6.

```java
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Date;

public class TestXmlDiff {
  StringBuffer xml1 = new StringBuffer("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>text1</b><c>text2</c></a>");
  StringBuffer xml2 = new StringBuffer("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy --><a><c>text2</c><b>text1</b></a>");

  @Test
  public void testDiff() throws Exception {
    Diff myDiff = new Diff(xml1, xml2);
    Date ref = new Date();
    assertEqual(1, myDiff.similar(ref));
    assertNotEqual(1, myDiff.identical(ref));
  }
}
```

Figure B.2: The type transformation applied to Task 6.

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import java.io.IOException;
import org.xml.sax.SAXException;

public class TestXmlDiff {
  String xml1;
  String xml2;
  String xml3;

  @Before
  public void setUp() {
    xml1 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>text1</b><c>text2</c><!-- comment --></a>";
    xml2 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy --><a><c>text2</c><b>text1</b></a>";
    xml3 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy --><a> <c>text2</c> <b>text1</b></a>";
  }

  @Test
  public void testDiff1() throws IOException, SAXException {
    Diff myDiff = new Diff(xml1, xml2);
    assertTrue(myDiff.similar());
    assertFalse(myDiff.identical());
  }

  @Test
  public void testDiff2() throws IOException, SAXException {
    Diff myDiff = new Diff(xml2, xml3);
    assertTrue(myDiff.similar());
    assertTrue(myDiff.identical());
  }
}
```

Figure B.3: The scenario transformation applied to Task 6.

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.io.IOException;
import org.xml.sax.SAXException;

public class TestXmlDiff {

  String xml1 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>text1</b
      ><c>text2</c></a>";
  String xml2 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy
      --><a><c>text2</c><b>text1</b></a>";

  @Test
  public void testDiff() throws IOException, SAXException {
    assertTrue(MyDiff.similar(xml1, xml2));
    assertFalse(MyDiff.identical(xml1, xml2));
  }
}
```

Figure B.4: The protocol transformation applied to Task 6.

Appendix C

Frequency of Terms

Empirical research has provided evidence that the tokens in programming languages—similar to words in natural languages—follow a *Zipf-Mandelbrot* law distribution (*Pierret and Poshyvanyk*, 2009). Table C.1 lists the top 30 terms collected from the class and method names of the 15,119 *Java* classes that appeared in test cases in our repository. Table C.2 lists the top 30 terms collected from the fully qualified package and type names in the same collection. For each term is frequency of classes that it appears in is indicated in the *Document Frequency* column.

Figures C.1 and C.2 demonstrate the long tail[1] distribution of terms in interface names and types of the classes under test in our repository, respectively.

[1]In long-tailed distributions a high-frequency population is followed by a low-frequency population which gradually tails off asymptotically. The events at the far end of the tail have a very low probability of occurrence.

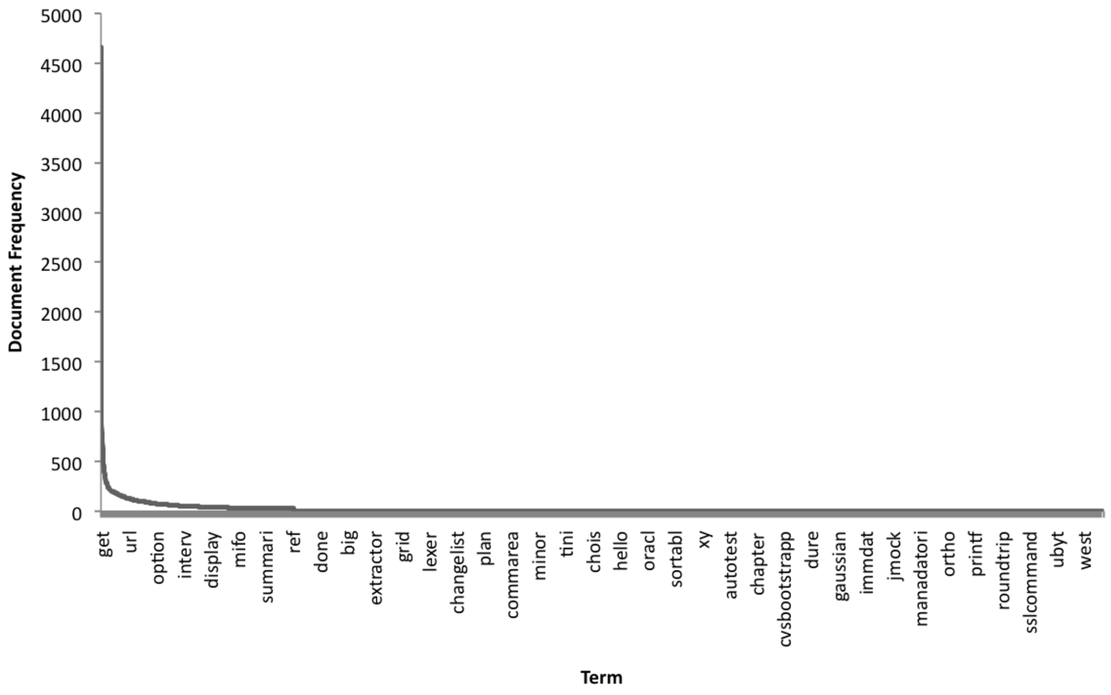


Figure C.1: The distribution of document frequency of terms collected from class interface element names. A term's document frequency is the number of documents (i.e., classes) in which a term appears. A sample population of terms (every 100th term sorted by descending order of document frequency) is indicated on the horizontal axis.

Table C.1: Top 30 terms collected from the interface element names of the classes under test. Terms are collected from the class or its method names. The *Term* and *Document Frequency* columns indicate the lower case stem of the word and the number of classes in which it appears, respectively. The total number of classes in the repository is 15,119.

Term	Document Frequency
get	4661
set	2151
name	997
add	867
to	775
is	754
string	719
valu	688
id	617
creat	505
size	472
type	445
test	417
list	409
mock	379
properti	362
file	347
valid	323
equal	286
data	282
class	276
user	275
date	272
messag	262
instanc	261
context	260
remov	236
array	231
object	223
all	222

Table C.2: Top 30 terms collected from the fully qualified name of interface element types in the classes under test. Terms are collected from the fully qualified name of the class, its method arguments, and method return types. The *Term* and *Document Frequency* columns indicate the lower case stem of the word and the number of classes in which it appears, respectively. The total number of classes in the repository is 15,119.

Term	Document Frequency
java	8947
unknownp	7415
lang	7315
org	7063
unknown	6949
void	5953
string	4499
util	3550
test	3403
class	2899
net	2565
int	2094
com	1724
mifo	1570
sf	1493
boolean	1398
list	1395
null	1077
object	759
io	750
core	740
mock	692
account	655
sourceforg	647
except	637
type	626
set	622
servic	613
common	602
map	600

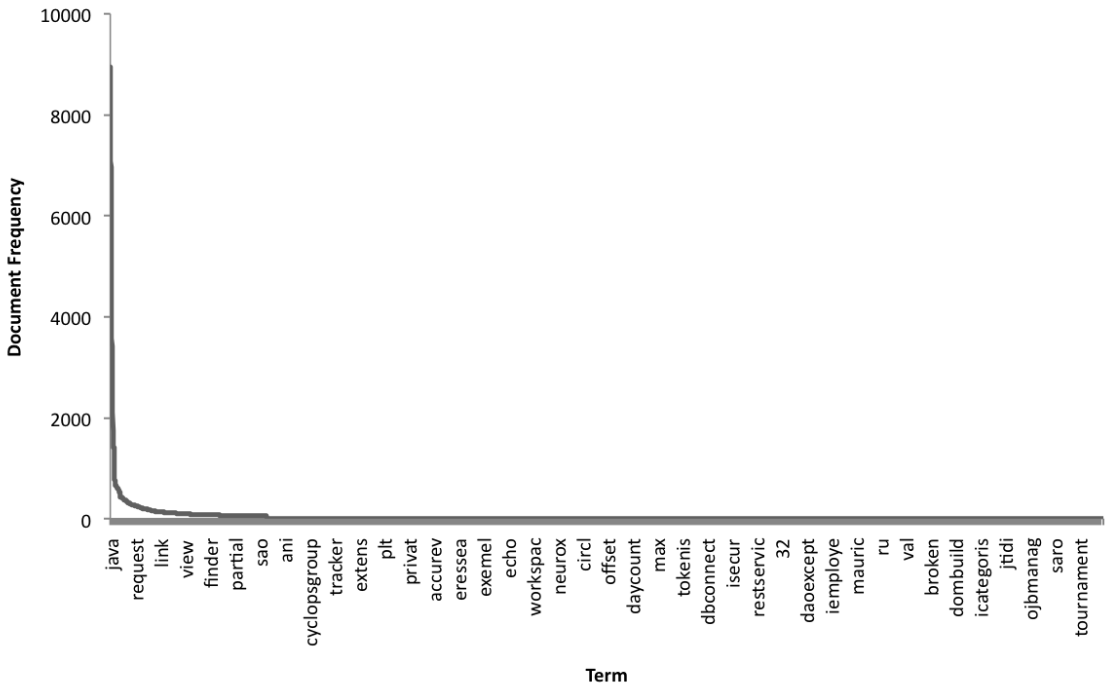


Figure C.2: The distribution of document frequency of terms collected from the fully qualified name of class interface element types. A term's document frequency is the number of documents (i.e., classes) in which a term appears. A sample population of terms (every 100th term sorted by descending order of document frequency) is indicated on the horizontal axis.

Appendix D

Control Structures in Tests

We used PMD[1]—the source code analyzer tool—to analyze the control structures in *JUnit* test code in our repository. Our simple data flow model in Chapter 4 does not take control structures into account while analyzing data dependency relationships in test code. Initially, we anticipated that the occurrence of control structures like **for**, **while**, **if**, and **switch** would be rather limited in test code. Therefore, modelling data flow relationships in tests can be rather simplified by ignoring control structures. Later, we decided to verify this assumption, hence we designed a PMD ruleset to identify control structures in *JUnit* test code.

Table D.1: The frequency of the four *Java* control structures **for**, **while**, **if**, and **switch** in *JUnit* test cases. The *Occurrence* column indicates the total number of occurrences of the control structure, while *Test Count* and *Percentage* columns indicate the number and percentage of *JUnit* test cases that happen to contain a control structure. The maximum, average, and standard deviation of the occurrences of control structures in *JUnit* tests are indicated in *Max*, *Average*, and *Stdev* columns respectively.

Control	Occurrence	Test Count	Percentage	Max	Average	Stdev
for	10276	3082	21.08%	132	3.34	5.12
while	2353	1107	7.57%	64	2.13	2.93
if	13285	3098	21.19%	199	4.30	8.37
switch	103	77	0.53%	4	1.34	0.66

[1] http://pmd.sourceforge.net

Table D.1 demonstrates the results of our analysis. As it turns out, the two control structures **for** and **if** occur in roughly 21% of *JUnit* test files in our repository. Therefore, control structures cannot be totally dismissed from a data flow model built for test cases. In light of the new knowledge, we decided to see if such a model should also provide support for nested control structures. We built a second PMD ruleset to analyze nested control structures in *JUnit* test code. As indicated in Table D.2 roughly 9% and 6% of test files in our repository contain nested control structures involving outer **for** and **if** controls.

Table D.2: The frequency of nested controls in the four *Java* control structures **for**, **while**, **if**, and **switch** in *JUnit* test cases. The *Occurrence* column indicates the total number cases in which a control structures nested inside the given control structure, while *Test Count* and *Percentage* columns indicate the number and percentage of *JUnit* test cases that happen to contain such a nested control structure. The maximum, average, and standard deviation of the occurrences of nested control structures in *JUnit* tests are indicated in *Max*, *Average*, and *Stdev* columns respectively.

Control	Occurrence	Test Count	Percentage	Max	Average	Stdev
for	2895	1299	8.89%	36	2.23	2.61
while	842	474	3.24%	19	1.78	1.76
if	2405	864	5.91%	73	2.79	4.34
switch	22	17	0.11%	3	1.29	0.57

While providing support for nested control structures would improve the precision of the model but at the same time it would slow down the processing of test code at index time and similarity matching of the input model at query time. Hence, the trade-off analysis whether to provide support for nested control structures in the data flow model remains to the future researchers.