A Test-Driven Approach to Code Search and its Application to the Reuse of Auxiliary Functionality

Otávio Augusto Lazzarini Lemos[a,1], Sushil Bajracharya[c], Joel Ossher[c], Paulo Cesar Masiero[b,2], Cristina Lopes[c]

[a]*Department of Science and Technology, Federal University of São Paulo, S. J. dos Campos, SP, Brazil*
[b]*Computer Systems Department, ICMC/USP, São Carlos Caixa Postal 668, 13560-970, São Carlos - SP - Brazil*
[c]*Donald Bren School of Information and Computer Sciences, University of California, Irvine*

Abstract

Context: Software developers spend considerable effort implementing auxiliary functionality used by the main features of a system (*e.g.* compressing/decompressing files, encryption/decription of data, scaling/rotating images). With the increasing amount of open source code available on the Internet, time and effort can be saved by reusing these utilities through informal practices of code search and reuse. However, when this type of reuse is performed in an ad hoc manner, it can be tedious and error-prone: code results have to be manually inspected and integrated into the workspace.
Objective: In this paper we introduce and evaluate the use of test cases as an interface for automating code search and reuse. We call our approach Test-Driven Code Search (TDCS). Test cases serve two purposes: (1) they define the behavior of the desired functionality to be searched; and (2) they test the matching results for suitability in the local context. We also describe CodeGenie, an Eclipse plugin we have developed that performs TDCS using a code search engine called Sourcerer.

Email addresses: otavio.lemos@unifesp.br (Otávio Augusto Lazzarini Lemos), sbajrach@ics.uci.edu (Sushil Bajracharya), jossher@ics.uci.edu (Joel Ossher), masiero@icmc.usp.br (Paulo Cesar Masiero), lopes@ics.uci.edu (Cristina Lopes)

[1]Financially supported by FAPESP, Brasil
[2]Financially supported by CNPq, Brasil

Method: Our evaluation consists of two studies: an applicability study with 34 different features that were searched using CodeGenie; and a performance study comparing CodeGenie, Google Code Search, and a manual approach.
Results: Both studies present evidence of the applicability and good performance of TDCS in the reuse of auxiliary functionality.
Conclusions: This paper presents an approach to source code search and its application to the reuse of auxiliary functionality. Our exploratory evaluation shows promising results, which motivates the use and further investigation of TDCS.

---

1. Introduction

With the popularity of the Open Source Software movement, there has been an increasing availability of source code over the Internet. This often makes software developers view the Internet as a *Scrapheap* for collecting raw materials to be used in production in the form of some reusable component, library, or simply examples revealing implementation details [1]. However, retrieving source code in the form of reusable self-contained pieces is usually hard and laborious, even with the help of keyword-based search engines: searches are usually text-based, dependencies have to be manually extracted, and pieces of code have to be manually copied and integrated into the workspace. To make pragmatic code search and reuse faster, safer, and more semantical, we introduce the use of test cases as an interface for automating this process.

Recently, there has been effort to develop search engines specifically targeted at source code [2, 3]. While these systems are promising, they do not leverage complex relations present in the code, and therefore have limited features and search effectiveness. In particular: (1) there is no strong support for integration of these search facilities in a development environment; (2) the mechanisms for expressing code queries are usually limited to keywords; and (3) there is little guarantee that the retrieved results correctly implement the behavior of the desired functionality in the local context.

Concerned with these limitations, and based on Sourcerer – a source code infrastructure [4, 8] developed within our group – we propose an approach to pragmatic source code search and reuse that integrates the use of test cases as inputs for the code search queries (following Holmes and Walker [20], we contrast pragmatic reuse with traditional, anticipated reuse approaches such as frameworks and product lines). Code queries are automatically generated

from information available on test cases (names of classes and methods, and interfaces), assuring quick retrieval of results that are most likely to be related to the desired functionality. Matching results are then executed against the test cases, providing knowledge of their conformance with the desired functionality. We call this approach Test Driven Code Search (TDCS).

Since TDCS should be tightly integrated with the development environment, we have developed a plugin for the Eclipse IDE named CodeGenie [27]. With CodeGenie developers design test cases, trigger the searching facility, and explore code results without resorting to any other tools. We believe TDCS can be applied in the reuse of several types of functionality but in this paper we evaluate its application to auxiliary functionality[3]. The evaluation of TDCS in the context of auxiliary functionality is important because developers perform this type of small-scale reuse regularly during the course of their development activities [13, 20]. Our evaluation consists in the exploration of CodeGenie in two ways: to check its applicability, we searched for several auxiliary features surveyed in our group as interesting to be reused, from a list used by Hoffmann et al. [17], and from examples used to describe a recent code search approach [40]; and to check its performance in the reuse of auxiliary functionality, we conducted an experiment with 34 undergraduate and 7 graduate students to compare CodeGenie with Google Code Search (CS), a well known code search engine, and with a manual approach (*i.e.*, implementing the feature by hand).

In the first study, we were able to find and reuse implementations of 34 features with CodeGenie. In the second study, CodeGenie was on average 50% faster than Google CS for the undergraduate students, and slightly faster for the graduate students (note that the sample size of graduate students was very small). In both experiments CodeGenie was significantly faster than the manual approach. The remainder of this paper is structured as follows. Section 2 presents background information about the main topics of this paper. Section 3 presents the TDCS approach along with requirements for a TDCS system implementation, and Section 4 describes a TDCS working example. Section 5 presents our TDCS implementation. Section 6 presents the results of the applicability and performance evaluation of TDCS through

[3]We define *auxiliary functionality* (or *auxiliary feature*) as a relatively small – comprising around 10–200 lines of code and involving around 1–3 classes –, characteristic, and supportive action of a system or component [24].

CodeGenie and Section 7 discusses related work. Finally, Section 8 concludes the paper.

2. Background

2.1. Source Code Search

Singer et al. [45] report code search as the most common activity for software engineers. Sim et al. [43], on the other hand, summarize a good list of motivations for code search where the use of several search forms – such as looking for implementations of functions and looking for all places that call a function – stand out. These studies provide strong evidence that source code search is an essential activity to software developers today.

Open source code repositories such as Sourceforge and Tigris provide simple searching capabilities, which are essentially keyword-based searches over the projects' meta-data. Because of that limited querying capability, when looking for source code on the Internet, developers usually resort to powerful general-purpose search engines, such as Google. Web search engines perform well for keyword-based search of unstructured information, but they are unaware of the specificities of software, so the relevant results are usually hard to find.

Currently there has been significant work in the development of large scale source code-specific search engines [2, 3]. Sourcerer is an infrastructure for code analysis and indexing that addresses some of the shortcomings of these systems; in particular by storing detailed information about the structural relations present in the code [4, 8].

2.2. Program Slicing

Program slicing is traditionally a technique for decomposing a program into the minimal set of statements that can affect (or are affected by) the slicing criterion [49]. The slicing criterion is a pair consisting of a program point and a subset of program variables. The resulting program must itself compile and should execute as the initial program with respect to the criterion. A concise definition of a program slice S is "a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P" [47].

Once the desired code is found in a code search task, it can often be difficult to manually extract what is necessary to make it work in a local context. In order to address this issue, we can use a variant on program

slicing. Instead of working at statement-level granularity, we can apply the same principles to entities such as classes and methods. Starting from an arbitrary set of seed entities E, all originating from the same program P, we automatically retain only those entities from P that are necessary for the entities in E to function properly. This approach is very similar to what is done by Tip et al.'s *Jax* [48], a tool for reducing the size of Java binaries through the elimination of unnecessary class files.

To give an example of how this works, suppose we are looking for a method that computes the probability mass function (pmf), that is, the probability that a discrete random variable is exactly equal to some value. In our search, we could come upon the *Math* class partially shown in Figure 1. The class contains a method to compute pmf – *public static double pmf(int k, int n, double p)* – and other math methods. Since we are only interested in pmf, we want to *extract* only those parts of the *Math* class that are related to it. By examining *Math*, we can see that *pmf* calls *exponentiation* and *combination* (which in turn calls *factorial*). Thus, to extract the *pmf* method out of *Math*, we also need these methods, everything else can be discarded. This is a type of *forward static slicing*, as we do not make assumptions regarding the program's input, and the dependency graph is traversed *forwards* from the declaration of the *pmf* method. Note that the slice may also span other classes and packages. This is the type of slicing approach we need when extracting code from code bases in a search task (see Section 5 for more details).

2.3. Test-Driven Development (TDD)

A *test case* is a set of inputs, execution conditions, and expected output for a specific function of a program [24]. The expected output is evaluated based on an *oracle* – in our case, the tester – which determines the correct result of the function given an input [12]. Test cases provide a context in which low-level design decisions can be made before having the actual implementation of a function. For instance, test cases can specify which classes and methods to create, how they will be named, what interfaces they will possess (by analyzing the input and output types), and how they will be used [14]. Test-Driven Development (TDD) takes advantage of that fact by guiding programmers to write functional test cases before production code [11].

The dynamics of writing the tests first – referred to as *Test-First* – is our focus in this paper. Test-First is usually supported by unit testing tools such as JUnit, a testing framework designed to support unit testing for Java

```
public class Math {
  public static final double PI = 3.141592653589793;

  public static double pmf(int k, int n, double p) {
   long firstTerm = combination(n, k);
   double secondTerm = exponentiation(p, k);
   double thirdTermBase = (1 - p);
   int thirdTermExp = n - k;
   double thirdTerm = exponentiation(thirdTermBase, thirdTermExp);
   return firstTerm * secondTerm * thirdTerm;
  }

  public static long combination(int k, int n) {
    ...
    long s = factorial(k);
    ...
    return s;
  }

  public static int abs(int i) { ... }

  public static long factorial(int x) {
    ...
    return f;
  }

  public static int min(int a, int b) { ... }

  public static double exponentiation(double base, int exponent) {
    ...
    return exp;
  }

  ...
}
```

Figure 1: A partial *Math* class being sliced.

programs. In JUnit, expected outputs are evaluated using special assertion methods – or *comparators* [12] (such as *assertEquals* and *assertTrue*).

The following is an example of a Test-First scenario using Java as the programming language and JUnit as the unit testing framework. Suppose a developer needs to implement a function that converts Arabic numbers to Roman numerals. An example of a test suite with test cases in the ordered pair form – <'input', 'expected output'> – for this function is: $T = \{< 1, \text{“}I\text{”} >, < 2, \text{“}II\text{”} >, < 4, \text{“}IV\text{”} >, < 10, \text{“}X\text{”} >, < 50, \text{“}L\text{”} >, < 1000, \text{“}M\text{”} >\}$. A partial implementation of T using JUnit is presented in Figure 2.

```
public class RomanTest extends TestCase {
  public void testRoman1() {
    assertEquals("I", Util.roman(1));
  }

  public void testRoman2() {
    assertEquals("II", Util.roman(2);
  }

  ...

  public void testRoman6() {
    assertEquals("M", Util.roman(1000));
  }
}
```

Figure 2: Partial JUnit test class.

Following the Test-First idea, to compile the test class, the developer should create a class named *Util* with a *roman* static method that implements the conversion of Arabic numbers to Roman numerals. If the test cases run successfully, there is evidence that the method is correctly implemented. Note that the test cases must be issued to a particular entry point in the program, which will also provide the output information – *i.e.*, the *actual* result [12] – given a specific input. In this case the entry point is the public method *roman*.

It is important to note that in TDD test cases are usually low level and no type of formal testing criteria (*e.g.*, functional or structural testing [24]) is used [14].

3. Test-driven code search (TDCS)

The same way that test cases can be used to define a software feature in TDD, they can also be used to describe a desired feature in a code search task. Moreover, in this context, we can take advantage of the following characteristics of TDD [14]:

1. Feedback: Test cases provide instant feedback about the suitability of a particular code result in the local context;
2. Task-orientation: The requirement of designing test cases first guides the developer in searching for self-contained and manageable software pieces, one at a time;
3. Quality assurance: Since code results might come from unknown sources which are not always trustable, tests cases help in assuring a certain

degree of quality. Up-to-date test cases also helps keeping track of the quality of the retrieved software pieces along the evolution of the system.

These observations indicate that test cases can be useful interfaces to code search. With respect to the search itself, test cases provide important information for querying the desired feature, such as the signature of the entry point operation. For example, consider the test cases for the Arabic to Roman function presented before. The signature of the entry point operation is *String roman(int)*. From the test class we can also extract the name of the class that contains the entry point: *Util*. In TDCS this type of information can be used to search for a particular piece of code. For instance, the terms 'roman' and 'util' can be used as keywords, and the signature can be used to match specific methods (entry points). The keywords help filtering out the solution set to candidates that are more likely to implement the desired functionality. The different types of information can furthermore be enabled, disabled, or relaxed to produce more or less restrictive queries. For instance, we might want to exclude the name of the class in the search, since it is harder to find an entry point that matches both keywords than one that matches only 'roman'.

Figure 3 shows a basic TDCS process. To describe a missing feature in the project, test cases are designed in the Integrated Development Environment (IDE). The search facility can then be triggered and, based on the information available on the test cases, a query is sent to a code search service capable of processing it. In the IDE, the developer can explore the results by integrating/testing and detaching them. To do that, a program slicing service to provide self-contained code pieces and a repository access service must be available at the Code Services side. Whenever the developer feels satisfied with a particular code result, it can be left integrated to the project. Detaching of a code result at any time can also be done.

There are two sides involved in TDCS: the IDE and the code services infrastructure. Next we explain the requirements for a TDCS system in both sides.

3.1. IDE support for TDCS

In the IDE: (1) test cases are designed, (2) test case-based search is performed, (3) integration/detachment of code results is performed, and (4) test case execution/analysis is performed.

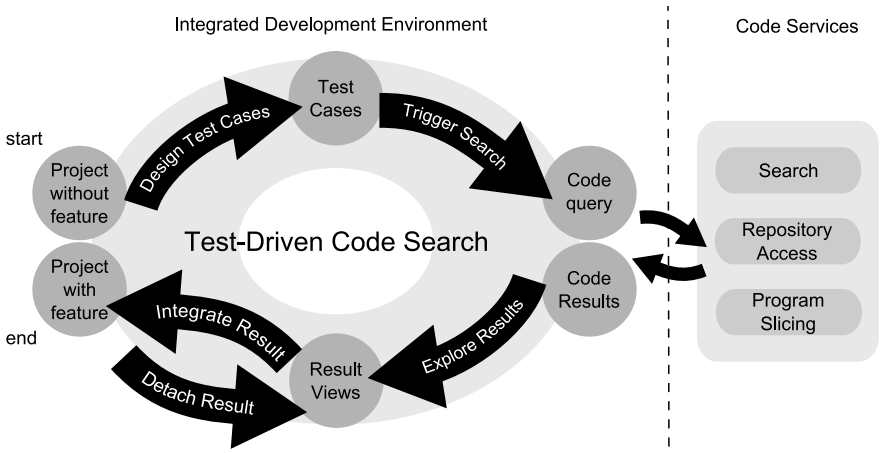


Figure 3: TDCS process.

To write test cases in a systematic way, some test development framework (such as JUnit for Java) must be supported by the IDE. To formulate test case-based code queries, the IDE must support the extraction of information contained in the test case modules. For instance, considering the example of a JUnit test class presented before, we must be able to extract the signature of the entry point method and the class name from the test class.

To integrate/detach code results, the IDE must support manipulation of code structures (*i.e.*, fields, methods, and classes) inside a project (such as merging/copying code structures). To maintain the traceability of the integrated code results, we must be able to identify each structure introduced into the application from a search task. In this way the developer can always tell whether a piece of the application comes from a search task or was implemented by someone in the team. Moreover, this structure tracking is also essential to make it possible the detachment of code results.

Finally, to perform test case execution and analysis, the IDE must support some type of testing tool.

3.2. Code services support for TDCS

At the Code Services side: (1) code searches are performed, (2) code slicing tasks are performed, and (3) repository accesses are performed. The first feature is the search itself, which is based on the test case-queries mentioned before. The search engine must be capable of processing queries with information about the code structure.

The second feature consists in being able to slice the source code to get smaller self-contained pieces of code for a particular feature. For the remainder of this paper, the term *slice* will be used to refer to the result of this type of slice. For instance, given a method m in a program, the slice of m is a program containing m along with its dependencies such that the new program will compile, and m will execute as it did for the original program.

Finally, the third feature consists in making available the access to the code base. We need to access the code base to show snippets of the source code in the IDE, for instance.

4. Working Example

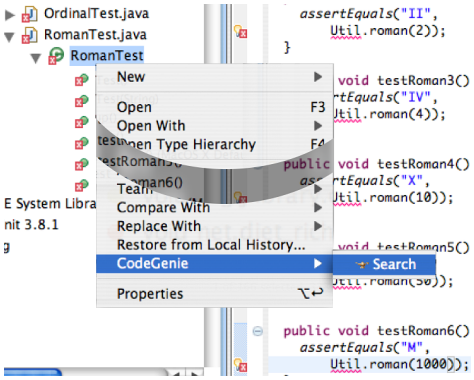
We implemented a system with the requirements described above. In this section we present a scenario example where two features – the Arabic to Roman function described in Section 2 and an Integer to Ordinal conversion function – are reused through TDCS, one of them using two implementation options. We illustrate our approach by using CodeGenie, the Eclipse plugin we implemented as the IDE part of TDCS (see Section 5).

4.1. Basic search

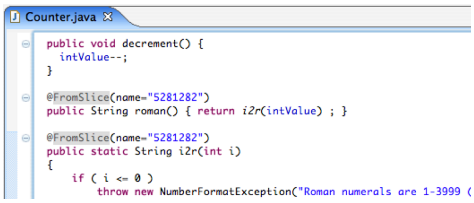
Consider the development of a document editing system. An important functionality of such systems are counters for sections, enumerations, pages, etc. The Arabic to Roman function discussed in Section 2.3 could be implemented in such system to present counters as Roman numerals. There are many ways of implementing this function, two examples would be: (1) a static *roman* method in a *Util* class that receives an integer and returns the corresponding Roman numeral in a string (such as the one defined by the test cases in Figure 2); and (2) an instance *roman* method inside a *Counter* class that returns the current counter value as a Roman numeral. Here we follow the first option and later we explain how the second option could also be used. The test cases for the static method implementation option were partially presented in Figure 2.

After designing some *ad hoc* JUnit test cases, the user can trigger the CodeGenie search facility by right-clicking on the test case class and selecting the CodeGenie Search menu option. Figure 4(a) shows the CodeGenie Search menu option being triggered for the Arabic to Roman JUnit test class. CodeGenie sends the query to Sourcerer which, in turn, returns code results. The keywords are initially formed by the terms coming from the

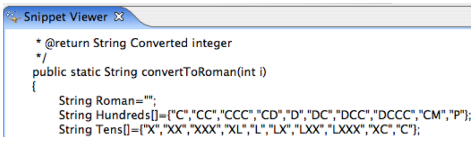
method name and the class name. In the example, 'util' and 'roman' are the initial keywords. By default every information on the test cases is used to generate the query: class name, method name, and method signature. After activating the search, the developer has the option to relax the query by enabling/disabling the return type, parameter types, and name terms as keywords.



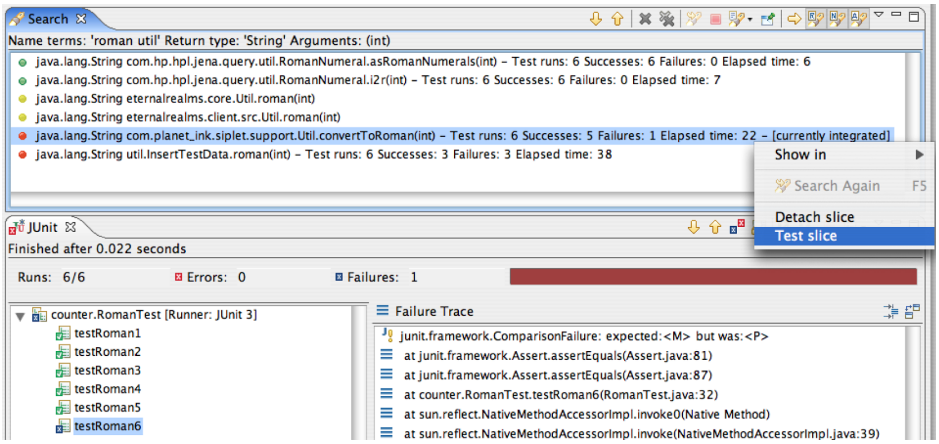
(a) Search triggering menu.



(b) Integrated result.



(c) Snippet Viewer.



(d) Search View.

Figure 4: CodeGenie screenshots.

The CodeGenie Search View with the returned results for the referred example is presented in Figure 4(d). From here, the developer can examine, integrate, test, and detach results by right-clicking on them and selecting the desired option. Green, red and yellow bullets are used to represent successful, failing and yet to be tested results. These are also used to order results accordingly. There is also a second-level ordering for successful results according to the test execution time. When a search result is selected, its code is presented in the Snippet Viewer (Figure 4(c)).

The Search View presented in Figure 4(d) refers to the state after integrating and testing four results for the example in question (two other results are shown as yet to be tested for the sake of completion). From these results, two are successful (green bullets grouped at the top) and two fail (red bullets grouped at the bottom). The important role of test cases to assure the quality of the retrieved code can be seen in this example. The two failing results are faulty implementations of the desired feature: one of them only converts numbers in the range 1–5, and the other converts the number 1000 to 'P' instead of 'M'. Figure 4(d) also shows the execution of the test set in the JUnit plugin for the mentioned faulty result.

At this point the developer has the following options: (1) choosing one of the green results (possibly after trying the other two yet to be tested candidates); (2) relaxing the query to get more results; or (3) enhancing the test set to further filter successful results. In the example, following (3), the developer could create more test cases to test other types of inputs to the Arabic to Roman function, for instance. In that case, test cases for the numbers 0 and -20, that do not have equivalent Roman numerals, could be created. For those numbers, the expected output could be to throw an exception. In this example (note that, from the yet to be tested candidates shown in Figure 4(d), one is successful and the other fails), after implementing these additional cases, the number of green results are reduced to two, making it easier for the developer to choose among them.

4.2. Query relaxation

Suppose the developer wants to add the feature of returning ordinal numbers given an integer in the same application. This function could be used to show the counter as an ordinal number. Now following TDCS, the developer should create test cases for such function. The test set could be the following: $\{< 1, \text{“}1st\text{”} >, < 2, \text{“}2nd\text{”} >, < 3, \text{“}3rd\text{”} >, < 11, \text{“}11th\text{”} >, < 21, \text{“}21st\text{”} >\}$. When we trigger the search, CodeGenie returns four results. However, none

of them pass the test cases because three are only returning the ordinal suffix without the number in front ('st', 'nd', 'rd') and one is faulty against the fourth test case (it returns '11st' when it should be '11th'). Following the query relaxation option, the developer might try to disable the use of the name of the class ('util'), since it might be filtering working results. When the search is triggered again, ten results come up and one of them is successful. Note that instead of relaxing the query to get more results, the developer could instead refactor the test cases to accommodate the majority of the results encountered, since they had a slightly different implementation of the function.

4.3. Instance methods

To explain how methods that manipulate fields can also be searched, next we show how the second implementation option of the Arabic to Roman function could be used. This example shows a feature being added to an existing class in the developer's project. Consider the implementation of the *Counter* class presented in Figure 5. To search for an Arabic to Roman instance method for this class, test cases have to be designed differently because the *roman* method does not contain any parameter, it manipulates the integer field of the *Counter* class. Figure 6 shows some test cases designed for this implementation option. Note that the assertion is made against a 'roman' instance method that receives no parameters and returns the roman numeral corresponding to the current counter value.

To search for this instance method with the current implementation of CodeGenie, the user can right-click on the test class to trigger the search. However, before integrating a result, the field in the local class must be renamed according to the name of the field in the result, as shown in the Snippet Viewer. In this way, when CodeGenie integrates the result, the classes are merged so that the field in the local class corresponds to the field in the reused code. For instance, in Figure 5 the integer field was renamed to *intValue* to match the name of the field in a code result. After integrating the results, all other operations are performed the same way. Figure 4(b) shows the *Counter* class with an integrated Arabic to Roman instance method in the developer's project. The *@FromSlice* annotation is used to keep track of the integrated code (see Section 5).

```
public class Counter {
  private int intValue;

  public Counter() {
    intValue = 0;
  }

  public Counter(int num) {
      intValue = num;
  }

  public int getValue() {
      return intValue;
  }

  public void setValue(int value)  {
      intValue = value;
  }

  public void increment() {
    intValue++;
  }

  public void decrement() {
    intValue--;
  }
}
```

Figure 5: *Counter* class.

```
public class RomanTesting
  extends TestCase {

  Counter n = new Counter(1);

  public void testRoman1() {
    assertEquals("I", n.roman());
  }

  ...

  public void testRoman2() {
    n.setValue(4);
    assertEquals("IV", n.roman());
  }
...
```

Figure 6: Some of the test cases for the instance method implementation of the Arabic to Roman function.

5. System Implementation

Our TDCS implementation comprises a plugin for the Eclipse IDE called CodeGenie and an infrastructure called Sourcerer. CodeGenie provides tight

integration of the automated search facility with a developer's environment while leveraging all code services that Sourcerer provides[4].

5.1. IDE Integration: CodeGenie [27, 28]

The IDE side of our TDCS implementation uses Eclipse, an extensible platform for tool integration that provides several Java software development services. Eclipse is suitable for the purposes at hand because it fulfills many of the TDCS requirements discussed in Section 3.1. Moreover, the extensible nature of Eclipse makes it easier to integrate the particular TDCS features.

The test case design part of CodeGenie is supported by JUnit, which is fully integrated with Eclipse. JUnit test classes must be created to define the desired features. Since the current entry point for a search task in CodeGenie is a single method, test cases have to target at least one missing method (which may be inside an existing or non-existing class in the current project). Test cases similar to the ones presented in Figure 2 must be created. There can be multiple missing methods inside the test cases, though the current version of CodeGenie will search for one at a time.

Once the test class is created, CodeGenie is ready to extract information about the desired feature. The tool extracts the interface of the missing method, and the names of the missing method and its class. It does that by analyzing the compiler errors present in the test cases due to the missing method or class. The Abstract Syntax Tree (AST) of the test class is explored to extract the return type and argument types of the missing method. The names of the missing method and class are used as initial keywords and, for these keywords, camel-case splitting (*e.g.*, a class named *BinaryTree* generates keywords 'binary' and 'tree') and heuristics based on common Java naming conventions are used (such as splitting based on numbers – *e.g.*, *roman2numeral* generates 'roman', '2', and 'numeral' – and non-alphabetic characters – *e.g.*, *binary_tree* generates 'binary' and 'tree'). After gathering all information, CodeGenie formulates queries that can be processed by Sourcerer. These queries contain three parts: (1) keywords that must be present in the full qualified name of the entry point method; (2) return type of the entry point method; and (3) parameter types of the entry point method. For example, given a test case with the following assertion:

[4]The CodeGenie plugin, supporting material, and some of the test cases used for the applicability study (Section 6.1) are available at `http:\\sourcerer.ics.uci.edu\codegenie`.

assertEquals("trevni", Util.invert("invert"))

CodeGenie formulates the following query:

fqn_contents:(util invert)
m_ret_type_contents:(String)
m_sig_args_sname:String

The query above means: "look for an entry point method that contains the strings 'util' and 'invert' somewhere in the full qualified name, returns a value of type String, and receives a parameter of type String". Queries are different if query relaxation options are used. For instance, if the name of the class is disabled, the *fqn_contents* part of the query will only contain the 'invert' string; if the return type is disabled, the *m_ret_type_contents* part is not generated. The format of these queries is defined by the programming interface provided by the Sourcerer search service. Further details about this service are available elsewhere [9, 5].

For the integration and detachment of code slices, CodeGenie applies the merge by name strategy as used by Hyper/J [34]. It simply copies all classes inside the chosen code result into the developer's project and merges classes with coincident names. The merging is done by a union operation on the classes' structures (*i.e.*, all fields, methods, and inner classes of the original class and of the added class are present in the resulting class). If there are coincident methods, fields, or inner classes, the structures that were already present in the target project have priority over the ones being added. Before a code result is integrated with the current project, the name of the entry point method and of the class that contains it are automatically refactored according to the names present in the test cases.

Java annotations are used to track the integrated structures inside the developer's project. Each code structure added to the developer's project is annotated with *@FromSlice*, indicating from which code result it comes from. A *name* element is used to identify the code slice. In this way, when a detachment operation is triggered, CodeGenie can remove all code structures coming from the related slice by analyzing the annotations. Note that if an added code structure (*e.g.*, a method) is changed by the user, a detachment operation still removes the structure (*e.g.*, the related method), since it originally refers to integrated code. If the user again decides to integrate the same code structure, it is integrated as the original version in the code repository. In this way, if the user wants to consider the added code structure

a definitive part of the project, he can simply remove the annotation. This would prevent CodeGenie from detaching the code structure in the future.

To test the woven project we use the JUnit plugin that comes with Eclipse. CodeGenie communicates with this plugin so that when search results are integrated and tested, testing results are updated in the Search View (as shown in Figure 4(d)).

5.2. The Sourcerer Infrastructure

Sourcerer is an infrastructure for large scale analysis and indexing of source code [4, 8, 29]. It has been designed to support software applications on top of the services it provides. Figure 7 shows the general architecture of Sourcerer. General description of Sourcerer's architecture and its repository is available in [8, 29].

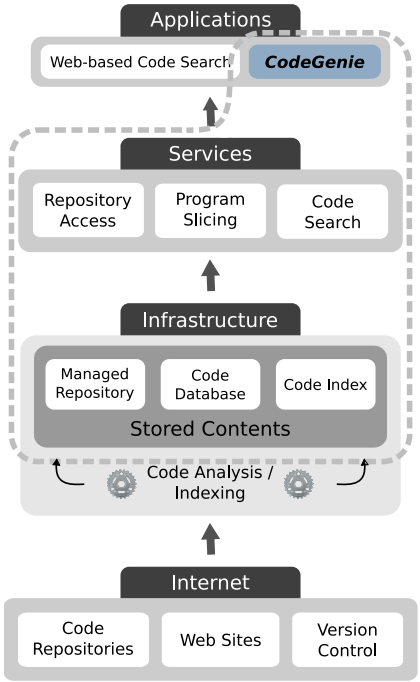


Figure 7: Sourcerer System Architecture (with CodeGenie and its required resources highlighted).

Sourcerer crawls the Internet looking for source code from various sources such as open source code repositories, public web sites and version control systems. The source code obtained from the Internet is analyzed, parsed,

and stored in the system in various forms: (i) *Managed Repository* keeps a versioned copy of the original contents of the source code and related artifacts such as libraries; (ii) *Code Database* stores the entire code-graph of all the code parsed maintaining full information about the structural dependencies in the code; and (iii) *Code Index* stores keywords extracted from the code during parsing for efficient retrieval.

All the artifacts managed and stored in Sourcerer are accessible through a set of Web-services. The details of these services are available elsewhere [9, 35]. In particular, CodeGenie uses these three services:

1. *Code Search:* This service implements the query processing facility. Client applications such as CodeGenie can send queries as combination of terms and fields (such as the one presented before) and the service returns a result set with detailed information on the entities that matched the queries. The query language is based on Lucene's implementation [6] and our extended query parser supports different query forms that CodeGenie requires to perform.
2. *Repository Access:* This service provides access to the Managed Repository in Sourcerer. All the code artifacts, libraries, and meta-data are accessible using this service. The Snippet Viewer (as shown in Figure 4(c)), for example, uses this service to request the snippets of the results to be viewed.
3. *Program Slicing:* This service implements the program slicing requirement for TDCS described earlier. Clients can request the slice by specifying an entry point of the program. In the case of CodeGenie the entry point is a method. The result from the slicer is a zip file, containing the newly fabricated program as well as some meta-data detailing any unresolved external references. Starting with any method in the *Code Database*, the slicer extracts all classes, interfaces, methods and fields necessary to ensure correct compilation and execution of the initial method. The slicer computes the transitive closure of certain relations (such as method and constructor calls and field accesses) based on the code dependency graph stored in the database, so no further code analysis is needed. Further analysis is done to ensure that the class hierarchy remains intact in cases where it is relevant, and that implemented interfaces are pulled in when appropriate. For example, if two classes are extracted and one of them is a descendant of the other, then the extends relation will be extracted, as well as all classes between

them in the hierarchy. A conservative approach is taken with respect to relations terminating outside of the project (*i.e.*, library calls), so the explicit types found in these relations are fully extracted. Information regarding these library accesses are included as meta-data.
For the purposes of search result extraction, it is beneficial to decrease the granularity in the standard definition (of slicing) from statements to code structures (methods, fields, classes, etc.), as that is the smallest unit of functionality that can be searched for. Furthermore, the standard slicing criterion can be relaxed to instead just specify a code structure, ignoring the variables entirely. With these modifications, computing in effect an approximate static forward slice enables the extraction of a specified method or class as well as everything on which it transitively depends. The slice is only approximate, as computing the true minimal slice is unsolvable. More information on Sourcerer's slicing service can be found elsewhere [35, 36].

6. Evaluation

Our evaluation of TDCS in the context of auxiliary functionality consists of two parts. First, we explored CodeGenie with a number of features, checking whether we could find working implementations for them. Second, we conducted a controlled experiment with 34 senior Computer Science students to evaluate the performance of CodeGenie against a well-known code search engine – Google Code Search (CS) [3] – in the process of pragmatic code search and reuse. To gather additional evidence on the performance of CodeGenie we also replicated the experiment with 7 graduate students. We chose Google CS for the comparison because it is a well known search engine and it represents other similar code search web applications. To check whether it was worth reusing the features using code search instead of simply implementing them, we also asked the students to implement the features manually, without the aid of any searching tool. Before and after the performance studies, we also conducted surveys to collect the impressions of students while using CodeGenie. Results are also summarized in this section.

6.1. Applicability study

We used CodeGenie to search several features suggested by members of our group as relevant functionality they would like to reuse (by informally surveying them), from a list of common functionality used by Hoffmann et

al. [17] in a code search study[5], and from examples used by Reiss [40] to illustrate his approach. Note that we selected features that could not be reused by simply calling library functions available in the Java distributions: they should require at least a minimum of 10 lines of code. To provide an idea of size, we collected the average number of lines of code (**LOC**) of the working candidates, when integrated to the workspace. We measured the number and percentage (**#adeq.**, **%adeq.**) of results that could be reused after being integrated to the developer's project and the number and percentage of results that could compile and run, though did not implement the feature according to the tests (**#run.**, **%run.**). Results are shown in Table 1 (features are sorted by their size in LOC).

These numbers are an evidence that TDCS using CodeGenie is not only feasible but also effective for auxiliary functionality, since we were able to reuse matching results for all 34 sample features. Note that, on average and per query, 71.75% of the matched candidates were reusable features that were successful against the test case queries. With respect to the amount of running results, on average, 95.43% of all results compiled and ran, an evidence that it is possible to slice, integrate, and compile features in the proposed way. The average number of candidates per query was 3.21, an evidence of the preciseness of TDCS (few candidates per query).

Some of the results required minor manual fixes before they could be compiled (for instance changing a private method to public). However, less than 10% of the results presented in Table 1 required this type of intervention.

6.2. Performance study

In this experiment, each of the 40 senior Computer Science students was asked to find/implement three features and have them working in an Eclipse workspace, using the following approaches: (1) CodeGenie, (2) Google Code Search, and (3) Manual, *i.e.*, implementing it by hand. The students had medium to advanced experience in Java and Eclipse (medium on average, according to a survey given to them). The three features they were supposed to find/implement were the following: (1) Conversion of Arabic numbers to Roman numerals (such as in the example used before); (2) Complement of DNA strings using the rule: adenine – thymine, cytosine – guanine (when 'a' or 'A' is found, change it to 't' or 'T', and vice-versa, and so on); and

[5]Sample queries were obtained through personal communication with the authors.

Table 1: Results of feature searches using CodeGenie.

Feature	total	#run.	#adeq.	LOC	%run.	%adeq.
Joining a list of strings in a single string	6	6	2	12	33.33	100.00
Trimming left spaces from a string	4	4	4	13	100.00	100.00
Extracting a file name from a full path	2	2	1	17	50.00	100.00
Computing the largest common prefix of two strings	2	2	2	20	100.00	100.00
Decoding a URL	1	1	1	23	100.00	100.00
Capitalizing first letters of a string	1	1	1	24	100.00	100.00
Converting normal strings to hexadecimal strings	6	4	2	27	33.33	66.67
Removing carriage return / line feed from strings	4	4	2	31	50.00	100.00
Sharpening an image	1	1	1	33	100.00	100.00
Capturing the screen into an image	1	1	1	35	100.00	100.00
Saving an image in JPG format	1	1	1	35	100.00	100.00
Converting arabic numbers to alphanumerics	6	6	1	38	16.67	100.00
Converting hyphenated strings to camel case strings	2	2	1	38	50.00	100.00
Generating the complementary DNA seq.	3	3	1	40	33.33	100.00
Converting camel case strings to phrases	3	3	1	42	33.33	100.00
Computing the MD5 hash of a string	10	9	9	45	90.00	90.00
Encoding Java strings for HTML displaying	2	2	2	48	100.00	100.00
Scaling an image	1	1	1	49	100.00	100.00
Converting byte arrays to hexadecimal strings	2	2	2	50	100.00	100.00
Filtering folder contents with specific file types	1	1	1	50	100.00	100.00
Rotating an image	2	2	1	50	50.00	100.00
Generating the reverse complementary DNA seq.	2	2	2	53	100.00	100.00
Encrypting a password	2	2	2	55	100.00	100.00
Blurring an image	2	2	2	56	100.00	100.00
Printing formatted strings for elapsed times given in ms	2	2	1	68	50.00	100.00
Converting arabic numbers to roman numerals	6	6	2	72	33.33	100.00
Converting hexadecimal strings to normal strings	6	4	2	72	33.33	66.67
Sorting objects using QuickSort	4	2	2	74	50.00	50.00
Computing the Easter holiday for a given year	2	2	1	101	50.00	100.00
Counting lines of a text file	1	1	1	103	100.00	100.00
Computing the Soundex hash of a string	8	8	8	108	100.00	100.00
Unzipping files	7	5	3	110	42.86	71.43
Zipping files	5	5	2	118	40.00	100.00
Parsing a CSV file	1	1	1	240	100.00	100.00
Avg	3.21	2.49	1.97	57.35	71.75	95.43

(3) Reversion of strings (for instance, given 'CodeGenie', it should return 'eineGedoC'). Before starting the experiment, we ensured that each feature could be found/extracted using both CodeGenie and Google CS by trying them. We did this because our main concern was the method of reuse proposed by each approach, and not the size of the code database. We also ensured that the features could be implemented by any undergraduate student in relatively short time, by previously analyzing the implementations.

For CodeGenie, each student should create a test case class for the desired feature, trigger the search and explore the results, until satisfied with a working implementation. For Google CS, they should go to the website, look for implementations, and extract them by copying and pasting results

into the workspace, until satisfied with a working implementation. No test cases were required for this approach. For the manual approach they should implement the feature themselves without the aid of any code search facility, until satisfied with a working implementation. No test cases were required for this approach either.

The time spent by each student in the whole process of finding/integrating/implementing a feature for each approach was recorded. For instance, for CodeGenie, we also considered the time spent to create test cases. Each student implemented each feature using one of the approaches at a time and the assignment of features and approaches was randomized among them, so that combinations were evenly distributed. We did this to cancel the effects of the difference among the features and of the order of application of the approaches. After gathering the data, some outliers were removed to obtain homogeneity (that is why only 34 results are shown). We believe these outliers were due to the inexperience in Java or Eclipse of some students. Table 2 shows the results of our experiment.

CodeGenie was the fastest approach for 20 out of 34 students (59%), and faster than Google CS for 22 students (66%). On average, CodeGenie was around 50% faster than both Google CS and the manual approach. Although the means differ visually, statistically speaking, such difference can be significant or not. To test whether the means do differ statistically, we must apply a t-test under the null hypothesis that the means are equal. If they are significantly unequal, the reached p-value has to be less than a threshold (usually set to 0.05, that is, a 95% confidence level). In our case, since the same students applied the different approaches, we can make use of paired t-tests. Therefore, to check whether the time means differed significantly, we conducted a Welch two sample paired t-test to compare the means, two by two. With 95% confidence level, the tests did indicate a significant difference between the means, both for CodeGenie against Google CS (p-value = 0.01548) and for CodeGenie against the manual approach (p-value = 0.005006). Since the Welch t-test assumes normality, we also conducted a Wilcoxon test, that does not require this assumption. Again with 95% confidence level, the Wilcoxon test also indicated a significant difference among the means, both for CodeGenie against Google CS (p-value = 0.03028) and for CodeGenie against the manual approach (p-value = 0.004964).

Note that students were not asked to test the resulting application while using the Google CS and manual approaches, even though they did it for CodeGenie (since creating test cases is part of the approach itself). If test

Table 2: Time (in minutes) spent to find/implement a feature in each approach by each student.

Subject	CodeGenie	Google CS	Manual
1	15.00	30.00	40.00
2	4.00	6.00	9.00
3	25.00	10.00	20.00
4	1.61	6.83	1.91
5	30.00	10.00	3.00
6	3.00	5.00	23.00
7	8.00	10.00	10.00
8	10.00	8.00	15.00
9	6.00	11.00	16.00
10	5.48	4.51	5.45
11	8.00	3.00	10.00
12	10.00	40.00	30.00
13	12.00	30.00	31.00
14	22.00	14.00	15.00
15	8.00	6.00	15.00
16	2.55	4.50	23.35
17	4.50	25.00	19.00
18	4.00	6.00	8.00
19	4.25	5.51	15.00
20	5.00	25.00	22.00
21	7.00	5.00	17.00
22	15.00	40.00	10.00
23	12.00	8.00	6.00
24	2.50	3.33	11.00
25	5.00	10.00	10.00
26	5.00	20.00	15.00
27	7.00	30.00	25.00
28	20.00	40.00	15.00
29	5.00	10.00	15.00
30	6.00	8.00	25.00
31	10.00	10.00	5.00
32	19.00	15.00	9.00
33	27.00	43.00	35.00
34	11.00	8.00	6.00
Avg	9.996765	15.02	15.756176

cases were required for the other approaches the difference among the means would probably be greater. Moreover, code retrieved using Google CS and implemented manually in the experiment could contain faults that would possibly be found if tested. For example, by examining an implementation reused by a student through a Google CS search (the Arabic to Roman conversion), we noticed that it did did not pass the test cases designed by another student to search the same feature using CodeGenie (it did not deal with the number 0, which does not have a Roman counterpart); that is, it contained a fault that was not revealed because the retrieved code was not tested.

6.2.1. Replication with graduate students

To gather more evidence about the performance of CodeGenie, we decided to replicate the experiment with graduate students. However, at this time we were able to gather only 7 – 5 MSc and 2 PhD – student volunteers, so we did not expect to reach statistically significant results (only exploratory evidence to support or reject the results achieved for undergraduate students). Again, the students had medium to advanced experience in Java and Eclipse (medium on average, according to a survey given to them), and the same procedure and set of features were used to be able to compare results. Table 3 shows the results of the replicated study.

Table 3: Time (in minutes) spent to find/implement a feature in each approach by each student.

Subject	**CodeGenie**	**Google CS**	**Manual**
1	11.00	9.00	60.00
2	9.00	8.00	11.00
3	5.24	4.27	7.5
4	9.60	5.94	16.67
5	10.00	19.00	7.00
6	5.37	5.55	37.08
7	4.71	9.55	22.45
Avg	7.8457	8.7585	23.1000

On average, CodeGenie was slightly faster than Google CS – by 0.92 minutes –, and more than three times faster than the manual approach. At this time, CodeGenie was the fastest approach for 2 of the students (28%), and faster than Google CS for other 3 students (43%). Note that when CodeGenie was faster than Google CS, it was faster by 4.67 minutes, on average; while when Google CS was faster than CodeGenie, it was faster by 1.9 minutes, on average. To check wether the time means differed significantly, we again conducted a Welch two sample paired t-test to compare the means, two by two. As expected, due to the small sample size, with 95% confidence level, the difference between the performance of CodeGenie and Google CS was not significant (p-value = 0.2783). With respect to the performance of CodeGenie against the manual approach, the test did indicate a significant difference (p-value 0.04195). Since the Welch t-test assumes normality, as commented before, we conducted a Wilcoxon test for the difference between the performance of CodeGenie and the manual approach. The test indicated a significant difference between the means (p-value = 0.04688).

Although at this time the sample size of subjects was limited – which has impacted on the significance of the results –, we can say that the study sup-

ports the evidence achieved in the main study with undergraduate students, since CodeGenie performed slightly better than Google CS and significantly better than the manual approach, on average.

Further analysis of both undergraduate and graduate students' data is available elsewhere [10].

6.3. Threats to validity

Both applicability and performance studies are exploratory and present limitations that must be considered when interpreting the results. The primary threats to validity are related to subject representativeness, affecting the ability of our results to generalize. The studies were applied only to small features – 57.35 lines of code on average – and it is still unclear how TDCS would scale for larger ones. In any case, in this paper we target the reuse of auxiliary functionality, which does seem to be handled adequately by TDCS. Moreover, some of the features explored in the applicability study were larger – more than 100 lines of code in two or more classes (see Table 1) –, which indicates that with improvements it may be possible to handle larger features.

The performance study was applied to senior and graduate Computer Science students, which does not guarantee the generalization of the results to professionals. However, some authors suggest that there are cases where students can provide an adequate model of the professional population [21, 38]. Studies that show opposite trends between these groups sometimes have concluded that the effectiveness of techniques depended to a large extent on skill (for instance, [7]). Intuitively, this evidence could work in favor of the presented approach, since more experienced developers could probably make a better use of TDCS (for instance, developers with experience in developing test cases and using JUnit will probably reach good results faster, since they will require less time in this step). However, more empirical evidences should be attained to be able to better generalize our findings.

6.4. Usage survey

We also surveyed the students to collect their opinions on pragmatic code search and reuse through CodeGenie and Google CS. Most of the undergraduate students preferred using CodeGenie over Google CS in the search/reuse tasks (61%). The other 39% preferred either implementing the features by hand, or using Google, because they were more familiar with these approaches. On the other hand, most of the graduate students preferred using

Google CS (57%) over CodeGenie (43%), mainly because they were more familiar with it. Moreover, most graduate students did not explore CodeGenie the best way: since it was the first time they were using the tool, some features – such as the Snippet Viewer – were not used. This indicated that training and practice is required to make developers fully explore the tool.

The main limitation of Google CS indicated by the students was having to manually navigate through results, extracting dependencies and copying and pasting code into the workspace without knowing whether it would compile/work. Moreover, some of them also found restrictive to search for code using keywords only. We believe that these are the main disadvantages of Google CS – used by itself – that makes CodeGenie perform better most of the times.

We also asked the students to evaluate the usability of CodeGenie by giving it one of the following grades: Great, Very Good, Good, Fairly Good, Bad, Very Bad. Among the undergraduate students, 10 (29%) graded it Great; 11 (32%) graded it Very Good; 8 (24%) graded it Good; and the remaining 5 (15%) graded it Fairly Good. Among the graduate students, 5 graded it Very Good (71.4%), 1 graded it Good (14.3%), and 1 graded it Fairly Good (14.3%).

We collected the opinions of the students on limitations of the tool for future enhancements. For instance, one of the students found the feedback when testing the results hardly noticeable (only the label of the related search result is changed); other students found somehow restrictive having to choose adequate names for the desired class/method to match more results. Still with respect to usability and limitations of the tool, while conducting the replication of the performance study – described in Section 6.2.1 –, we asked some of the students to record a video of their usage of CodeGenie. We then analyzed the videos to check how the tool was used, and how it could be further enhanced. One of our main findings in this analysis was that the experience in using the required technologies – such as JUnit – is of great impact on the performance. Moreover, we also noted that some students spent considerable time in small technical issues, such as how to trigger the CodeGenie search menu and how to create a test class suitable for searching (*i.e.*, with a missing class or method). All feedbacks will be used for future enhancements of the tool and also to possibly generate adequate training material.

7. Related Work

Since the late 1960s, software reuse has been a well advocated and widely explored topic in software engineering [31, 25, 32, 40]. There are several aspects of reusability that make it a hard problem, including creating reusable code, finding code to reuse, and adapting reusable code to the new application [40]. Several approaches to tackle such aspects have been proposed, but only recent work explore the vast quantity of code available in open source repositories.

During the early 1980s, several advances in the software reuse field originated in Freeman's research group [15], and also in industrial reuse projects in Japan, the U.S., and Europe [16]. More related to this paper, in the 1990's, program semantics was commonly explored as a means to enhance the search of reusable components [40]. Zaremski et al. [50] presented a method for achieving this goal by using signature information derived from components. This approach was later extended to matching more formal semantics using λprolog and Larch-based specifications [41].

Podgurski & Pierce [37] developed Behavior Sampling (BS), a retrieval technique which executes code candidates on a searcher-supplied sample of operational inputs and compares the outputs to outputs provided by the searcher. Differently from TDCS, inputs for the desired functions are randomly generated and expected outputs have to be supplied by users. TDCS implements one of the extensions considered by Podgurski & Pierce to improve BS: the ability to retrieve code based on arbitrary tests. PARSEWeb [46] is a tool that combines static analysis, text-based searching, and input-output type checking for a more effective search.

According to Reiss [40], all these early techniques did not really succeed because either they require too little specification of the desired feature, or too much. Signature or type matching used by themselves do not seem to be very effective, although PARSEWeb shows that in combination with textual search they can provide more interesting results. Full semantic matching requires the specification of too much information and thus is quite difficult to accomplish. Our approach uses test cases, which are generally easy to check and easier to provide. In a recent work, Reiss [40] also incorporates the use of test cases and other types of low-level semantics specifications to source code search. He also implements various transformations to make available code work in the current user's context. However, in his approach, there is no slicing facility and therefore only a single class can be retrieved

and reused at a time. Moreover, the presented implementation is a web application, which also requires code results to be copied and pasted into the workspace in an *ad hoc* way. CodeGenie has the advantage of being tightly integrated with the IDE: code candidates can be seamlessly integrated to – and detached from – the workspace.

The idea of using test cases to search and reuse software pieces has also been explored by Hummel et al. [22]. The presented approach – named *extreme harvesting* – requires a basic implementation of the class structure (a *stub*) before searching, which is not required by TDCS (only test cases are needed). With respect to the slicing facility, which prevents developers from retrieving pieces of software unrelated to the desired functionality, it is not clear whether the authors' infrastructure provide this type of capability. On the other hand, Sourcerer, the infrastructure used by CodeGenie, maintains code relations in its database, which allows it to support a fine-grained slicing of candidate features. Recently, the same group has proposed an evolution of their approach in a tool named CodeConjurer [23]. This implementation presents several enhancements with respect to the search interface in the form of a new type of proactive reuse recommendation. Moreover, the code repository used by CodeConjurer is very large, maintaining over 10 million indexed files. However, to the best of our knowledge, the slicing facility presented is still limited in comparison with Sourcerer, which computes a more fine-grained slice (*i.e.*, in the method level). This is due to the way Sourcerer manages code in its database, recording detailed information about the relations among modules [35, 29].

Program slicing has been used before for purposes of assisting code reuse. Our approach is similar to many others in its reliance on program dependency graphs. Transform slicing [26] is one such approach, and is designed to extract reusable functions from existing programs. The primary difference lies in our relaxation of the slicing criterion, while transform slicing instead further constraints it. Although not program slicing, Holmes's feature sketching approach to unanticipated reuse allows the user to manually explore dependencies [18]. The slicer presented here performs similar tasks automatically.

Modern software engineering tools are bringing more sophisticated search capabilities into the development environment extending the traditionally limited browsing and searching capabilities [19, 30, 44, 39, 42]. These tools vary in terms of the features they provide but some common ideas that are prevalent among them are the use of the developer's current context

to generate queries and the integration of ranking techniques for the search results.

In summary, TDCS support in CodeGenie reduces some drawbacks of code-based reuse while still retaining its basic advantage of having a small *cognitive distance* [25]. The problem of *selection*, *specialization* and *integrating* that exists in Code-scavenging [25] techniques of reuse is greatly reduced by the automation CodeGenie provides; in particular, with test-driven search and validation of code results for expected behavior.

8. Conclusion

In this paper we have presented an approach to source code search and pragmatic reuse based on test cases. To provide evidence of the feasibility of TDCS, we implemented CodeGenie and performed an exploration with 34 examples of auxiliary functionality. We also conducted a controlled experiment with 41 students to evaluate the performance of CodeGenie against Google Code Search and a manual approach in the reuse of auxiliary functionality. While these studies remain exploratory, they provide evidence of the feasibility and good performance of TDCS for this kind of feature.

In the future we intend to study the scalability of TDCS, exploring the reuse of larger and more general functionality. Moreover, we also want to improve our tool to make it more effective. For instance, we plan to extend CodeGenie to make it automatically test candidate results. Another considered extension is enhancing the search mechanism to allow matches based on synonyms and common code abbreviations (for example, *DB* for *database*), and also support search for *aspects* (as in *aspect-oriented programming*), other types of crosscutting modules, and test code. Other enhancements suggested by the students involved in the performance study will also be taken into account.

The way test cases should be designed in TDCS with respect to the range of tested inputs is also an important issue. As commented in Section 2, in TDD test cases are usually low level and no type of formal testing criteria is used [14]. This is because test cases are used primarily as a design decision facility [11]. This design decision aspect of test cases is also useful to TDCS, because it supports the search of specific self-contained modules. However, to enhance confidence in the retrieved code, we could design test cases in a more systematic way. Thus, we also consider the integration of testing criteria such as equivalence partitioning [33] to TDCS to deal with this limitation. This

would obviously increase the time required to start a search, so a balance between completeness and agility must be carefully studied.

Acknowledgements

We thank Ellen Barbosa, Pierre Baldi, Ricardo Morla, and Mario Andrade for their contribution to this project; Eclipse Innovation Grant, CNPq and FAPESP for financial support.

References

[1] Scrapheap Challenge Workshop, OOPSLA 2005. `http://www.postmodernprogramming.org/scrapheap/workshop`.

[2] Koders web site. `http://www.koders.com`.

[3] Google Code Search. `http://www.google.com/codesearch`.

[4] Sourcerer web site. `http://sourcerer.ics.uci.edu`.

[5] Sourcerer Web-services. `http://sourcerer.ics.uci.edu/services`.

[6] Lucene web site. `http://lucene.apache.org/`.

[7] E. Arisholm and D. Sjøberg. A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Technical report, Simula Research Laboratory, June 2003. (available at: `http://www.simula.no/photo/tr20036sep10.pdf`).

[8] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Comp. to the 21st ACM SIGPLAN OOPSLA*, pages 681–682, New York, NY, USA, 2006. ACM Press.

[9] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *SUITE '09: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[10] S. K. Bajracharya. *Facilitating Internet-Scale Code Retrieval*. PhD thesis, University of California Irvine, September 2010.

[11] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[12] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] R. Cottrell, R. J. Walker, and J. Denzinger. Jigsaw: a tool for the small-scale reuse of source code. In *ICSE Companion '08: Companion of the 30th Int'l Conf. on Softw. Eng.*, pages 933–934, New York, NY, USA, 2008. ACM.

[14] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, 2005.

[15] P. Freeman. Reusable software engineering: Concepts and research directions. In *Proceedings of the Workshop on Reusability in Programming*, pages 129–137, 1983.

[16] H. Gall, M. Jazayeri, and R. Klösch. Research directions in software reuse: where to go from here? *SIGSOFT Softw. Eng. Notes*, 20(SI):225–228, 1995.

[17] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST '07: Proc. of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, New York, NY, USA, 2007. ACM.

[18] R. Holmes. Unanticipated reuse of large-scale software features. In *ICSE '06: Proc. of the 28th Int'l conference on Softw. Eng.*, pages 961–964, New York, NY, USA, 2006. ACM Press.

[19] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[20] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *ICSE '07: Proc. of the 29th Int'l Conference on Softw. Eng.*, pages 447–457, Washington, DC, USA, 2007. IEEE Computer Society.

[21] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. *Emp. Softw. Eng.*, 5:201–214, 2000.

[22] O. Hummel and C. Atkinson. *Agile Processes in Software Engineering and Extreme Programming*, chapter Supporting Agile Reuse Through Extreme Harvesting, pages 28–37. Springer-Verlag, 2007.

[23] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[24] IEEE. IEEE Standard Glossary of Software Engineering Terminology. New York, 1990. IEEE Computer Society Press.

[25] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[26] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, 1997.

[27] O. A. L. Lemos, S. Bajrachary, and J. Ossher. Codegenie: a tool for test-driven source code search. In *OOPSLA '07: Comp. to the 22nd ACM SIGPLAN OOPSLA*, pages 917–918, New York, NY, USA, 2007. ACM.

[28] O. A. L. Lemos, S. Bajrachary, J. Ossher, P. C. Masiero, and C. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proc. of the 22nd IEEE/ACM Int'l conference on Automated Softw. Eng.*, pages 525–526, New York, NY, USA, 2007. ACM.

[29] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, Apr. 2009.

[30] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.

[31] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proc. of NATO Softw. Eng. Conference*, pages 138–150. Garmisch, Germany, 1969.

[32] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Trans. Softw. Eng.*, 21(6):528–562, 1995.

[33] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. John Wiley & Sons, 2nd. edition, 2004.

[34] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '01: Proc. of the 23rd Int'l Conference on Softw. Eng.*, pages 821–822, Washington, DC, USA, 2001. IEEE Computer Society.

[35] J. Ossher, S. Bajracharya, and C. Lopes. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source java projects. In IEEE, editor, *MSR '09: Proc. of the 6th IEEE Working Conference on Mining Software Repositories*, pages 183–186, 2009.

[36] J. Ossher, S. K. Bajracharya, and C. V. Lopes. Automated dependency resolution for open source software. In *MSR '10: Proc. of the 7th International Working Conference on Mining Software Repositories (Co-located with ICSE 2010)*, pages 130–140. IEEE, 2010.

[37] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303, 1993.

[38] A. Porter and L. Votta. Comparing detection methods for software requirements inspection: A replication using professional subjects. *Emp. Softw. Eng.*, 3:355–380, 1995.

[39] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS - an eclipse plug-in for source code exploration. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

[40] S. P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[41] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In *International Conference on Logic Programming*, pages 173–187, 1991.

[42] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM Press.

[43] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *IWPC*, page 180, 1998.

[44] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 905–908. ACM, 2006.

[45] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[46] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2007. ACM.

[47] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[48] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–305, New York, NY, USA, 1999. ACM.

[49] M. Weiser. Program slicing. In *ICSE '81: Proc. of the 5th Int'l conference on Softw. Eng.*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[50] A. M. Zaremski and J. M. Wing. Signature matching: a key to reuse. *SIGSOFT Softw. Eng. Notes*, 18(5):182–190, 1993.