THE UNIVERSITY OF CALGARY

Supporting Repetitive Small-Scale Changes

by

Mark Michael McIntyre

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
SEPTEMBER, 2007

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Supporting Repetitive Small-Scale Changes" submitted by Mark Michael McIntyre in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Robert James Walker
Department of Computer Science

Dr. Jörg Denzinger
Department of Computer Science

Dr. Dietmar Pfahl
Department of Electrical and Computer Engineering

Date

Abstract

When modifying a software system, a developer may find it necessary to repeat a given change throughout its source code. While the change itself may not be difficult to implement, discovering the locations where it should be applied can be onerous. Syntactic differences in otherwise semantically similar code can render traditional tools ineffective. This thesis describes a heuristic search technique to help find locations required to complete a repetitive small-scale change (RSC). By observing the developer perform a change once, it is possible to infer semantic information about that change and automatically suggest locations where the same change might need to be made. This technique is implemented in a tool called Reverb. The utility of this technique is evaluated by comparing Reverb's search results against those of traditional approaches, for RSCs conducted on two open source applications; Reverb is found to have superior recall and precision in the cases evaluated.

Acknowledgements

It is a pleasure to acknowledge the people who helped make this thesis possible.

First, I wish to thank my supervisor Dr. Robert Walker, to whom I cannot overstate my gratitude. Without his advice, encouragement, teaching, and occasional whip-cracking, I would have been lost.

I would like to thank my friends and colleagues at the University of Calgary. Especially Kevin Viggers, for convincing me to seek out my own topic and re-igniting my enthusiasm; Reid Holmes, for his advice, inspirations, experience, humour, and friendship; and, with no less emphasis, Joseph Chang, Brad Cossette, Rylan Cottell, Puneet Kapur, Shafquat Mahmud, Mohammad Minhaz, Bhavya Rawal, Jamal Siadat, and Carmen Zannier for their consultation, advice, and help in my adjustment to life in Calgary.

For helping me express my technique in mathematical notation, I thank Dr. Jörg Denzinger.

For their unconditional love and support, I thank my parents, Arthur and Linda; my sister, Lindsay; and my partner, Mark Driedger. Thank you all. I could not have finished this thesis without you.

Finally, I owe much of my fascination with technology to a man whom I do not know. During the summer of 1986, it was decided—without my consultation—that my birthday present was to be a skateboard instead of the Nintendo Entertainment System for which I had repeatedly requested. It was while wandering the isles of Toys 'R Us, ostensibly to look at skateboards (I had purposely meandered off alone to see the Nintendo section), that my parents witnessed a man, also looking for a

skateboard as a gift, slip and concuss himself while trying out the new toy. Sometime between the arrival of the fire department and ambulance, and without knowing if this man had regained consciousness, my parents conceded that a Nintendo might be the safer gift. My first Nintendo was received on my sixth birthday. I suppose I've been enchanted by technology ever since.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As software systems evolve [Parnas, 1972; Belady and Lehman, 1976; Lehman and Parr, 1976; Lehman and Belady, 1985], small-scale changes are often required [Purushothaman and Perry, 2005]. While these changes can be as small as a few lines of code, they may occur in repetition—requiring attention in multiple locations throughout a software project. These changes may also vary between repetitions, creating difficulties for many traditional approaches. Any small-scale change task can potentially be a repetitive small-scale change (RSC); the difficulty lies in determining whether additional targets exist and discovering the locations of those targets.

We introduce this thesis with a short example: Debug statements help developers understand the execution of their code by outputting information to a log file or console screen as a program's statements are executed. While this information is useful to the developer, it should be removed from the program before it is deployed so end-users do not have to suffer through the output, or its resulting performance decrease.

Traditionally, developers guard their debug statements with a `DEBUG` flag, a condition that the program checks before executing any debug-related statements. If the program is configured to output the debug information by setting this flag to `True`, then the debug statements are processed; otherwise they are ignored. Using the syntax of the Java programming language, this often manifests itself as follows:

```
if(DEBUG)
    System.out.println("Value␣of␣x␣is:␣" + x);
```

This approach, while effective, has drawbacks. The `DEBUG` flag can only be on or off, the results cannot be configured (by, for example, filtering, changing the order, or re-formatting the outputted statements), and the differentiation of informational, debugging, error, and warning situations is difficult.

A better solution is to make use of a logging framework. Logging frameworks relegate management of the debug information to a centralized component, and new features can be added to the framework without much disruption. A typical logging framework may use the following syntax:

```
log.debug("Value␣of␣x␣is:␣" + x);
```

where `log` is a reference to a logging framework object, `debug` is a message indicating that some debugging information is to be collected, and the parameter is a `String` representing the debug information.

If a developer wishes to change all guarded print statements to use a logging framework instead, he runs into practical difficulties. A lexical search for "debug" is likely to return comments about the need to debug certain functionality. The search may also miss diagnostic output statements that are not guarded with a test of the `DEBUG` flag, or be unable to distinguish the cases that denote informational output from those that denote warnings or errors.

In general, a problem arises when attempting to discover the targets of an RSC using traditional tool support. The targets of an RSC can be similar, but not identical. Brute force investigation requires much time and effort from the developer,

likely leading to mishandled situations. Clone detectors [Baxter et al., 1998; Kamiya et al., 2002; Li et al., 2006; Duala-Ekoko and Robillard, 2007] and lexical search tools [Crochemore and Perrin, 1988; Miller and Marshall, 2004] are geared towards discovering identical matches. Regular expressions [Brzozowski, 1964; Baeza-Yates and Gonnet, 1989; Clarke and Cormack, 1997] and exact query languages [Holt, 1999; Janzen and Volder, 2003; Beyer, 2006] require special attention to the lexical pattern of the change, if one exists. Refactoring tools [Opdyke, 1992; Griswold and Notkin, 1993; Tokuda and Batory, 1999; Fowler, 2002] provide pre-defined transformations with a heavy emphasis on correctness; new refactoring tools must be created for specific problems.

1.1 Proposed Technique

Our approach to finding RSC target locations is to extract a set of facts from an example change and use those facts to search for locations where a similar change should be made. Facts can be extracted from any code segment and can represent variable declarations, messages and parameters passed between references, control flow graph information, and other explicit or inferred data.

When combined with information about a change, this extracted information can be used as a search heuristic to locate places related to that change. More specifically, given a "before" and "after" snapshot of a change, the set of facts extracted from the code segment can be reduced to those most relevant to the change. Searching the remainder of the codebase for these change-related facts should yield locations likely to require the same change.

To evaluate our approach, we have implemented it as a tool called Reverb. Reverb observes the developer as he navigates about a source project to automatically create the necessary "before" and "after" snapshots of small-scale changes. Upon request, Reverb will search the codebase for locations where the most recently observed change could also be applied. The results are returned as a sorted list of methods relevant to the query and can be browsed sequentially until the developer is satisfied that the RSC has been completed.

1.2 Thesis Statement and Organisation

The thesis of this dissertation is that a semi-automated heuristic search discovers target locations for repetitive small-scale changes with better precision and recall than traditional techniques, and with a greater tolerance to variation.

The remainder of this thesis is organized as follows. Chapter 2 presents a motivational example to better demonstrate the problem. Chapter 3 describes our proposed solution and its implementation. Chapter 4 evaluates the performance of our technique relative to traditional approaches. We end with a discussion of our technique's drawbacks and future direction (Chapter 5) and a presentation of related work (Chapter 6).

Chapter 2

Motivation

The focus of this thesis is on assisting repetitive, small-scale changes (RSCs). To better understand the nature of these changes, and why they are problematic, we examine two motivational scenarios. In Section 2.1, we consider improving the logging mechanism in Apache Struts 1.1 to use a logging framework instead of guarded "debug" statements. In Section 2.2, we consider using a new for-loop syntax introduced by Version 5 of the Java programming language in an open-source drawing application called JHotDraw. We discuss a classification of changes in Section 2.3.

2.1 Migrating to a Logging Framework

Consider a developer charged with the task of ensuring that guarded print statements in Apache Struts 1.1 (a popular web application framework written in the Java programming language) are made to consistently use a logging framework.

The Struts codebase, for the most part, uses such a framework. However, there are exceptions, as demonstrated in Figure 2.1. Code Sample A uses a logging framework; a string representing some textual debug information is sent to the logger, and depending on how that logger is configured, the information is either ignored or sent to a log file, memory dump, or console. Code Sample B uses a `DEBUG` flag to guard a standard `println` message. The debug output is either output directly to the console where it cannot be organized or re-ordered, or not at all. Both of these

```
// Code Sample A
log.debug("LogonAction: User '"
        + user.getUsername()
        + "' logged on in session "
        + session.getId());

// Code Sample B
if(debug)
        System.out.println("Got definition "
                           + catalogDef);
```

Figure 2.1: Two inconsistent debug statements found within the Apache Struts distribution.

examples were found within the Apache Struts distribution, but they output debug information inconsistently. Ideally, all debug statements would be output using the debugging framework demonstrated in *Code Sample A* instead of the guarded print statement in *Code Sample B*. The developer wishes to find examples of the latter and modify them. While modifying any particular guarded print statement to use a logging framework is a straightforward procedure, discovering the locations where such modifications are required is more onerous.

The simplest approach to discovering these locations may be to use the search and replace features of the developer's integrated development environment (IDE). However, a search for the term "debug" over the Apache Struts codebase returns 234 hits. The developer must look through each of these hits before determining that 214 of them are false, either using the `debug(...)` method already, being embedded within the program comments, or otherwise being unrelated to the task at hand. With sufficient determination, it is possible to look through all the unordered results; however, time pressures and fatigue may result in missed legitimate occurrences.

```
catch (DefinitionsFactoryException ex) {
        if (debug)
                ex.printStackTrace();
        // Save exception to be able to show it later
        saveException(pageContext, ex);
        throw new JspException(ex.getMessage());
}
```

Figure 2.2: A `DEBUG` flag used for non-logging purposes in Apache Struts

```
// Code Sample A

/**
 * Debug flag.
 * @deprecated This will be removed in a release after Struts 1.2.
 */
public static final boolean debug = false;

// Code Sample B

/** debug flag */
public static boolean debug = true;
```

Figure 2.3: Different global `DEBUG` flag used in Apache Struts. The word "debug" also occurs in the comments.

Another approach may be to use the syntactic search features of the IDE; however, a similar problem manifests itself. Not all references to a global variable named "debug" are necessarily used for guarded print statements (Figure 2.2), and the use of local variables and multiple global variables can complicate matters (Figure 2.3).

2.2 Migrating to a New Iteration Syntax

A for-loop is a programming construct that allows a statement or sequence of statements to be executed in repetition, once for every incrementation step until a speci-

fied condition is reached. In some object-oriented languages, this syntax can be used in conjunction with the Iterator design pattern [Gamma et al., 1993] to help traverse collections of items.

The following example, written using the syntax of the Java programming language, shows a collection traversal using a for-loop. An object of type `Iterator` is initialized from a collection *c*. Using the for-loop, each of the items in the collection are traversed via the `Iterator`'s knowledge of the collection. Subsequently, each of these items is sent a `stop()` message, presumably requesting that the item cease its activities.

```
private void stopAll(Collection c) {
        for(Iterator  i  = c.iterator ();  i.hasNext() ;  ) {
                Stoppable s = (Stoppable)i.next();
                s.stop();
        }
}
```

While this syntax performs well and is traditionally considered a good programming practice, exposing an `Iterator` class to the developer requires the developer to couple that class being written to `Iterator`. This can also cause programmer error; a common mistake is to accidentally invoke the `next()` method instead of `hasNext()`—effectively skipping an element at each iteration—or invoking the `next()` method too deep within a nested iteration structure [Sun Microsystems, 2004].

The Java programming language [Gosling et al., 2005] was revised in September 2004 with the introduction of Version 5.0. Among the new features in this version

is a new for-loop syntax[1] that can iterate over container classes without the need to expose a separate `Iterator` or `Enumerator` class[2]. This syntax relegates the responsibility of managing the incrementation and end conditions to the Java language itself, reducing the potential for programmer error. The resulting code also appears less cluttered and easier to understand [Sun Microsystems, 2004].

We can rewrite the above method using Java 5's for-each syntax as follows:

```
private void stopAll(Collection<Stoppable> c) {
        for(Stoppable s : c)
                s.stop();
}
```

By migrating the example to the new for-each syntax, the size of the method has been reduced, the explicit coupling and management of the `Iterator` class has been removed from the source, and there is less likelihood for programmer error.

Although the old collection iteration syntax has not been deprecated or removed, it is reasonable for developers to desire that their source code be updated to use the new syntax. The rewards of such an activity might not seem worth the effort via traditional tools, but adequate software assistance might change that perception. This is particularly true if a software project has a reasonably long expected life-span, as improved modularization assists evolvability [Sullivan et al., 2001], and improving the encapsulation of iteration mechanisms is a form of modularization.

Consider, for example, a developer who is faced with migrating JHotDraw 5.4 (a graphical user interface framework for technical diagrams) to use the new for-

[1] This is sometimes referred to as a *for-each* loop, although the Java keyword remains "for"

[2] This feature is implemented via parameterized classes; the new syntax implicitly uses the iterative functions of the collection class.

each syntax. While this framework was created with best programming practices in mind, it was developed before the for-each iteration syntax was introduced to the Java language.

Collection iteration is traditionally performed in two ways, and JHotDraw uses a mixture of both, depending on the data structure being traversed. The first uses an `Iterator` or `Enumeration` class to traverse the collection. This is normally done in a while-loop, but can be done using a for-loop as well. The second way is to use a numerical index. This is almost always done within a for-loop, and is particularly common when traversing simple array types. Both of these methods can be transformed to the new for-each loop.

Migrating these iterations at any particular location is usually straightforward. The change is localized within each target, and there is a clear set of steps one can perform to change the old syntax into its new form. However, gathering these locations in the first place is not easy.

A simple, common approach to finding the index-based targets would be to perform a lexical search for the keyword "for" using the search-and-replace feature of the developer's IDE. However, performing such a search on JHotDraw returns 3977 results; 3886 of these results are false positives, containing for-loops unrelated to collection iteration. The remaining 91 are true positives, but are not presented by the IDE in any particular order, thereby requiring that the developer investigate all 3997 results manually.

For object-based iterations, the developer could choose to perform a syntactic search for references to the `Iterator` type using syntactic search, but this only returns 78 results and only 38 of those are true positives. 52% of the results are false

positives, and 49% of the locations are false negatives (missed entirely).

2.3 Repetitive Small-Scale Changes

Although individual changes may be small, their repetition becomes problematic. Migrating to a debug framework and updating code to use a new for-loop syntax are both difficult scenarios to address with traditional techniques. A tool to help locate the desired targets could improve the situation.

When a change is introduced into a codebase, it is important to understand what effect that change will have on the existing code [Yau et al., 1993]. If the new change is part of a major unplanned feature or modifies widely used interfaces, substantial restructuring may be required. If, on the other hand, the change involves a localized behaviour, it may be possible to accommodate the change by modifying a single line of code.

The motivational examples presented in the previous section are problematic—at least in part—due to the repetition of the modifications required to complete the change. The unknown extent of the repetition makes locating the appropriate target locations difficult using traditional tools. We term this class of changes as repetitive small-scale changes (RSCs).

It is important to note that not all RSCs are problematic or un-addressable with existing techniques and methods. While the examples presented at the beginning of this chapter were inadequately solved via traditional techniques, there are classifications of RSCs that are well addressed through traditional methods. Renaming a class or data structure within a codebase, for example, could be classified as RSCs,

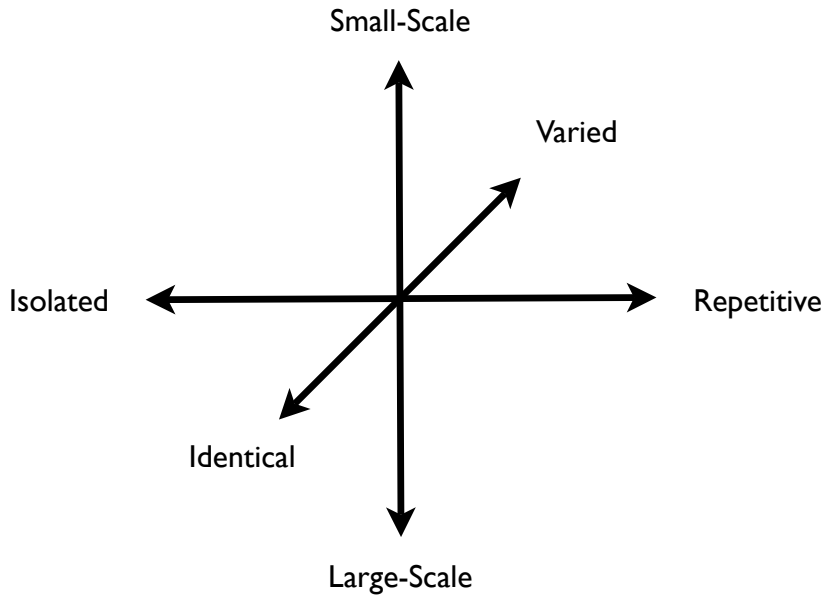

Figure 2.4: A broad range of changes

but can also be applied instantly via lexical search and replace[3]. Even in the cases where search and replace will not properly rename an entity, generalized refactoring tools normally perform this without error.

Some RSCs are more difficult to perform than others via traditional techniques. To understand why, we consider the set of all changes with respect to three axes having a direct impact on the resources required to implement each change. These axes are, respectively, the *repetitiveness* of the change, the *scale* of the change, and the *variance* of the change. Any particular change made to a codebase would fall somewhere inside the three-dimensional space formed by these axes. A diagram of this space is presented in Figure 2.4.

A multi-line lexical search tool spans the plane formed by the repetitiveness and

[3]This, of course, depends on the uniqueness of the name being replaced, but many programmers employ best practices and naming conventions to avoid hard-to-change names.

scale axes, but is restricted on the variance axis to the "identical" end, only finding segments of identical code. Regular expressions increase the variance range slightly, permitting simple lexical variations, but are still relatively limited to the identical side of the variance axis. Regular expressions also limit the range of the scale axis, as large-scale regular expressions become burdensome in complexity. Clone detectors too, are limited to a small range along the "identical" side of the variance axis.

As we plot related work in the problem space, a commonality emerges; traditional techniques that span most of the repetition axis normally require limited variation to deliver the best results. The more varied and repetitive a change becomes, the fewer the techniques that are available to solve it.

We assert that the motivational examples in this chapter are problematic not only because they are repetitive, but also because they have variation. Intuitively, these changes are semantically similar, but lack the lexical or syntactic similarity[4] necessary for traditional approaches to function well.

2.4 Summary

Significant cost may be expended on making changes to existing code, even if the individual changes are trivial to perform. Finding each target location for an RSC is not adequately addressed by the most popular tools designed to locate sections of code. Lexical search and replace and regular expressions both require a lexical pattern to return a match; RSCs, however, may have significant syntactic variation while retaining extremely similar semantics. A syntactic search does consider certain

[4]We discuss the concept of semantic similarity and its consequences to our implementation in Section 5.1

language semantics to eliminate some of the false positives that a lexical search may otherwise return, but lacks a compositional aspect; that is, it still finds occurrences of a lexical string (while paying attention to type) without considering the relationship that string has to the structures around it.

While the repetition of the changes are a fundamental facet of the problem, we determine the variation to be a key part of what makes these particular changes problematic. A tool that addresses RSCs with a semantic similarity, but lexical variance, would help alleviate the problem.

Chapter 3

Detecting Repetitive Small-Scale Change Targets

We propose a technique to detect relevant RSC targets based on a prototypical change. A developer can normally demonstrate an RSC target by changing a known location. Structural and semantic facts extracted from this prototype can help form a query suitable for locating other, potential RSC target locations. If the prototype contains facts related to the change's code and structure, semantic queries may be formed without significant developer input.

In this chapter we explain our technique and describe our prototype tool, Reverb[1]. We begin by presenting a brief usage scenario of Reverb in Section 3.1. In Section 3.2, we describe the data model used to form queries and compare methods. We discuss how queries can be formed from this model in Section 3.3 and discuss how similarity is measured in Section 3.4. The implementation details of the Reverb prototype are discussed in Section 3.5.

3.1 Usage Scenario

Before describing our technique, we present a brief RSC target discovery scenario based on Reverb, our prototype tool. In particular, we consider going about the task described in Section 2.2.

[1]The word *reverb* is the nounal form of *reverberate*, "to have a prolonged or continuing effect; to resound in a succession of echos" [Houghton Mifflin Company, 2004]. We chose this title for its cognitive imagery of quickly repeating an action throughout a defined space.

```
// A JHotDraw method iterating through a collection
protected void basicMoveBy(int dx, int dy) {
                Iterator iter = points();
                while (iter.hasNext()) {
                        ((Point)iter.next()).translate(dx, dy);
                }
        }

// The same JHotDraw method after changing it to use
// the new Java 5 for−loop syntax.
protected void basicMoveBy(int dx, int dy) {
                for(Point p : fPoints)
                        p.translate(dx, dy);
        }
```

Figure 3.1: An example change made to JHotDraw. The developer modifies a code segment iterating over a collection class to use the new Java 5 for-loop syntax.

The developer begins by performing an example small-scale change manually within the Eclipse integrated development environment (IDE) with the Reverb plugin installed. The developer navigates to an iterator-based loop somewhere in the JHotDraw code base and changes it to use the new Java 5 for-loop syntax[2]. An example of this change is presented in Figure 3.1.

The single change is straightforward, but might need to be repeated to complete the RSC. To discover locations that may require a similar change, the developer invokes Reverb by clicking the search icon in the toolbar or right-clicking on the method and selecting the "Find Similar" option (Figure 3.2). This extracts information from the example change and launches the query process.

After the query has executed, the Results View (Figure 3.3) appears in the Eclipse

[2]We assume that the relevant container classes have already been updated to use Java's `Iteratable<E>` interface.

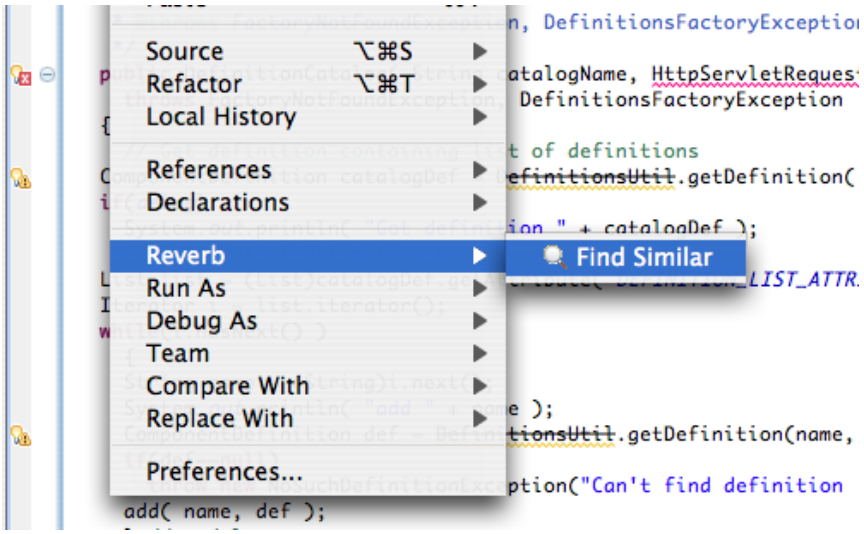

Figure 3.2: Invoking reverb

workspace.

The Results View contains a list of methods that were found in the source project, sorted by their similarity to the sample change. Similarity is indicated by a metric calculated during the query process, and Reverb places a check-mark beside the most highly ranked results[3]. From our perspective, highly ranked methods are locations in the source code that likely require the same change, according to the query.

As the developer browses the list of returned methods, he can double-click any of them to open a new Java editor in the Eclipse workspace containing that method. As relevant methods are browsed, the developer can perform the necessary changes in the editor and continue browsing through the list. The process stops once we are satisfied that the RSC is complete, usually indicated by advancing into poorly-ranked results, or encountering a noticeable section of false hits.

[3]In our prototype implementation, the threshold similarity metric used to designate a check-mark is hard-coded, and not derived from any statistical analysis of the results. We discuss this more in Chapter 5

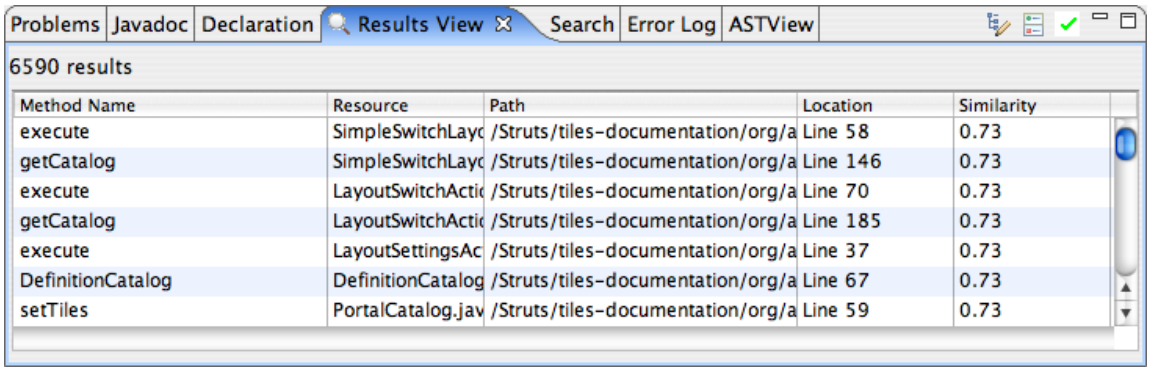

Figure 3.3: The Reverb Results View. Each item listed represents a method in the project. The methods are presented in order of their similarity to the change that was just made.

3.2 A Model for Code Comparison

To analyze similarity between two code segments, we require comparable information from each. Information about the code structure, dataflow, control-flow, references, and other entities can be pre-processed and included as "facts" about any particular segment. This abstract representation of the code can be tested for the existence of individual facts and establish a means to detect and compare defined changes.

Determining what constitutes a "code segment" depends on the goals of the comparison. Since we are dealing with changes on the small-scale and would like to extract the changes automatically, we define a code segment as a single method. While adopting a limited scope affects the sort of changes that can be compared (we discuss these in Chapter 5), methods have well-defined boundaries and interfaces that facilitate real-time analysis.

Facts can be extracted from methods via the project's abstract syntax tree (AST). An unoptimised parse tree, in particular, is verbose enough to contain all informa-

Fact Name	Fact Description
Reference	A reference to some data
Method	A Java method or operation
MethodInvocation	An invocation of some method or operation
CFGNode	A node in the control flow graph

Table 3.1: The four basic fact types used by our technique.

tion[4] available about the original code in a format that can be searched more easily than the original source text.

For our model, the information extracted from each method is comprised of three major entities: *facts*, which I discuss in Section 3.2.1; *attributes*, discussed in Section 3.2.2; and *associations*, presented in Section 3.2.3. Together, we refer to this information as a factbase.

3.2.1 Facts

The information of greatest importance to our technique is contained within simple facts extracted from a method. A fact itself is simply a declaration that something exists within a particular method. This may include references, nodes in the control flow graph, method invocations, and the method signature itself. Table 3.1 describes each of these fact types.

The fact types listed in the above table are not intended to represent all the information that could be available in a method. Rather it indicates the existence of high-level structures often related to RSCs. These facts can appear in any number from 0 to many in association with a method, with the exception of the `Method` fact

[4]The AST used by the Eclipse Framework can be used to output the original code, for example.

```
public void SampleMethod(String s) {

        if( s.equals("foo") )
                s = "Foo to you too.";
        else if( s.equals("bar") )
                s = "Bar none.";
        else
                s = "What?";

        System.out.println("s has been re-assigned to: ");
        System.out.print(s);

}
```

Figure 3.4: An example Java method.

type which occurs exactly once. That is, several references, control flow graph nodes, and method invocations can occur inside a method, but declarative information about the method itself can only be stored once. Operators (such as `+`, `++`, <=, etc) are also treated as `MethodInvocations`.

To better understand the high-level information extracted into facts, compare the Java method presented in Figure 3.4 to the facts extracted from that method in Figure 3.5. As seen in these two figures, the extracted facts are declarations of the existence of high-level elements.

3.2.2 Attribute Attachment

Declaring the existence of certain fact types is not enough. Not only do methods often contain more than one of the same type of fact, but we may envision scenarios where it is important to know information attributed to each fact. For example, we may wish to know the return type of a method invocation, the name of a variable, or

Fact Type	Description
Method	The `SampleMethod` signature
Reference	The `s` parameter
Reference	The `"foo"` literal
Reference	The `"foo to you too."` literal
Reference	The `"bar"` literal
Reference	The `"Bar none."` literal
Reference	The `"What?"` literal
Reference	The `System.out` external reference
Reference	The `"s has been re-assigned to:  "` literal
MethodInvocation	The first `equals` message
MethodInvocation	The second `equals` message
MethodInvocation	The `println` message
MethodInvocation	The `print` message
CFGNode	The `if` condition and block
CFGNode	The `else if` and `else` blocks
CFGNode	The `else if` condition and block
CFGNode	The `else` block

Figure 3.5: Types of facts extracted from the code sample in Figure 3.4

whether a reference is a literal. In order to perform more complex queries involving these types of information, extra knowledge beyond the mere existence of facts is required.

We provide this information through the use of attributes. A fact attribute is an additional bit of information about any particular fact. An arbitrary number of these attributes may be attached to a given fact during the fact extraction process, and a query is welcome to place importance on or ignore any number of the associated attributes. The attributes themselves are represented by simple key-value pairs, where the key is a pre-defined description of the attribute type, and the value is any comparable data type.

A list of the attributes used in our prototype implementation, along with their associated fact types and a brief description of the values each takes, is presented in Table 3.2. Figure 3.6 shows attributes extracted from the code sample in Figure 3.4 that have been added to the appropriate facts. The attribute definitions in this figure are also those used by the Reverb implementation, though none are explicitly required by our proposed approach or specifically affect the similarity ranking process. In the development of Reverb, attributes were added, removed, and renamed without having to change the similarity calculation algorithms, for example.

3.2.3 Fact Associations

In addition to facts and attributes, we need a way to relate facts to each other. Information about where facts occur in the code's structure, and how data flows between references may be required to more accurately detect similarities. Our technique accomplishes this through associations. An association is a labeled directed

Attribute Name	**Fact Type**	**Description**	**Possible Values**
isType	All	The type of fact	catchClause, forLoop, whileLoop, forEachLoop, ifTrue, ifFalse, try, *[any resolved data type]*
hasName	Method, Reference, MethodInvocation	The assigned name or operator name	*[any Java name]*
isConstructor	Method	Whether or not a method fact is a constructor	true, false
returnsType	MethodInvocation, Method	The return type	*[any resolved data type]*
isA	Reference	Reference designation	literal, returnValue, variable
isLoop	CFGNode	Whether or not the CFGNode is a loop	true, false
hasValue	Reference	The value of a literal	*[Any numerical, boolean, character, or string value]*
isParameter	Reference	Whether or not a reference is a method's parameter	true, false

Table 3.2: Attributes used by the Reverb implementation.

Method :

hasName SampleMethod

Reference :

hasName s
isType String
isParameter true

CFGNode :

isType ifTrue

CFGNode :

isType ifFalse

MethodInvocation :

returnsType boolean
hasName equals

Reference :

isType java.lang.String
isA literal
hasValue foo

Reference :

isType java.lang.String
isA literal
hasValue Foo to you too.

CFGNode :

isType ifTrue

CFGNode :

isType ifFalse

MethodInvocation :

returnsType boolean
hasName equals

Reference :

isType java.lang.String
isA literal
hasValue bar

Reference :

isType java.lang.String
isA literal
hasValue Bar none

Reference :

isType java.lang.String
isA literal
hasValue What?

MethodInvocation :

returnsType void
hasName println

Reference :

isType java.lang.String
isA literal
hasValue S has been re-assigned to:

MethodInvocation :

returnsType void
hasName print

Figure 3.6: Attributes extracted from the method in Figure 3.4.

edge between two facts that establishes a relationship. These relationships can store information about dataflow among references, the control flow graph structure, or other important associative properties.

Like attributes, associations can be defined on an implementation basis. Adding or changing associations in different implementations of our technique does not affect the core calculation process. However, there are certain relationships that we believe are fundamental to the discovery of relevant RSC target locations. These are dataflow information between references, and presence within the control flow graph. We discuss them in turn.

Dataflow associations An important similarity between two syntactically different code segments could be the overall flow of data in their execution. For example, the two code segments presented in Figure 3.7 differ primarily in the syntax of how information is passed from one reference to another. While the second code segment lacks the additional reference declaration, data still flows from the method parameters to the attributes of the `println` method. By pre-processing the potential dataflow paths and including them as associations between facts, some false negatives can be avoided.

The `dataFlowsFrom` association helps identify the origin and flow of data. It is possible for any single reference to be the target of multiple data flow paths. Whether or not the data actually flows to a particular reference in execution (due to conditional branches or user input, for example) is not relevant to the calculation. If dataflow information is selected as being important for the query, then it still needs to be addressed during the RSC process, regardless of the likelihood of it occurring

```
// Code Sample A
public void sampleA(String s1, String s2) {
        String temp = s1 + s2;
        System.out.println(temp);
}

// Code Sample B
public void sampleB(String s1, String s2) {
        System.out.println(s1 + s2);
}
```

Figure 3.7: The origin of the data passed to the output statement is the same in these two code segments.

during execution. Additionally, the dataflow target of a literal or parameter should be itself, in addition to whatever other assignments that reference takes.

Control flow graph structure A second association of importance is the existence of any particular fact within the control flow graph. This may include other control flow graph nodes, in addition to the other non-`Method` fact types. For example, the motivational example presented in Section 2.1 involved a search for output statements guarded by a debug condition. If the output statement is not contained within a branch testing for the right condition, then it should not be considered a potential RSC target. The `inCFG` association helps distinguish such scenarios.

With the exception of dataflow and control flow relationships, association definitions are not specific to our technique, and any can be added on an implementation-by-implementation basis without affecting the similarity calculation process. During the development of Reverb, for example, relationships were added and renamed without consequence to the similarity metrics.

Table 3.3 provides a list of the associations used in our prototype implementation

Association Name	Description	Origin	Destination
hasParameter	Specifies that a reference is a parameter to a method invocation or operation, or a method signature	MethodInvocation, Method	Reference
inCFG	Specifies that a destination fact is inside the specified CFG node	Reference, MethodInvocation, CFGNode	CFGNode
dataFlowsFrom	Specifies that data flows from one fact to another	Reference	Reference, MethodInvocation
invokedOn	Specifies that a message is sent to an object reference	MethodInvocation	Reference

Table 3.3: Associations used by the Reverb implementation

along with a description. Figure 3.8 shows the associations extracted from the code sample in Figure 3.4.

3.3 Query Formation

We would like to discover RSC target locations based on an example change, not a full profile extracted from a method. While the data model discussed in Section 3.2 allows us to form a high-level, comparable representation of methods, we selectively match data based on a prototypical change.

By collecting facts about what has changed, information relevant to locating targets that require the same or a similar change may be extracted. In particular, information that is missing or changed from the "before" snapshot in the "after" snapshot is likely of interest.

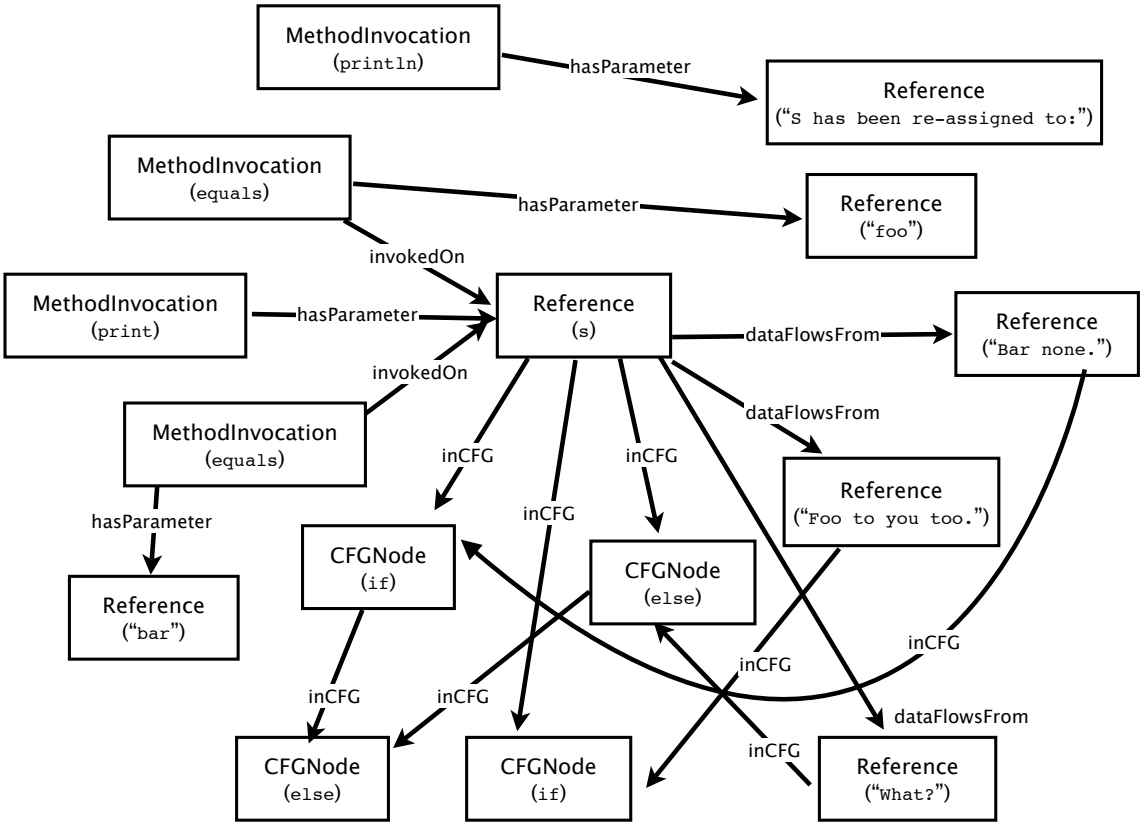

Figure 3.8: Associations extracted from the code sample in Figure 3.4

Our technique forms queries based on set arithmetic. The factbase extracted from the "before" snapshot can be compared to that of the "after" snapshot; facts that are missing or have changed attributes in the "after" snapshot are added to the query, and facts that remain the same or are new to the "after" snapshot are ignored. Note that it is possible for set arithmetic to select irrelevant facts, depending on the nature of the change being queried. For scenarios in which this arithmetic forms an inappropriate query, some user intervention in the form of query refinement may be necessary. Query refinement is discussed in Section 3.5.2.

To better explain the query formation step, we introduce a notation representative of our factbases and their transformations:

We represent the "before" and "after" snapshots of the demonstrated example source as B_{source} and A_{source} respectively. While querying the source code, we represent the method being considered for similarity comparison—essentially a second "before" snapshot—with B_{target}. The transformation applied by the developer can be considered a second "after" snapshot, A_{target}.

The process by which a factbase is extracted from each of the snapshots is represented by the function $Factbase(m)$, where m is any method snapshot, and the result is the factbase extracted from m.

Once two factbases are extracted, we can compute the difference between them with the function $\mathit{Diff}(fb_1, fb_2)$, where fb_1 and fb_2 are both method snapshots. The result from the *Diff* function is a new factbase, the set of modifications and deletions between the two factbases fb_1 and fb_2.

Our query set is then represented as follows:

$Queryset = \mathit{Diff}(Factbase(B_{source}), Factbase(A_{source}))$

Once a query consisting of important facts has been derived, a means to rank similarity based on that query is required.

3.4 Ranking Similarity

Similarity assignment and ranking is the most important step in the RSC target location process. From the developer's perspective, this is also how relevant locations are defined and presented after an example change has been provided and the query has completed. A method is said to be similar to another if a subset of facts, attributes, and associations is closely matched to the other. The more of these facts that are matched, the higher the similarity.

For our technique, similarity is represented as a real number in the range from 0 to 1, where 1 represents complete similarity based on the provided query, and 0 represents complete dissimilarity. It should be noted that these rankings are based on the facts deemed important by the query; two methods that are ranked as completely similar are not necessary identical, but may contain all the facts that are expected by the query.

To compute the similarity value, we first define a function, $Bestmatch(f, fb_1, fb_2)$, which determines the best possible match within the factbase fb_1 for the fact f, with respect to the facts in fb_2.

Our query may then test for the existence of facts, along with their attributes and associations. If the facts exist, the similarity ranking is strengthened; if they do not exist, the similarity is weakened. Overall, the ranking is expressed as a percentage of the important facts, attributes, and relationships that a particular

method contains with respect to the query. In our notation, we represent this as the function $sim(Queryset, Factbase(B_{target}))$, defined as follows:

$$\frac{\sum_{f \in Queryset} Factsim(f, Bestmatch(f, Factbase(B_{target}), Queryset))}{|Queryset|}$$

where $Factsim$ is a function determining a similarity metric in the range $0 \leq Factsim(f_1, f_2) \leq 1$, for any individual facts f_1 and f_2. If the attributes and associations of a fact, f, are represented by $att(f)$, then we define $Factsim$ as follows:

$$Factsim(f_1, f_2) = \frac{|att(f_1) \bigcap att(f_2)|}{|att(f_1)|}$$

Method snapshots may contain multiple differences, so discovering a correspondence is necessary to form a query. In particular, we must infer whether any particular fact in the "after" snapshot is a new fact, a fact that has been changed from the "before" snapshot, or one that is completely identical to the original. This inference is inherently imprecise; however, we can attempt to find the best-fitting matches. We discuss the matching process of the factbase data types and describe their implementation in turn.

Fact matching Any given fact in method A can be said to "match" a fact in a method B if A's fact is of the same type as B's and the fact in B contains more of A's attributes and associations than any other fact in B. The "closeness" of the match is expressed as a ratio of the number of matched and unmatched attributes and associations, $Factsim$.

In our notation, we say that a fact $f_1 \in Factbase(B_{target})$, matches another fact, $f_2 \in Queryset$, if the following is true:

$$\neg\exists\, f_3 \in Queryset \mid Factsim(f_1, f_3) > Factsim(f_1, f_2)$$

Attribute matching An attribute, on the other hand, is said to either match or fail to match with no values in between. This is done by testing for equivalence on both the name and value of the attribute. If either of these comparisons fail, the match is rejected.

In our notation, we say that an attribute a_1 matches another attribute, a_2, only when $a_1 = a_2$.

Association matching As with attributes, associations either match or fail to match completely. Unlike attributes, whether or not an association matches another depends on another fact—the target of that association. For example, if a particular Reference has a `dataFlowsFrom` association, we must determine whether the target of that association is similar enough between the "before" and "after" snapshots to consider it to be the same relationship. Since associations may form a cyclical graph, they cannot be followed forever to determine this.

Our prototype implementation overcomes infinite recursion during the association matching process by calculating a hash value from the association's target, and comparing it to the hashes of its potential matches. The hash value is comprised of common attributes that should give a reasonable indication of whether or not the associated fact is a match. Specifically, we look at the type of the fact, its name, its data type (if applicable), and its value. While imperfect, this approximation was made to improve performance and simplify the debugging process.

An improved implementation would recursively calculate a similarity metric, following associations only a few nodes deep to avoid infinite recursion. Interim metrics

could be cached to avoid needless re-calculation, and any association that passes a pre-defined similarity threshold[5], could be considered a match. If the association's similarity does not meet the threshold, it would be declared a mismatch.

The original implementation of the Reverb prototype was designed to use the limited recursion approach; however, without caching previously calculated similarities, performance was sluggish. Additionally, debugging the similarity calculation process became overly burdensome and the approximation approach was adopted for our prototype.

In short, we say that an association as_1 matches another association as_2 if $Factsim(target(as_1), target(as_2)) \geq threshold$, where $threshold$ is some similarity threshold, and $target(as)$ is the fact target of the one-way association as.

Once we have defined how facts, attributes, and associations match each other between versions, we need to correlate the two as accurately as possible. Correctly identifying and resolving what has changed between two factbases is not always possible [Berzins, 1986]; however, a best fit can be found for any two sets. We do this via a "bidding" system in which each fact analyses its candidate matches and sorts them based on the closest match. If a fact's preferred candidate is better matched to another fact, the next choice is considered until all matches have been made. Orphaned facts are considered additions or deletions depending on which factbase contains them. All others are considered either identical (i.e., a perfect match) or a modification (an imperfect, but closest match). A pseudocode algorithm is presented in Figure 3.9.

[5]Empirical experimentation would be helpful in defining an appropriate value for such a threshold.

```
for all b | b ∈ bidders do
  m ← best match for b in candidates
  bids ← bids + m
end for
for all c | c ∈ candidates do
  w ← maximum of bids
  if ¬∃ w then
    c.status ← DELETION
  else
    if ∃ a | a ∋ (c.attributes ∩ w.attributes) ∪ (c.associations ∩ w.associations)
    then
      c.status ← MODIFIED
    else
      c.status ← MATCH
    end if
    candidates ← candidates − c
    bids ← empty set
    for l | l ≠ w + l ∈ candidates do
      m ← best match for b in candidates
      bids ← bids + m
    end for
  end if
end for
```

Figure 3.9: A pseudocode algorithm for discovering additions, deletions, and modifications between two factbases.

Once we have information on which facts are new, changed, and deleted, simple set arithmetic can be used to determine which should be added to the query.

3.5 Prototype Implementation

To better understand the benefits and drawbacks of our technique for the purposes of evaluation, we have created Reverb, a prototype implementation. The usage scenario presented in Section 3.1 provides an example of how our tool is used from the developer's perspective. We now discuss the implementation of Reverb in more detail.

We have implemented Reverb as an Eclipse[6] plug-in capable of working on programs written using the Java language. Although the technique for locating RSCs is language independent, we chose to work with Java and the Eclipse environment because of their robust toolsets and personal familiarity with each.

At an abstract level, Reverb can be seen as an entity derived from the Observer design pattern [Gamma et al., 1993]. Reverb observes changes to the source code as they are made, and upon request, can present a list of recommended locations based on those observations.

While a constant observation is necessary for Reverb to function, the observations are limited in scope. The recommendation process is where the bulk of the code analysis is done, and the architecture of the plugin is organized around this idea.

Since Reverb is an Eclipse plugin, some of its architectural decisions were influenced by the Eclipse framework; however, unlike most other plugins designed for

[6]Eclipse is an open source integrated development environment written in Java

Eclipse, Reverb is activated immediately after the IDE has initialized. Most other plugins wait for a user-initiated "lazy" activation, but Reverb needs to begin observing changes as soon as the workspace is ready.

Observing Changes

As we navigate through a Java project, making additions and changes, Eclipse has knowledge of the active Java file and caret position at all times. When the user moves the caret within the text environment, either by typing or making a new selection, Eclipse reports that change to any registered observer, via the Observer design pattern [Gamma et al., 1993]. Reverb registers itself as such an observer, receiving reports of any text selection changes.

When a selection change is reported, Reverb checks to determine in which method the text caret is contained. If the developer has moved the caret into a method, then a snapshot of that method is saved to memory in the form of a subtree of Eclipse's parse tree. Since developers may be working on several classes at the same time and task-switch between them, several snapshots are tracked at the same time. We could, for example, enter a new method M1 in class A, begin making some changes, switch to class B and edit a different method M2, then re-enter class A and resume editing M1. In such a situation, Reverb would only save one initial snapshot for M1, which is the first time we entered it. A replacement snapshot would not be taken unless we were to begin editing another method within the same class.

The source code may or may not be syntactically correct during active development. For this reason, Reverb opts to ignore selection changes that cannot be determined to be contained inside a method and fundamental changes to a method's

signature are not considered as intra-method changes. While work exists to detect changes to identifiers and method signatures [Malpohl et al., 2000; Kim et al., 2005], an analysis based on line numbers could also overcome these limitations, due to the immediacy of the change.

3.5.1 Fact Selection

Facts are extracted from partial ASTs that were either saved during the change observation process described in Section 3.5 or during the query process described above. The extraction is done using the Visitor design pattern [Gamma et al., 1993]. As the extractor visits each node in the partial AST, it takes note of information deemed to be of interest, expressed as the facts, attributes, and associations described in Section 3.2. The fact types used by Reverb are identical to those listed in Table 3.1.

As Reverb extracts the facts from a method, it assigns a unique identifier to each one. The identifier is assembled using a combination of the fact type and the order it was encountered. It is used internally to help compare and distinguish facts, but is also presented to the user in the Factbase Comparison dialog and Query Refiner dialog (see Section 3.5.2).

In our implementation, fact associations are represented as a standard attribute which has another fact as its value. It should be noted however that, during the similarity ranking process, Reverb treats these attributes as the abstract associations described in Section 3.2.

3.5.2 Manual Query Refinement

At some point, we may encounter a result that does not exactly fit the RSC we are trying to address. In this situation, a few concepts for improving and understanding the results are available. First, we can view a rationale of why the result was chosen. This is done by clicking the factbase comparison icon in the Results View, which opens up the Factbase Comparison window (see Figure 3.10). The window contains a tree structure containing the query's Facts and Attributes present in that particular method[7]. If there is a green dot beside a particular fact, attribute, or association, then an important fact was found in the method. If there is a red dot, the associated fact, attribute, or association was designated as important, but was not found. If there is a grey dot, that means that particular fact, attribute, or association is considered unimportant from the perspective of the query, and ignored in the calculation.

There is a possibility that we may disagree with the facts that were determined to be important during the query formation. In such a situation, we may manually specify which facts are most important to the task at hand. This is done through the Refine Query Dialog (see Figure 3.11), which appears after clicking the Refine Query icon on the Results View. This dialog shows every fact, association, and attribute that was extracted from the original version of the changed method. Each element is written beside a checkbox that can be clicked to toggle its important/unimportant state. By toggling the importance values, we are able to specify which information is used in the similarity calculation process. Once the dialog is committed, the project code will be re-searched and the Results View will be updated with the newly ranked locations.

[7]We sometimes refer to these as "important" facts.

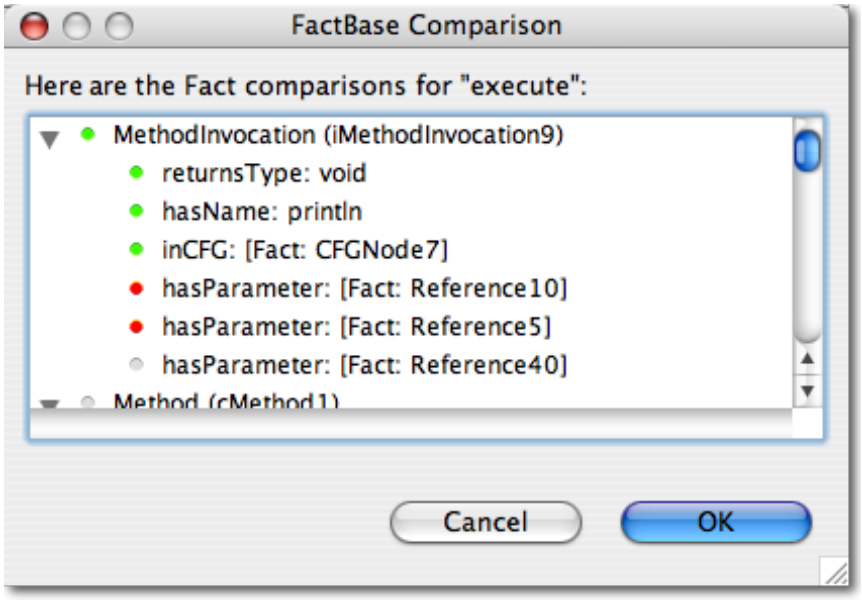

Figure 3.10: The Factbase Comparison window. This allows the developer to see the rationale for a particular method's ranking in the results. The facts, attributes, and associations shown in this diagram are explained in greater detail in Section 3.2.

3.6 Summary

To query for similar, but non-identical methods, we use a factbase data model based on facts, attributes, and associations. Fact represent declarations of the existence of one of four types: methods, references, method invocations, and control flow graph nodes. Attributes represent data associated with each of the fact types and can occur in any quantity. Associations are one-way relationships between facts.

A query can be formed by selecting the facts that have been deleted or modified from an AST snapshot taken before and after a modification. A similarity metric is calculated based on the ratio of query facts present in any particular fact base. We search for similar locations based on this prototypical change by calculating the similarity metric for all methods in the project and sorting them based on that

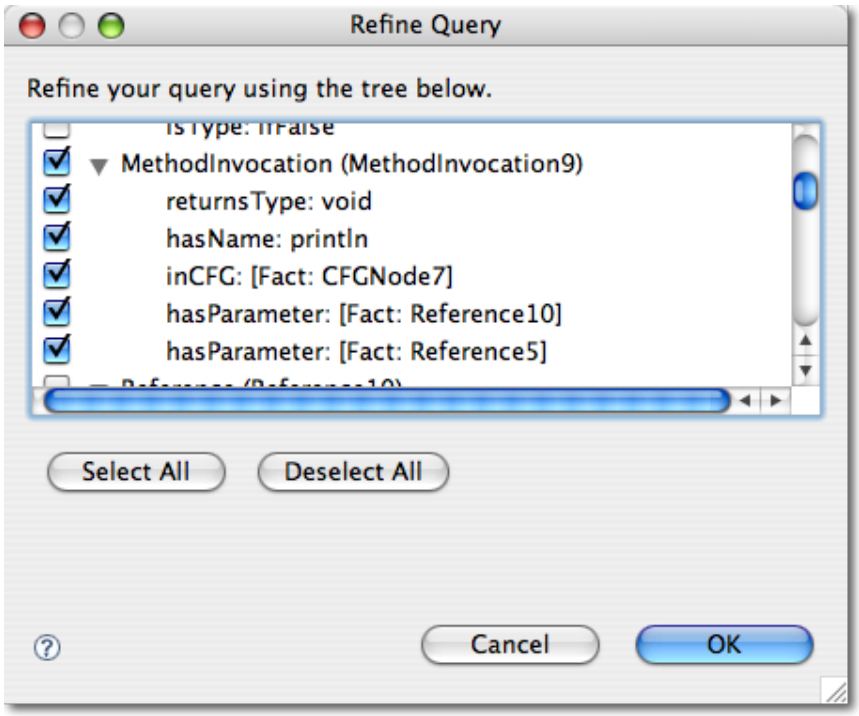

Figure 3.11: The Refine Query dialog. This allows the developer to modify which information is deemed important when querying other methods in the project. A checkmark beside a particular fact, attribute, or association implies that it is important.

metric. We've implemented the technique in an Eclipse plug-in called Reverb.

Chapter 4

Evaluation

To examine the effectiveness of Reverb, we must develop a means by which the results can be compared to other approaches. We evaluate performance primarily on the precision and recall of the results.

The focus of Reverb is to return relevant target locations for an RSC, so we compare the returned locations to those of the actual RSC targets. We evaluate the accuracy of our approach in locating RSCs performed on two development scenarios involving popular, open-source software systems. In the first scenario (Section 4.1), we attempt to evolve Apache Struts 1.1, a web application framework, to consistently use a logging framework instead of guarded print statements. In the second scenario (Section 4.2), we attempt to evolve JHotDraw 5.4, a diagramming tool and framework, to use Java 5's newly introduced syntax for iterating through collections. We attempt to complete each of these scenarios using four approaches: lexical search, syntactic search (using Eclipse's built-in search features), a clone detector, and the heuristic search technique proposed in this paper. The results are presented in their respective sections.

To evaluate the performance of clone detection in supporting RSCs, we used CCFinderX [Kamiya et al., 2002]. CCFinderX is a hybrid approach to clone detection using token-based comparisons and language-specific filtering. The tool was selected for its availability and optimized performance.

4.1 Moving to a Logger Framework

Apache Struts is a popular open-source framework for creating web applications in Java. While browsing the source files, it was discovered that simple "debug" flags, as described in Section 2, are being utilized in parts of the source distribution. It is unknown to what extent these flags are used, but since a logging framework is leveraged elsewhere in the codebase, we wish to evolve the source, eliminating all "debug" flags in favour of that framework.

Lexical Search Results Since the debug output that we are interested in finding must occur within a conditional test for the "debug" flag, we performed a lexical search on the word "debug" and manually inspected each of the reported results.[1] 234 results were returned, 20 of which we deemed to be true positives. These 20 results occurred within 6 methods.

We describe these results more concisely by their *precision* and *recall*. The formulas for calculating these metrics are shown below:

$$\text{precision} = \frac{\text{true-positives}}{\text{total-positives}}$$
$$\text{recall} = \frac{\text{true-positives}}{\text{total-actual}}$$

For the lexical search, this corresponds to a recall of 100%, but a precision of only 9%.

[1] We opted to search for the term "debug" as opposed to "if (debug)" because the former is a common term between both, and we wished to find relevant locations regardless of bracket placement and whitespace.

Syntactic Search Results To perform a syntactic search, we selected an occurrence of the "debug" flag and instructed the Eclipse IDE to search for other locations where this same flag is referenced. Eclipse returned 3 results, and only 2 of them were legitimate RSC target locations. This corresponds to a recall of 10%, and a precision of 66% relative to the lexical results. The poor syntactic search results can be explained by Struts's use of fields instead of a shared, globally available variable to define the debug flag.

Clone Detector Results To obtain results from CCFinderX, we instructed the tool to find code clones within the Apache Struts project and navigated to a location that we knew to contain a relevant guarded print statement. Two clones were associated with our known location, and both were indeed relevant to our RSC. This corresponds to a precision of 100%, but a recall of 10%.

Since we are also interested in potentially helpful clones that were not detected as being associated with our known location, we also investigated the performance relative to the complete set of 489 detected clones. Of these, 2 clone sets were related to our RSC, representing 4 of the the actual change target locations. This yields a precision of 0.004% with a recall of 25%.

Reverb Results We invoked Reverb on an example change made to one of the RSC target locations. Specifically, we changed one of the two guarded output statements located in the `DefinitionCatalog` constructor. After the invocation, the Results View opened, showing all methods in the Apache Struts project, sorted by their similarity metrics to the example change. We did not launch the Query Refiner to adjust the default query.

Unlike the previous techniques, Reverb returns a list of all methods, sorted by their similarity to the situation at hand. Since a single precision and recall metric cannot express this, we present a truncated graph of Reverb's precision and recall metrics as a function of the number of results returned in the order they were presented (Figure 4.1). The formulas used for these metrics are as follows:

$$\begin{aligned} \text{precision}(k) &= \frac{\text{true-positives}(k)}{k} \\ \text{recall}(k) &= \frac{\text{true-positives}(k)}{\text{total-actual}} \end{aligned}$$

The variable k in these equations represents the ordered set of results. To graph these formulas, starting at the first result, we increment k until the entire set is exhausted. This graphing technique represents how the developer is intended to view the results. One starts at the first result and works steadily through the rest until the frequency of false positives becomes too great. When studying the graphs of Reverb's result, we are looking for a large set of true positives clustered at the front of the order.

The graph of a best-case scenario would be an uninterrupted cluster of true-positives from 1 to n, and false-positives from $n+1$ to m, the last result. An occasional false positive interspersed through the relevant results is still considered very good. The worst-case scenario would be an uninterrupted cluster of false-positives from 1 to n, and true-positives from $n+1$ to m. A random order of true- and false-positives would be almost as bad.

As shown in Figure 4.1, Reverb returned perfect results for this example. The

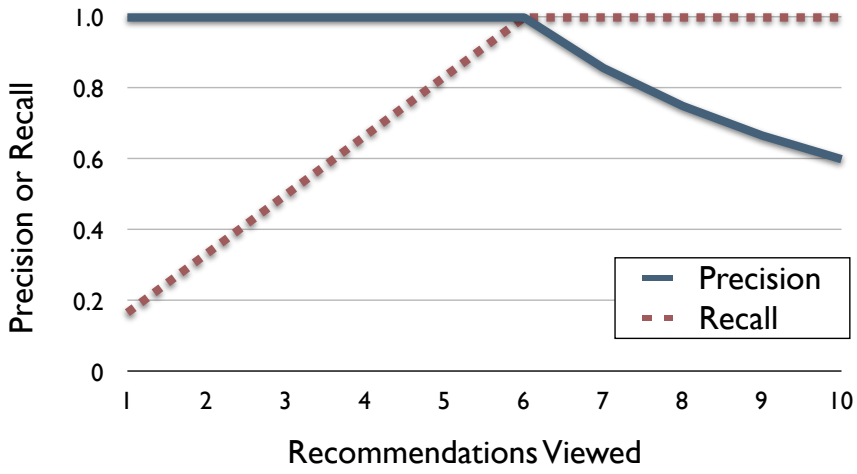

Figure 4.1: Truncated results for the debug example

recall percentage increases with a precision of 100%, until all results have been returned. The precision then drops exponentially as k increments to include all of the 6590 methods, most of which contain negligible similarity metrics. These results are excellent, as they match our best-case scenario.

To better understand Reverb's precision and recall for this example, we also graph these figures with respect to the similarity metrics assigned by Reverb (Figure 4.2). The graph results show that, even though the order of the results were perfect, there were still a few false positives ranked with the same similarity metric. A graph of the precision versus the recall (Figure 4.3) also reveals the effect of the false positives, as a perfect graph would remain a horizontal line at the 1.0 value without dipping at the end.

These results are substantially better than the traditional approaches evaluated. To demonstrate this more clearly, we graph the precision and recall of the lexical

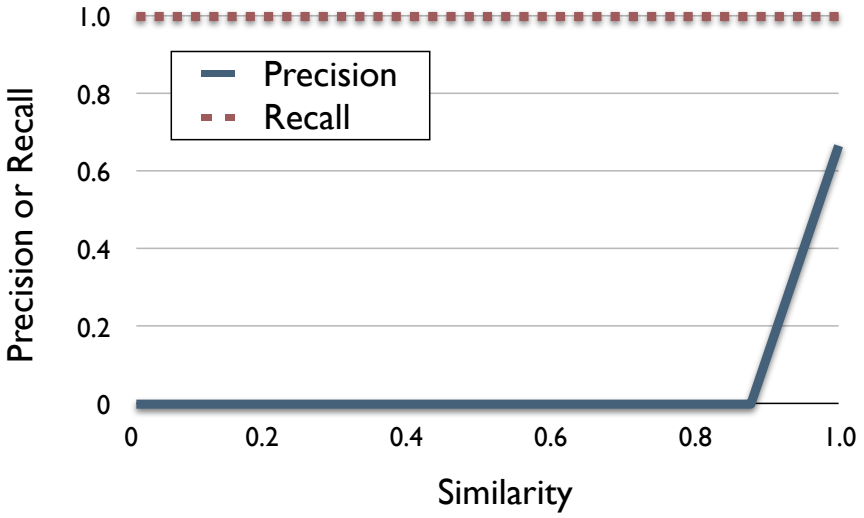

Figure 4.2: Precision and recall versus Similarity for Debug Example

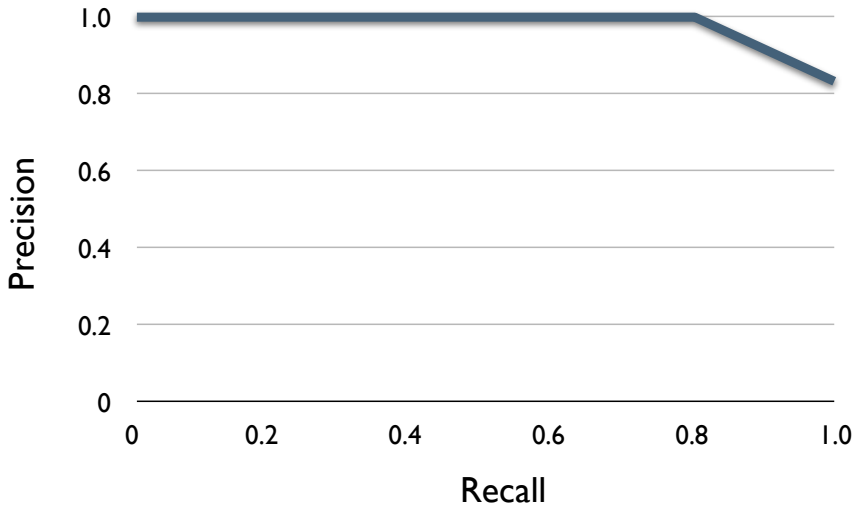

Figure 4.3: Precision versus recall for the debug example

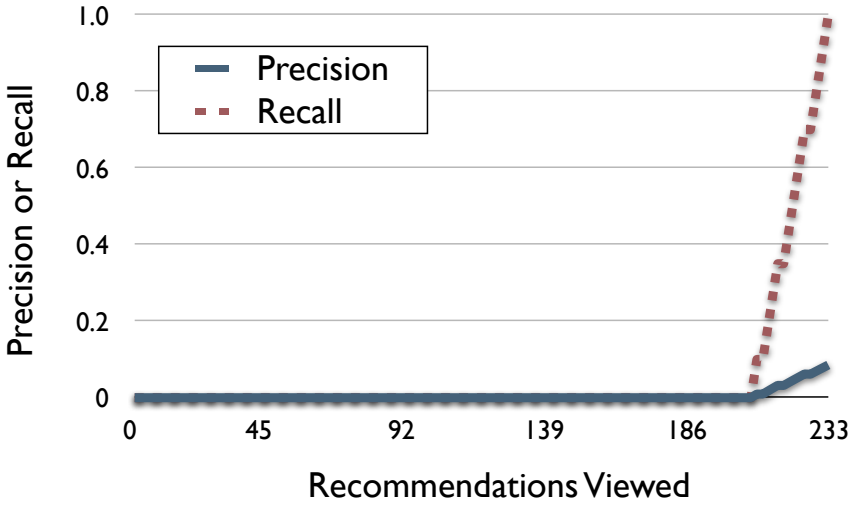

Figure 4.4: Precision and recall for Lexical Search

results[2] as a function of the order the results were returned (Figure 4.4). Comparing this to Reverb's results (Figure 4.1), we see that the lexical search returned almost a worst-case scenario. Since lexical search results are not returned in any meaningful order (Eclipse sorts them alphabetically based on the filename in which they occur, and this case had most hits in a low alphabetical file), it is important to rank results based on their relevance to the task at hand.

4.2 Migrating to For-Each Loops

Figure 4.5 shows how a collection iteration in the `basicMoveBy(...)` method of JHotDraw's `PolyLineFigure` class could, potentially, be updated to take advantage of the new for-each syntax.

[2]We did not repeat this additional comparison for the syntactic search and clone detector results because the former focuses on returning a few, exact results, and the latter is presented in a two dimensional scatter plot, without a concept of order.

```
public void basicMoveBy(int dx, int dy){
        Iterator  iter  = points();
        while(iter.hasNext()) {
                ((Point)iter.next()).
                translate(dx,dy);
        }
}
```

```
public void basicMoveBy(int dx, int dy){
        for(Point p : points())
                p.translate(dx,dy)
}
```

Figure 4.5: Possible collection iteration migration in JHotDraw

Like our previous example, once initial preparations are made, updating any single loop to use the new "for each" syntax is relatively well-defined. However, this change must be repeated throughout the codebase in order to complete the evolutionary step.

We wanted to locate and transform all possible collection iterations. With this in mind, we split the task into two queries: finding RSC target locations for object-based loop traversals (using iterator, enumeration, or other traversal classes), and finding RSC target locations for integer-based index loop traversals. We evaluated these two queries with respect to the same approaches as before.

Lexical Search Results To search for object-based traversals, we decided to search for the keyword "while" and manually investigate each of the results to determine which were true positives. This search fared well, returning 190 matches, with 138 of them representing legitimate RSC target locations at the method granularity.

For the subset of migratable iteration locations utilizing object-based traversals, this represents a recall of 98% with a precision of 75%.

For our second query, we searched for the keyword "for", since there is no obvious lexical element that connects all index-based iterations. Eclipse returned 3977 results, but only 91 of these occurred within legitimate RSC target methods. For the subset of migratable iteration locations utilizing integer-based index loops, this represents a recall rate of 100%, but a precision of only 2%.

Syntactic Search Results To use a syntactic search to find object-based traversals, we opted to search for accesses to references of the type `Iterator`. Eclipse returned 78 results within a total of 73 methods. 38 of these methods contained legitimate migratable RSC targets. This corresponds to a recall of 51% and a precision of 52%.

The second query did not lend itself well to a syntactic search. The keyword "for" is the most common lexical similarity when using an integer-based index traversal, but the Eclipse syntactic search does not allow the specification of keywords. Since there was no particular field, method name, or other non-keyword string suitable for performing this query, we determined this query to be too complex for a syntactic search and opted to forego this evaluation step.

Clone Detector Results Using CCFinderX, we processed the JHotDraw codebase for clones and navigated to a point known to contain an object-based traversal. A single clone was detected for our known location, and it was a relevant change location. This corresponds to a recall of 1% and a precision of 100%.

As using a seed location did not result in adequate performance, we also investi-

gated the performance relative to the complete set of 1123 detected clones. Of these, 13 clone sets were related to our RSC, representing 25 of the actual change target locations. This corresponds to a recall of 50% and a precision of 2%.

Using the same technique to search for index-based traversals, CCFinder did not find any clones for our known location. However, of the 1123 detected clones, we found 38 clone sets related to our RSC representing 91 of the actual change target locations. This yielded a recall of 65% and a precision of 3%.

Reverb Results We started this scenario, as in the previous example, by invoking Reverb on an example change. We transformed, at random, a location within JHotDraw that uses an `Iterator` class and a while loop to traverse over a collection to use the new for-each syntax instead. Although our example used an `Iterator` class, we were interested in all object-based traversals, including `Enumeration` traversals. The Query Refiner was launched to remove any facts that were part of the example change, but not generally relevant to our RSC[3]. Using the same graphing technique as before, we plot the results with respect to the results in the order they were returned. Figure 4.6 represents the first 500 results graphed.

As can be seen from the graph, Reverb performs very well. By the 96th result, Reverb had already reached 95% recall with a precision of 93%. 100% recall, however, is not reached until the 3630th result due to a bug in the way Reverb handles anonymous inner classes. Ignoring outliers due to that bug[4], Reverb reaches 100% recall by the 120th result at a precision of 98%.

[3]We investigate the difference between the refined and unrefined query in Section 4.3.

[4]All of the graphs we present here reflect the unfiltered results, affected by the anonymous inner class bug.

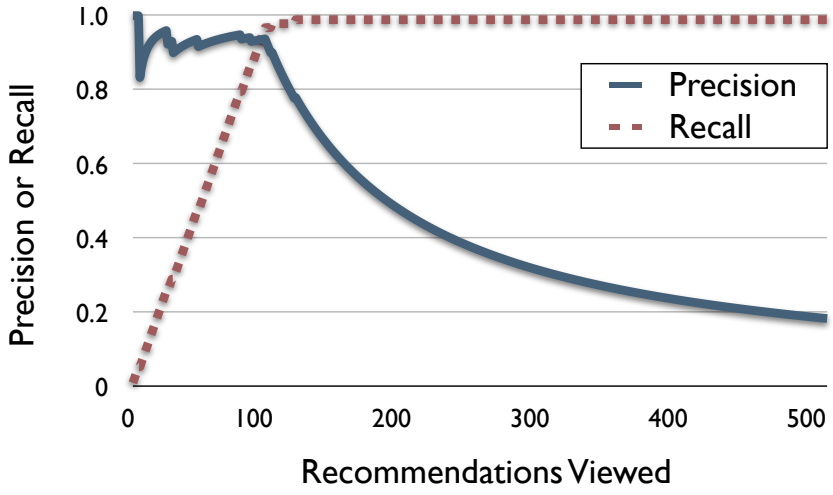

Figure 4.6: Results for object-based iteration migration

Figure 4.7 depicts the precision and recall versus the similarity metric for the object-based iteration. Since the recall is high for the strong similarity metrics (0.8 and above), the precision remains high until similarity drops below 0.6, when large numbers of false positives begin to appear. Graphing the precision versus the recall (Figure 4.8) shows that the precision remain strong as the recall value increases. The sudden drop in precision at the end of the graph marks the value at which the inner class outliers are reached.

For our second query, we transformed, at random, a location within JHotDraw that uses an index-based for-loop traversal over a collection and graph the results. Figure 4.9 represents the first 500 results.

Although the results are not as strong as before, Reverb still performs very well. By the 103rd result (out of 4,889), Reverb has reached a recall of 95%. Ignoring the outliers introduced by the anonymous inner class bug, this point signifies the 100%

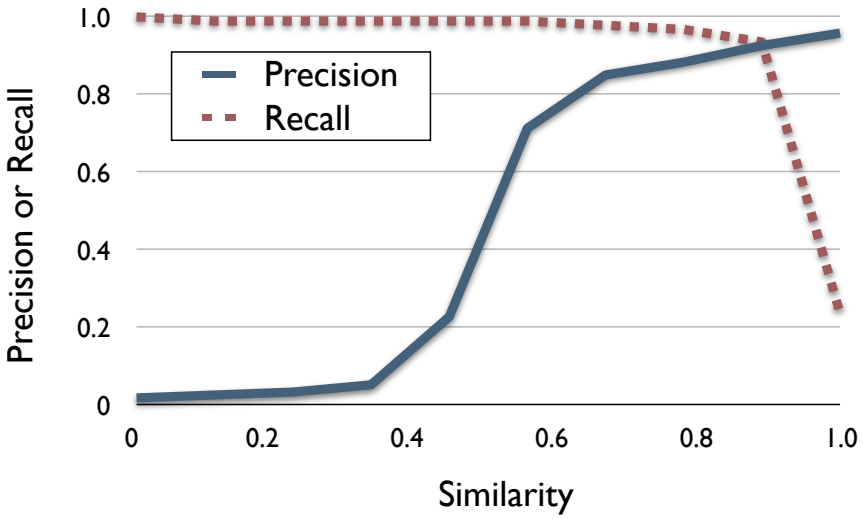

Figure 4.7: Precision and recall versus similarity for object-based iteration

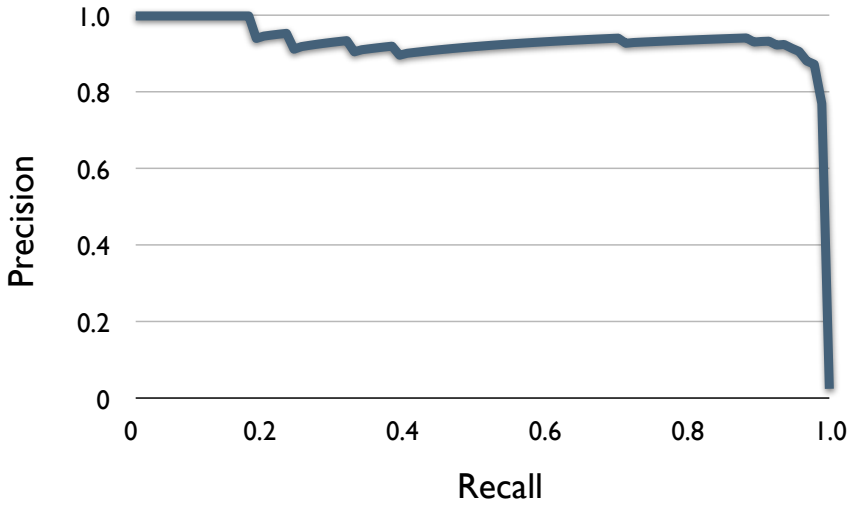

Figure 4.8: Precision versus recall for object-based iteration

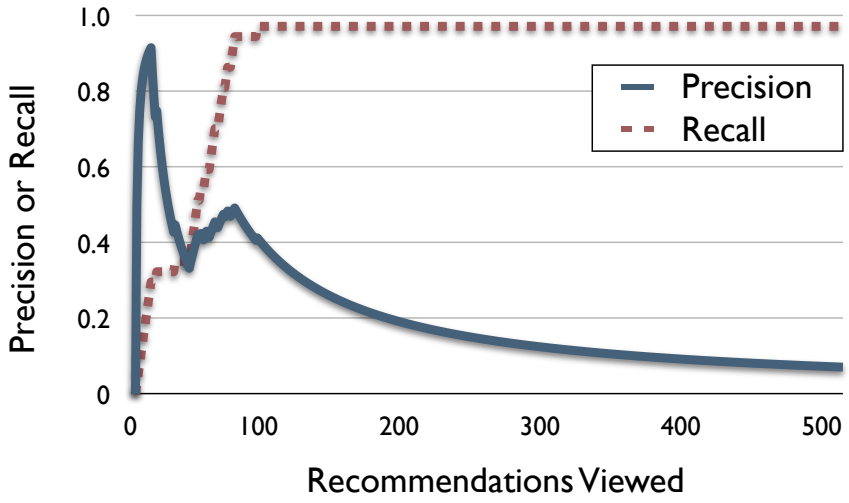

Figure 4.9: Results for the index-based iteration migration

recall point. The precision is initially very high (above 90% for the first 10 results), but drops between the 18th and 41st result. By the time the precision permanently drops below 50%, a recall rate of 88% has been achieved.

Graphing the precision and recall versus the similarity (Figure 4.10) shows that the precision and recall are quite high for the highest similarity levels before dropping. However, graphing the precision versus the recall (Figure 4.11) shows the effects of two groups of false positives, one within the first few results, and another around the 0.3 recall point. Aside from these pockets, the precision and recall rate remains relatively good until the anonymous inner class outliers appear as the recall rate approaches 1.0.

4.3 Evaluating The Fact Selection Process

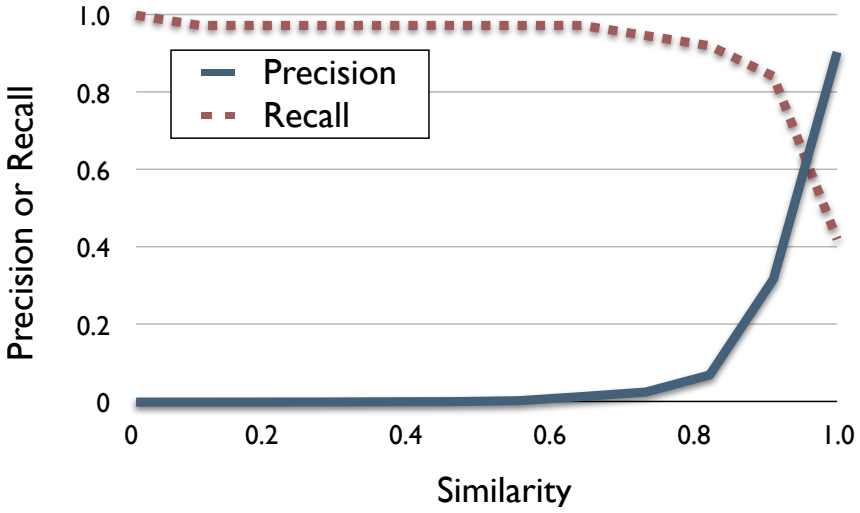

Figure 4.10: Precision and recall versus similarity for index-based iteration

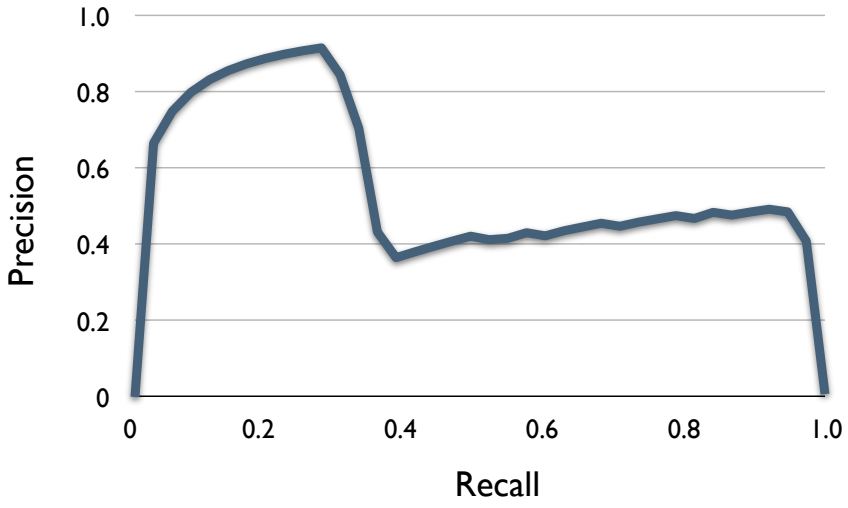

Figure 4.11: Precision versus recall for index-based iteration

Reverb selects which facts are designated as query facts through a simple set subtraction process. Any facts that are removed or modified after the change are selected for query. The Query Refiner allows developers to manually change the facts selected for query. In Section 4.1, the Query Refiner was not used to improve the results. In Section 4.2, the Query Refiner was used to achieve better results. We now evaluate how the fact selection and refinement processes affected those query results.

4.3.1 Moving to a Logger Framework

Through Reverb's fact selection process, an optimal query was achieved by default for the logging framework RSC demonstrated in Section 4.1. The query facts were selected based on the differences between the "before" and "after" snapshots of the example method provided. To understand how Reverb would perform without this selection process, we repeated the same task with all facts selected for query instead of the reduced subset based on the change. Figure 4.12 presents the results of this query, graphed using the same precision and recall metrics.

As can be seen from the graph, the query results without Reverb's fact selection process is not impressive. 100% recall is not reached until the 69th result, and precision drops below 50% on the second result at a precision of only 16%.

4.3.2 Migrating to a for-each loop

For the loop migration examples, the Query Refiner was used to help improve the query results. While investigating why human intelligence was able to improve upon the initial results is beyond the focus of this thesis, a simple comparison could help understand the depth of such a gap. Figure 4.13 depicts the results for the index-

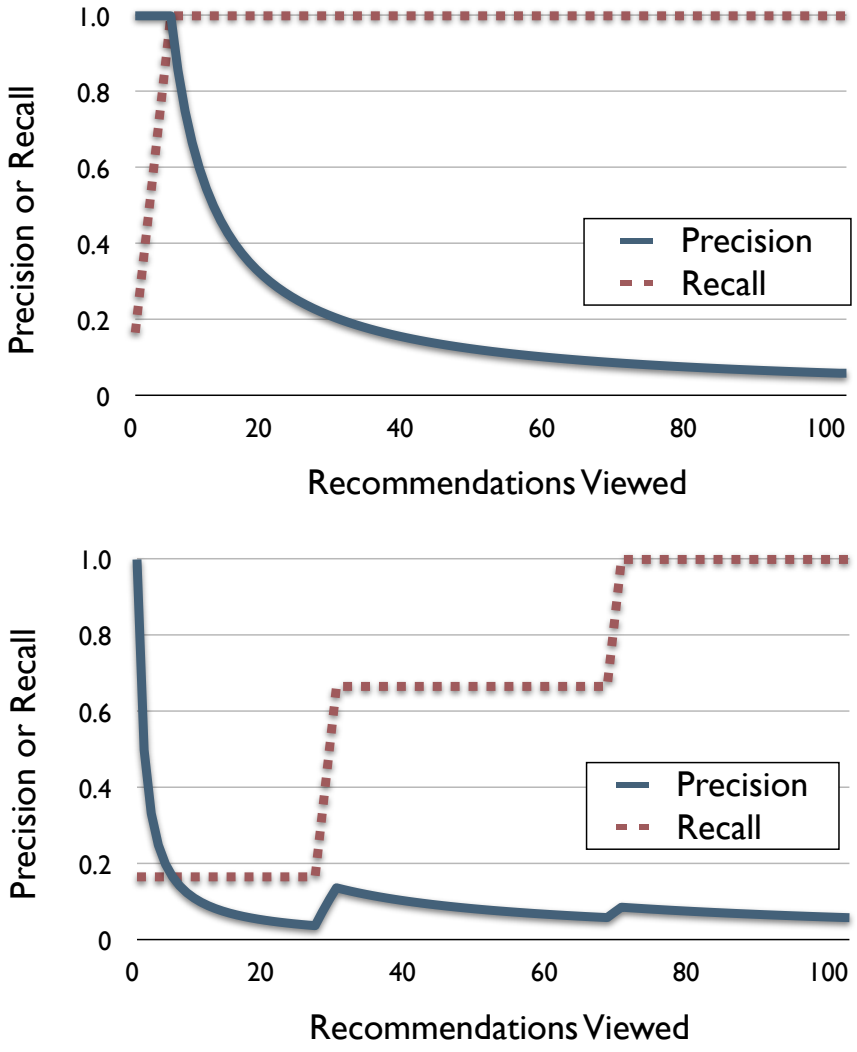

Figure 4.12: Logging framework results with (above) and without (below) automated fact selection.

and object-based loop examples presented in Section 4.2.

As can be seen from the default object-based query graph, the results benefitted from the manual Query Refinement. Outliers were reduced, as the 95% recall rate was not achieved until well after the 500th result without some manual query refinement. Precision was also improved with the refinement.

The difference between the default and manually refined index-based query is not as large as that for the object-based query, but Reverb's results are still noticeably better after some manual refinement. Again, outliers were reduced as a recall of 90% was reached only by the 308th result without refinement, versus the 61st result with refinement. The pocket of false hits was also shortened, improving the overall precision.

4.4 Summary

Using the Reverb tool, we performed three evaluations based on two RSCs. In the first example, we attempted to migrate Apache Struts to use a logging framework consistently, removing guarded debug statements. Reverb achieved optimal search results without query refinement, though some false positives were given the same similarity metric as some true positives. For the second and third example, we attempted to migrate JHotDraw to use Java 5's for-each syntax. Reverb performed very well for both object-based and index-based iterations, especially after some manual query refinement.

The gap between Reverb's fact selection process and a process using human intelligence was relatively narrow, but still noticeable. In cases where optimal results

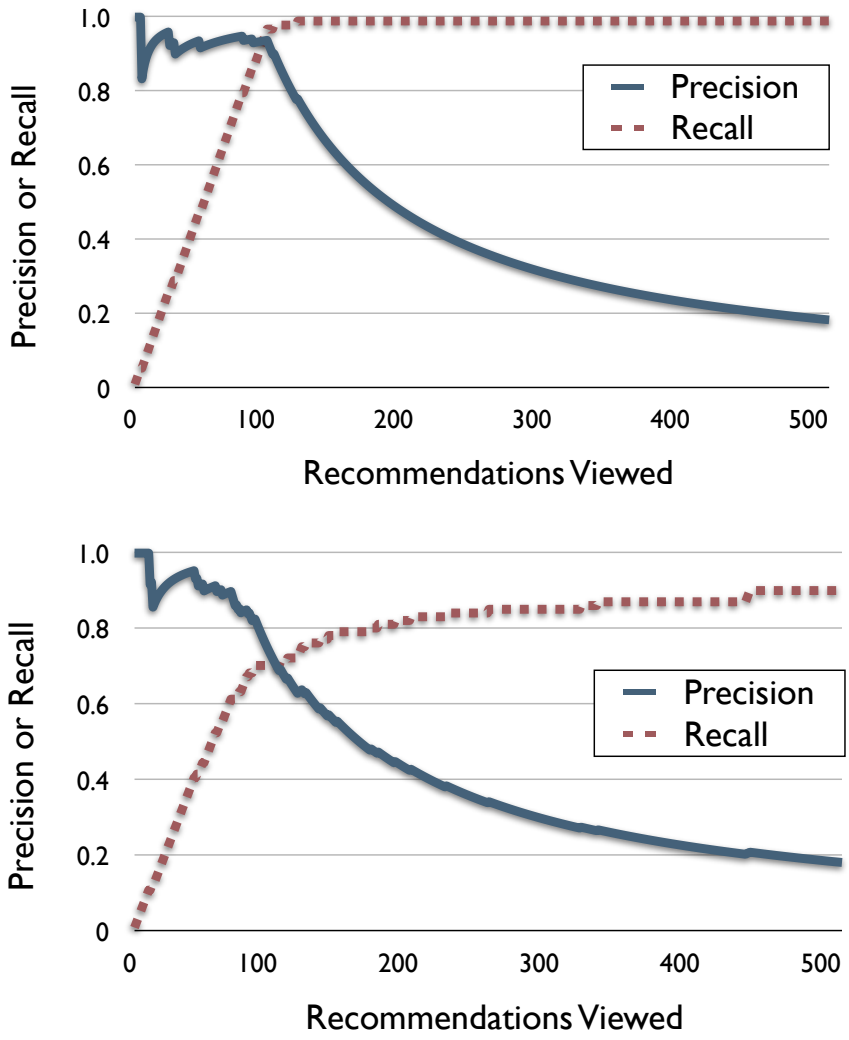

Figure 4.13: Object-based iteration migration results with (above) and without (below) query refinement

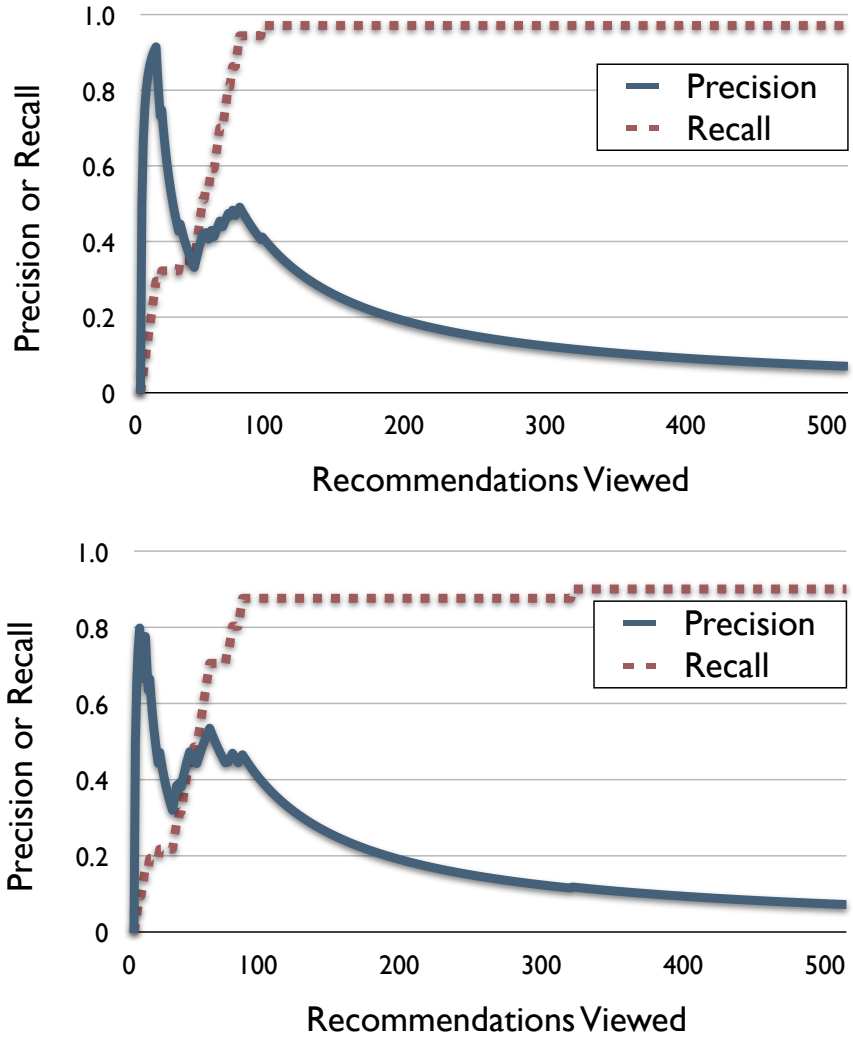

Figure 4.14: Index-based iteration migration results with (above) and without (below) query refinement

were not achieved by default, manual refinement helped improve the results.

Chapter 5

Discussion

In the previous chapters we described the problem with RSCs and showed how our approach can help find relevant RSC target locations within a codebase better than traditional approaches for the scenarios evaluated, both in terms of developer effort and quality of results. We now discuss possible limitations and pitfalls of our approach, alternate uses, and possibilities for future extension.

5.1 Semantic Similarity

While lexical similarity can be expressed in terms of character patterns, the same cannot be said for semantic similarity. For a segment of code to be labeled "semantically similar," it need not perform the same function or have the same result, and it is certainly not required to have the same same lexical or syntactic structure[1].

Semantic similarity can have many interpretations, and to adequately solve the RSC problem, the definition of such similarity may depend on what the developer is trying to accomplish. To put this in the form of an example, we observe the two lines of code in Figure 5.1 and compare them to those in Figure 5.2. Depending on the developer's goals, similarity can be assumed or dismissed in each of these scenarios.

We consider the idea that the developer does find the examples in the figures

[1]In this thesis, we define lexical matches as having a one-to-one correspondence in the character data. Syntactic matches may contain differences in variable names and literal values, but otherwise has identical keywords and symbols.

```
System.out.print("Hello, World!");
System.err.print("Hello, World!");
```

Figure 5.1: A pair of possibly similar code segments

```
System.out.print("Hello, World!");

String a = "Hello, World!";
System.out.print(a);
```

Figure 5.2: A second pair of possibly similar code segments

above as being similar. Each of the examples could be considered similar because the lexical string "Hello, World!" is being output, regardless of whether the output is sent to `System.out` or `System.err`, or whether the output is passed as a String literal or as a variable that had been assigned that value. On the other hand, if the developer considers the examples in Figure 5.1 to be similar, but not those in Figure 5.2, then perhaps we conclude that the RSC involves outputting "Hello, World!" directly, without intermediate variable storage. Likewise, if the examples in Figure 5.2 are considered similar, and not those in Figure 5.1, then we can conclude that outputting "Hello, World!" to `System.out` is an important aspect of the RSC. If none of them are similar, then the developer may have other intentions in mind entirely.

Since RSCs are difficult to locate because of their semantic structure and not necessarily their syntax, we must develop a technique whereby semantic similarity is used to locate RSC targets. However, we cannot make too many assumptions about what constitutes this similarity, which is why we provide the developer with tools to adjust our idea of what similarity is in each situation.

5.2 Granularity and Inter-Method Relationships

Reverb observes changes and returns results at the method-level granularity. While we argue that small-scale changes normally exist within the confines of a single method, this is not always the case. There are scenarios where a relatively small change target may be defined by relationships that cross method boundaries.

An example of an RSC that may include method-spanning Facts is presented in Figure 5.3. When the SWT user interface framework was updated to version 3.0, internal changes were made such that a method, `setLayoutData`, would throw an exception if the layout being used of the `RowLayout` class. This was not the case in previous versions, and the SWT documentation suggests that developers search for all locations where `setLayoutData` is being called on objects being positioned with a `RowLayout` layout type.

As can be seen in Figure 5.3, it's possible that the information needed to locate RSC targets of this nature can be scattered across methods; it would be impossible for our approach to find these locations.

While our inter-method tracking limitation is problematic for these situations, implementing an inter-method RSC target finder would have been more complicated. Since Reverb is a research prototype, it was decided to test the concept on intra-method relationships to simplify the implementation and make the results easier to analyze. In addition to reducing the complexity of the query, selecting a limited, but well-defined granularity allows us to define changes by observing developer navigation.

```
/* Method 1 */
public void interMethodExample() {
    Display display = new Display();
    Shell shell = new Shell(display);

    this.prepareShell( shell );

    // Now we add an object with LayoutData
    Button myButton = new Button(shell, SWT.PUSH);
    myButton.setText("Click␣Me");
    myButton.setLayoutData( new RowData(50, 40) );

}

/* Method 2 */
pubic void prepareShell( Shell shell ) {
    shell .setLayout( new RowLayout() );
}
```

Figure 5.3: An RSC target location that contains inter-method relationships. It would be impossible for our approach to locate this situation because it only tracks intra-method relationships.

5.3 Performing an Example Change

For the examples evaluated in this paper, we knew of at least one RSC target location for each scenario that we used to demonstrate an example change for Reverb. However, there may be situations where the developer would not know of such locations straight away. For example, if a developer learns of an API method deprecation, this could trigger an RSC where the launch point is unknown.

In this situation, we recommend that the developer create a temporary method demonstrating the state of a method before the change. The change could be demonstrated to Reverb using this temporary method, and locations elsewhere in the project's source where the same change ought to be made should be returned. If the results are ranked with a very low similarity, it is possible that the RSC is not applicable for that project.

5.4 Defining Change Snapshot Points

Navigation is tracked for each java file allowing the developer to task-switch to other classes and return without affecting the initial "before" snapshot for that class. This "before" snapshot will not be replaced with a new one unless the developer navigates to another method within the same file, signaling that a new small-scale change has started. While this allows for ad-hoc usage without sacrificing task switching between other classes, one can conceive of scenarios where this behaviour would not be appropriate. In particular, if the developer begins a change, task-switches within the *same* class, and returns to completing the original change, then only the post-task-switch portion of the change will be detected by Reverb.

To avoid such situations, "begin observation" and "end observation" commands could be added to the Reverb user interface. These commands would override the automated behavior and allow the developer to define the change snapshots manually. While helpful, this functionality does not contribute to the precision and recall of our evaluation and was not added to our prototype.

5.5 Fact selection

During the development of Reverb, we experimented with many different fact and relationship types. The ones that yielded the most relevant results were incorporated into the tool.

Determining whether or not a fact contributes to a "relevant result" depends on the RSC being addressed. Unfortunately, erring on the side of including as many facts as possible was not a solid approach. Relevant Facts would become buried in a wealth of information, and it became increasingly difficult to identify which Facts were important from a similarity perspective. In particular, certain operators, meta information about where references are defined, and special emphasis on call chains were found to provide too much noise. Also, some of this information, such as references involved in call chains, was able to be retrieved implicitly through the existence of other facts, attributes, and associations.

While a quantitative analysis of which facts, attributes, and associations provide the best average results would require too much time for this study, we are confident based on qualitative evidence that the factbase selections perform reasonably well to support the thesis of this dissertation.

5.6 Ranking similarity

Our approach calculates similarity using straightforward algorithms for the purpose of keeping evaluation simple. In particular, "important" facts are selected through basic set arithmetic. Facts that are in the initial snapshot that are not in the second are automatically selected as being "important." Also, changes in attributes from the initial to the final snapshot are noted as being important.

Using set arithmetic generally produces good results, but since the queries generally improved after developer refinement, there are many possibilities for improvement in selecting important facts. One particular idea that was experimented with was the concept of using a scalar value as an importance weighting to various fact types. Since RSCs do not typically involve heavy dependence on lexical properties, attributes such as `hasName` could be given a small weight, whereas dataflow attributes such as `flowsFrom` could be given a large weight. While experiments showed that such a ranking is possible, it was deemed unnecessarily complicated for an initial evaluation of the concept. It also introduced additional problems such as how to empirically decide on an appropriate weighting for each fact and attribute.

An additional problem with ranking is the depth at which relationships are addressed. Since relationships between facts form a graph, determining which facts are matched by relationship is a recursive process with an arbitrary threshold as to what constitutes a "match."

For the purposes of our prototype tool, matches in fact relationships were decided by a pre-calculated meta value. The meta-value is an identifier formed from the most important Attributes for a particular fact based on empirical tests. This currently

involves the `isType`, `hasValue`, `hasName`, and `isLoop` attributes, as well as the actual type of the fact itself. These values together form a threshold identifier that is not necessarily unique, but usually turns out to be unique for any particular method. This process is quick and gives results that perform well for the amount of resources required to do the computation.

The meta value calculation, however, is not error-proof. A better implementation would follow relationships as deeply as possible. Initial algortithms used in the development of Reverb used such approaches, but were dropped in the interest of clarity for evalution.

Another shortcoming in the similarity ranking algorithm is that queries are formed using simple AND boolean logic. That is, the existence of a particular fact, attribute, or association strengthens the similarity, and the absence of those facts, attributes, and associations weakens it. In reality, there are situations where OR and NOT boolean logic could improve the query.

In our first motivational example, if we invoked Reverb on a codebase that already had some of the new "for each" loops converted, Reverb would probably include many of these in the results. It would be therefore desirable to indicate that it is important for the `CFG` Fact to not have a `isType: forEachLoop` attribute, but still have an `isLoop: True` attribute. This is currently not possible in Reverb's prototype implementation.

5.7 Performance

Reverb took approximately three minutes for each of the scenarios presented in Chapter 4 on a MacBook with 2GHz Intel Core Duo processor and 1 GB of RAM. As Reverb has not been optimized with respect to execution time or memory usage, a more thorough performance evaluation should be conducted once optimizations have been implemented beyond our prototypical implementation.

5.8 User refinement

Reverb provides the developer with a query refiner to help give extra knowledge about which facts, attributes, and associations are deemed important. While it is useful to present this information to the developer in its current implementation, we can use even simpler information from the developer to derive additional facts.

This could be implemented by a simple "yes" or "no" question directed at the developer for each result returned: "Is this result correct?" If the developer indicates that no, the result is not correct, additional boolean logic could be applied to refine the query. We can discover what is different about the inaccurate result that could be removed from the query.

5.9 Summary

While we believe our technique adequately validates the premise of this thesis, there are still pitfalls and shortcomings that could stand to be addressed. The method-level granularity leaves out a subset of RSCs that might otherwise be addressed and results are presented at a higher granularity than is actually represented in the source code. Fact selection could stand to use some qualitative analysis to select the

best general cases. Similarity ranking is currently binary which treats all Facts and Attributes as having the same importance, even though some may be more relevant than others to the developer. Relationships between Facts are calculated based on pre-calculated meta information that doesn't follow multiple levels of relationships explicitly, and more advanced boolean logic is not applied when forming a query. Finally, user refinement is more complicated than it has to be, since simple "yes" or "no" questions could be asked to the developer.

Chapter 6

Related Work

The task of locating relevant targets in source code has mostly been limited to query languages and lexical pattern matching. Nevertheless, there is work in the academic community to make searching source code easier and allow for less dependence on lexical similarity. In this chapter, we discuss these ideas and how they relate to RSCs and our technique to discover them.

We begin by considering clone detection techniques and how our technique relates to them in Section 6.1. Plagarism detection (Section 6.2) attempts to discover code locations that have been deliberately altered from their original incarnations. Refactoring, an ad hoc approach to program modification, is discussed in Section 6.3. Sections 6.4 and 6.5 outlines exact and approximate query techniques that may be used to locate similar code locations. Section 6.6 describes work done to correlate changed source code. Programming by example (Section 6.7) shares our goal of automating repetitive tasks through demonstration, rather than explicit instruction. Finally, we discuss other related works in Section 6.8.

6.1 Clone Detection

The most obvious approaches that address problems similar to our motivational scenario are clone detection techniques. Clone detection is geared towards finding duplicate or near-duplicate segments of code, typically arisen from copy-and-paste

programming practices [Lange and Moher, 1989]. Unless such redundancies are detected and eliminated, a necessity to change multiple code segments can arise [Geiger et al., 2006].

There is disagreement in the research community as to what constitutes a clone [Koschke, 2007] and there has been more than one attempt to classify the concept [Balazinska et al., 1999; Kapser and Godfrey, 2003]; however, much work is geared toward discovering locations that may have lexical differences in identifiers or type changes, but would otherwise be identical.

The simplest form of clone detection can be performed through line-by-line matching algorithms [Baker, 1992] or string comparisons [Johnson, 1994]; however, interest has been focused on more efficient techniques, or those that allow for the detection of clones regardless of identifiers, whitespace, and other simple differences.

Baker [1995] has developed a technique using a suffix-tree algorithm [Gusfield, 1997] to compare entire tokens, without introducing syntax into the comparison. Kamiya et al. [2002] propose a hybrid approach based on Baker's token comparison technique and lightweight, language-specific filtering rules.

Baxter et al. [1998] promotes the comparison of abstract syntax trees to detect subtree matches within a code base. Although specifics of the process are not described, only exact or near-miss clones are claimed to be supported. This process is similar to that employed by DECKARD [Jiang et al., 2007], which additionally introduces characteristic vectors to represent abstract structural information. Coogle (Code Google) [Sager et al., 2006] also detects similar code by tree-matching on Java classes, but for the purposes of example finding.

Developers intending to address bugs introduced by incorrect copy-and-modify

operations may also benefit from clone detection approaches [Li et al., 2006]. Kontogiannis et al. [1996] suggest pattern matching to help developers discover clones to better understand and maintain legacy software systems. Duala-Ekoko and Robillard [2007] propose clone region descriptors to track code clones between software versions and notify developers maintaining the software when clones are being modified.

Clone detection approaches focus primarily on discovering large segments of code without much variation. The problem we address may involve as little as a single line of code and contain several syntactic or structural differences. More fundamentally, valid assumptions in the context of clone detection as to the provenance of clones do not hold in our context.

6.2 Plagiarism Detection

Detecting the plagiarism of computer code can be considered as a branch of clone detection, with an important distinction: Plagiarism, unlike code clones, is often camouflaged with deliberate changes to the code, making plagiarized code segments harder to identify. This requires a broader definition of similarity, sharing some goals with our approach.

Prechelt et al. [2000] aim to detect student plagiarism in Java code through token analysis of syntactic elements. This approach does not consider semantic information about the code, discarding identifiers, data types, and other information in favour of a simple analysis of specified keywords and symbols. Similarly, Jankowitz [1988] constructs program templates for Pascal programs based on syntactic elements, while ignoring most semantic elements.

Unlike our approach, code plagiarism detection techniques are geared toward structural similarity rather than semantic similarity. Since student assignments inherently bear a semantic similarity even when not directly plagiarized, plagiarism detection purposefully avoids detecting the type of similarities that are beneficial to RSC target location.

6.3 Refactoring

Refactoring [Opdyke, 1992; Griswold and Notkin, 1993; Tokuda and Batory, 1999; Fowler, 2002] involves the meaning-preserving transformation of code, facilitating a particular change. These transformations, like those required to complete RSCs, often occur in an unknown number of locations within a codebase. Examples of refactorings include renaming an indentifier, adding a parameter to a method, and moving a method up a class hierarchy.

While many refactoring tools do not focus on changes on the small-scale, a few do, such as local variable renaming utilities. Such utilities work on sufficiently well-defined problems based on invariable knowledge about syntax, allowing them to guarantee correctness and completeness of the transformation. Refactoring emphasizes this behaviour preservation and correctness. Opdyke [1992] defines this in terms of pre- and post-conditions; any input values must result in the same output values before and after the transformation. A more general or inexact problem requiring a notion of similarity is not well-suited for refactoring.

The for-loop migration problem described in our motivational chapter is an example of a transformation with behaviour preservation; however, the problem is not

solved by available refactoring tools. Object-based iterations—particularly custom classes not based on any standard interface—require knowledge about the semantics of the iteration methods before they can be transformed. Reverb observes the developer to obtain some of this knowledge and assign similarities based on the observation. A generalized refactoring tool could not rely on this information being available.

6.4 Exact Search Approaches

There exists much work in the research community to explicitly query for patterns in source code relevant to developer tasks.

JQuery [Janzen and Volder, 2003] is a query language for browsing source code within the Eclipse development environment. It works by automatically populating a rule-based logic processor with observations extracted from a project's source code. The developer is able to query the logic processor using a Prolog-like syntax, with JQuery facilitating the correlation between the returned results and the actual codebase. Conceivably, one could locate certain well-defined RSC targets by building queries using JQuery; however, doing so places significant cognitive burden on the developer to specify a sufficiently broad query to capture all desired locations while being sufficiently narrow to avoid capturing too many undesired locations.

CrocoPat [Beyer, 2006] uses relational programming to analyze software system architecture and discover graph patterns related to structural problems, design patterns, and code clones. Grok [Holt, 1999] is a relational calculator usually populated with facts and relationships extracted from large software systems. Software facts

manipulated with grok are suitable for extracting architecture diagrams, and other reverse engineering tasks. Like JQuery, both of these techniques are based on rule-based logic processors to derive answers to queries and can relate those answers with source code locations.

TXL [Cordy, 2006] is a transformation language for source code. Specifically, it allows a developer to transform occurrences of one defined structure into another using two separate grammars. These two grammars are based on formal tree-rewriting principles, and the concept fits well into the design process allowing for rapid prototyping using formal design notations. While the search for and transformation of code is similar to the goals of this paper, TXL's execution is designed for a different purpose. TXL focuses heavily on formal design principles, whereas RSC completion occurs at a finer granularity than the formal design process. Developers do not always follow stringent design principles during software development, and it would therefore be burdensome to use TXL to address small-scale problems, such as RSCs.

Each of these exact search techniques are designed to locate specific structural patterns. Unfortunately, when faced with an RSC, there may exist variation between target locations. At the time of query formation, we do not know the limit to the variation present within the codebase. The user becomes burdened to describe—in precise terms—the structure of a problem that could be expressed in any number of ways [Holmes et al., 2006].

An exact query attempting to capture all possible variations must determine functional equivalence, an undecidable problem [Turing, 1936]. We use the notion of similarity to overcome this issue. Though such approximation potentially introduces false and missed positives, less well-defined structures can still be discovered.

6.5 Approximate Search Approaches

The concept of extracting facts and relationships from source code to maintain a query-able approximation is not unique to our technique.

Holmes and Murphy [2005] promote Strathcona, an example recommendation tool that can be used when inadequate documentation is available for a specific framework feature. Like the technique described in this paper, Strathcona automatically forms queries based on developer-provided code and returns results that it deems relevant to the developer's examples. Unlike our technique, Strathcona is intended as a replacement for poor documentation, locating other segments of source code that can assist the developer in using various framework objects and methods. In particular, Strathcona's perception of the codebase is at a much larger granularity than appropriate for RSC target location, focusing on inter-method relationships while ignoring the smaller-scale details. Additionally, since Strathcona is primarily designed to work with static frameworks, it requires pre-processing every time the code of that framework has changed. Lastly, while Strathcona will have succeeded if a single good example is returned, a utility to discover RSC targets must return all relevant locations.

Suade [Warr and Robillard, 2007] recommends locations relevant to a task based on a developer-provided context of fields and methods. The developer manually specifies the names of fields and methods of interest, and Suade returns locations that are either called, called by, or accessed by that context. Like Strathcona, Suade relies on a pre-generated database of program information and is intended to assist in source code understanding, succeeding if a few good locations are returned. We

are interested in much finer-grained similarities than these techniques, where control flow and dataflow is as important a consideration as the structure of the code.

6.6 Correlating Changes

Techniques to discover corresponding entities between two or more versions of a software system [Horwitz, 1990; Laski and Szemer, 1992; Apiwattanapong et al., 2004] share a common necessity with our technique. To calculate a similarity metric, we must infer correspondence between the "before" and "after" picture. As changes are made to a software system, it becomes increasingly difficult to correlate changed sections with their original incarnations [Berzins, 1986].

Kim et al. [2007] propose a way to match code segments between software versions using a grammar of atomic change rules, incuding the replacement or deletion of packages, classes, procedures, and arguments. Change distilling [Fluri et al., 2007] attempts to extract changes based on tree differencing. Both techniques use a similar iterative matching technique to Reverb's once sufficiently atomic change definitions have been decided; however, the larger abstract concepts required by Reverb, such as dataflow and control flow presence, cannot be defined as succinctly. We overcome this by suggesting either an approximate, or limited recursive approach to matching relationships (see 3).

History mining applications like ROSE [Zimmermann et al., 2005] and Hipikat [Čubranić and Murphy, 2003] use meta information extracted from versioning repositories to facilitate correlation. This information is not necessarily available to developers faced with an RSC.

6.7 Programming by Example

Programming by example (PBE) [Cypher et al., 1993] is a means to automate common, repetitive actions through demonstration rather than explicit instruction. While this field of work is primarily geared toward user interface improvement, it shares a common goal of trying to repeat a given action by means of example. Unlike our technique, previous approaches to PBE are based on pattern analysis of user actions, and does not operate on the semantic spaces required for a structural analysis of source code. Still, Reverb's implicit query extraction could be viewed as a novel PBE approach.

6.8 Other Related Work

While the representation of our extracted factbase is relatively informal, there exists work to formalize facts extracted from software to facilitate exchange [Lin and Holt, 2004]. Such formalizations could be beneficial for future iterations of our work, but are unnecessarily detailed for an initial prototype analysis.

Richter [2004] has provided formal foundations to justify context-specific notions of similarity; however, these formalizations do not permit the developer to be the arbiter of what constitutes similarity for a given task.

6.9 Summary

There exists much related work that attempts to support the developer in navigating or discovering code locations relevant to certain tasks.

Clone detectors search for exact or near-exact code segments for the purposes of reducing redundancies and potential maintenance problems. While some RSCs could loosely be considered clones, the scale, variation, and provenance of RSCs make them ill-fitted for the classic definition of code clones. Plagiarism detection loosens the definition of clones by searching for code that has been copied and deliberately camouflaged; however, these techniques purposefully avoid emphasis on semantic similarities beneficial to RSC detection, since non-plagiarized code would also bear such similarities.

Refactoring involves the meaning-preserving transformation of code segments for the purposes of improved understanding or evolvability. While some RSCs may be considered refactorings, we wish to address a wider range of changes, even those that are not necessarily meaning-preserving. More importantly, the strict notion of correctness applied to refactoring techniques makes them unsuitable for discovering some RSC targets, in which precise semantic information may not be available.

Exact query techniques search for specific patterns in source code based on the well-defined requirements of the query author. A developer attempting to discover RSC target locations may not have the information required to make such an explicit query, as RSCs contain variations of unknown extent. Approximate query techniques, like our approach, use a notion of similarity to discover relevant source locations, however most such techniques are focused on high-level details and focus on returning only a few good matches.

Chapter 7

Conclusion

Repeating a small change throughout a software system is often necessary; however, finding each location where a change should be applied can be difficult. A developer cannot always rely on the existence of lexical or syntactic patterns to make use of traditional search techniques.

We have presented a heuristic search technique that uses observations of the developer's changes to find likely repetitive small-scale change locations. These observations occur in the background and are used to infer semantic information about the change at hand. This information is then used to query other methods in the developer's project with the intent to discover locations where a similar change ought to be made.

The thesis of this dissertation is that a semi-automated heuristic search discovers target locations for repetitive small-scale changes with better precision and recall than traditional techniques, and with a greater tolerance to variation. In support of this, we have conducted an evaluation of our prototype implementation against three RSC tasks in two open source software systems.

In the first study, we attempted to update Apache Struts 1.1 to consistently use a logging framework in place of guarded debug statements. Struts inconsistently used a combination of such statements, along with a proper logging framework, but the quantity and extent were unknown. By performing an example change and invoking Reverb, we were able to find all of the relevant RSC target locations without viewing

any false positives. None of the traditional techniques evaluated (lexical search, syntactic search, and clone detection) met the 100% precision and recall of Reverb.

In our second study, we attempted to migrate JHotDraw 5.4 to use the new for-each loop introduced in Java 5. This task was split into two RSCs: migrating index-based collection iterations, and migrating object-based collection iterations. For both of these tasks, we found that Reverb resulted in better precision and recall than lexical search, syntactic search, and clone detection techniques—which suffered in the face of code variations; and manual query refinement improved the precision of Reverb's automated heuristic even more than the default query.

7.1 Future Work

A semi-automated heuristic search has demonstrated potential for supporting repetitive small-sclae changes and warrants further research and development. In particular, there are five topics of future work that could be addressed in the near-term: Granularity of change analysis, improvement of query refinement, formalization of factbase elements, improvement of query logic, and reduction the result set.

The technique presented in this thesis works at an intra-method granularity. While this limitation helps define boundaries for observing a change, detecting RSCs that span methods could expand the problem space addressed by our technique. We have identified one such case, presented in Chapter 5, which involves the layout of AWT components. While the change is still small-scale and potentially repetitive, it cannot be addressed without tracking relationships between methods.

Query refinement is an important facet of our tool, as the definition of similarity

is inherently imprecise. While we provide a facility for manually refining queries, simpler information may be gathered from the developer. Specifically, results can be deemed to be "helpful" or "unhelpful" and the query can be adjusted by observing differences that could have potentially warranted the designation.

While the fact, association, and relationship types used for our prototype implementation performed well, an empirical analysis and formalization of fact types required for our technique would improve the generality of our approach. While it would have been premature to define an ideal set of factbase data for our initial analysis, we now have a basis for such refinement.

The query and comparison logic used by our approach is based on simple AND arithmetic. While this performs well for the cases evaluated, there are scenarios where the absence of a particular fact, attribute, or association may be important in correctly identifying RSC targets.

The result set of our technique consists of each method in a project, sorted by their similarity metric. Further analyzing the result set of our technique could help specify a threshold to eliminate the number of false positives returned, and define a point at which the developer can confidently stop browsing.

There are other, more long-term goals of our approach that warrants additional research. While this thesis focused on locating repetitive small-scale change target locations, we envision a tool that could actively suggest changes at each location. Such technology would assist software developers in completing particularly repetitive and error-prone tasks; identifying the locations where these tasks are required is a positive start.

Bibliography

Apiwattanapong, T., A. Orso, and M. J. Harrold (2004). A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, Washington, DC, USA, pp. 2–13. IEEE Computer Society.

Baeza-Yates, R. A. and G. H. Gonnet (1989). A new approach to text searching. In *SIGIR '89: Proceedings of the 12th annual international ACM SIGIR conference on Research and development in information retrieval*, New York, NY, USA, pp. 168–175. ACM Press.

Baker, B. S. (1992). A Program for Identifying Duplicated Code. *Computing Science and Statistics 24*, 49–57.

Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, Washington, DC, USA, pp. 86. IEEE Computer Society.

Balazinska, M., E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis (1999). Measuring clone based reengineering opportunities. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, Washington, DC, USA, pp. 292. IEEE Computer Society.

Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna, and L. Bier (1998). Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Confer-*

ence on Software Maintenance, Washington, DC, USA, pp. 368. IEEE Computer Society.

Belady, L. A. and M. M. Lehman (1976). A model of large program development. *IBM Systems Journal 15*(3), 225–252.

Berzins, V. (1986). On merging software extensions. *Acta Informatica 23*(6), 607–619.

Beyer, D. (2006). Relational programming with crocopat. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, New York, NY, USA, pp. 807–810. ACM Press.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM 11*(4), 481–494.

Clarke, C. L. A. and G. V. Cormack (1997). On the use of regular expressions for searching text. *ACM Transactions on Programming Language Systems 19*(3), 413–426.

Cordy, J. R. (2006). The txl source transformation language. *Science of Computer Programming 61*(3), 190–210.

Crochemore, M. and D. Perrin (1988). Pattern matching in strings. In V. Cantoni, V. Di Gesu, and S. Levialdi (Eds.), *Proceedings 4th Conference on Image Analysis and Processing*, pp. 67–79. Plenum Press.

Cypher, A., D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. My-

ers, and A. Turransky (1993). *Watch what I do: programming by demonstration*. Cambridge, MA, USA: MIT Press.

Duala-Ekoko, E. and M. P. Robillard (2007). Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, pp. 158–167. IEEE Computer Society.

Fluri, B., M. Würsch, M. Pinzger, and H. C. Gall (2007, July). Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering PrePrint, accepted for publication (to appear)*(-), 17.

Fowler, M. (2002). Refactoring: Improving the design of existing code. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, London, UK, pp. 256. Springer-Verlag.

Gamma, E., R. Helm, R. E. Johnson, and J. M. Vlissides (1993). Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, London, UK, pp. 406–431. Springer-Verlag.

Geiger, R., B. Fluri, H. C. Gall, and M. Pinzger (2006, March). Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, Number 3922 in Lecture Notes in Computer Science, Vienna, Austria, pp. 411–425. Springer.

Gosling, J., B. Joy, G. Steele, and G. Bracha (2005). *The Java Language Specification, Third Edition*. The Java Series. Boston, Mass.: Addison-Wesley.

Griswold, W. G. and D. Notkin (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering Methodology 2*(3), 228–269.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York, NY, USA: Cambridge University Press.

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

Holmes, R., R. J. Walker, and G. C. Murphy (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering 32*(12), 952–970.

Holt, R. C. (1999). Software architecture abstraction and aggregation as algebraic manipulations. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pp. 5. IBM Press.

Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, New York, NY, USA, pp. 234–245. ACM Press.

Houghton Mifflin Company (2004). *The American Heritage Dictionary of the English Language, Forth Edition*. Houghton Mifflin Company.

Jankowitz, H. T. (1988). Detecting plagiarism in student pascal programs. *The Computer Journal 31*(1), 1–8.

Janzen, D. and K. D. Volder (2003). Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA, pp. 178–187. ACM Press.

Jiang, L., G. Misherghi, Z. Su, and S. Glondu (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, pp. 96–105. IEEE Computer Society.

Johnson, J. H. (1994). Substring matching for clone detection and change tracking. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, pp. 120–126. IEEE Computer Society.

Kamiya, T., S. Kusumoto, and K. Inoue (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28*(7), 654–670.

Kapser, C. and M. Godfrey (2003). Toward a taxonomy of clones in source code: A case study.

Kim, M., D. Notkin, and D. Grossman (2007). Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, pp. 333–343. IEEE Computer Society.

Kim, S., K. Pan, and J. E. James Whitehead (2005). When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings*

of the 12th Working Conference on Reverse Engineering, Washington, DC, USA, pp. 143–152. IEEE Computer Society.

Kontogiannis, K., R. de Mori, E. Merlo, M. Galler, and M. Bernstein (1996). Pattern matching for clone and concept detection. *Automated Software Engineering 3*(1/2), 77–108.

Koschke, R. (2007). Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein (Eds.), *Duplication, Redundancy, and Similarity in Software*, Number 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

Lange, B. M. and T. G. Moher (1989). Some strategies of reuse in an object-oriented programming environment. In *CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, pp. 69–73. ACM Press.

Laski, J. and W. Szemer (1992). Identification of program modifications and its applications in software maintenance. In *Proceedings of the International Conference on Software Maintenance 1992*, pp. 282–290.

Lehman, M. M. and L. A. Belady (1985). *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc.

Lehman, M. M. and F. N. Parr (1976). Program evolution and its impact on software engineering. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, pp. 350–357. IEEE Computer Society Press.

Li, Z., S. Lu, S. Myagmar, and Y. Zhou (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering 32*(3), 176–192.

Lin, Y. and R. C. Holt (2004). Formalizing fact extraction. *Electronic Notes in Theoretical Computer Science 94*, 93–102.

Malpohl, G., J. J. Hunt, and W. F. Tichy (2000). Renaming detection. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, Washington, DC, USA, pp. 73. IEEE Computer Society.

Miller, R. C. and A. M. Marshall (2004). Cluster-based find and replace. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, pp. 57–64. ACM Press.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph. D. thesis, Urbana-Champaign, IL, USA.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*(12), 1053–1058.

Prechelt, L., G. Malpohl, and M. Philippsen (2000). Jplag: Finding plagiarisms among a set of programs.

Purushothaman, R. and D. E. Perry (2005). Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering 31*(6), 511–526.

Richter, M. M. (2004). Logic and approximation in knowledge based systems. In W. Lenski (Ed.), *Logic versus Approximation*, Volume 3075 of *Lecture Notes in Computer Science*, pp. 184–204. Springer.

Sager, T., A. Bernstein, M. Pinzger, and C. Kiefer (2006). Detecting similar java classes using tree algorithms. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, New York, NY, USA, pp. 65–71. ACM Press.

Sullivan, K. J., W. G. Griswold, Y. Cai, and B. Hallen (2001). The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, pp. 99–108. ACM Press.

Sun Microsystems (2004). Jdk 5.0 documentation.

Tokuda, L. and D. Batory (1999). Evolving object-oriented designs with refactorings. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, Washington, DC, USA, pp. 174. IEEE Computer Society.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*(42), 230–265.

Čubranić, D. and G. C. Murphy (2003). Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA, pp. 408–418. IEEE Computer Society.

Warr, F. W. and M. P. Robillard (2007). Suade: Topology-based searches for software investigation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, pp. 780–783. IEEE Computer Society.

Yau, S., J. Collofello, and T. MacGregor (1993). Ripple effect analysis of software maintenance. pp. 71–82.

Zimmermann, T., P. Weigerber, S. Diehl, and A. Zeller (2005, June). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering 31*(6), 429–445.

Appendix A

Loop Migration Locations in JHotDraw

- org.jhotdraw.applet
  - DrawApplet
    - createColorChoice(FigureAttributeConstant)
    - createFontChoice()
    - createButtons(JPanel)
    - setupAttributes()
  - DrawApplication
    - createColorMenu(String, FigureAttributeConstant)
    - createFontMenu()
    - createFontSizeMenu()
    - createLookAndFeelMenu()
    - checkCommandMenus()
    - checkCommandMenu(CommandMenu)
    - fireViewSelectionChangedEvent(DrawingView, DrawingView)
    - fireViewCreatedEvent(DrawingView)
    - fireViewDestroyingEvent(DrawingView)
- org.jhotdraw.contrib.dnd
  - JHDDropTargetListener
    - drop(DropTargetDropEvent)
  - DNDFigures
    - DNDFigures(FigureEnumeration, Point)

  - DragGestureListener
    - dragGestureRecognized(DragGestureEvent)
  - JHDSourceListener
    - dragDropEnd(DragSourceDropEvent)
  - RemoveUndoActivity
    - redo()
  - RemoveUndoActivity
    - release()
  - AddUndoActivity
    - undo()
  - AddUndoActivity
    - release()
  - DragNDropTool
    - createDragGestureListener()

- org.jhotdraw.contrib.html
  - HTMLContentProducer
    - getHTMLFontSizeEquivalent(int)
  - ContentProducerRegistry
    - getSuperClassContentProducer(Class, Class)
    - write(StorableOutput)
  - ETSLADisposalStrategy
    - dispose()
  - HTMLTextAreaFigure

        * getPolygon()

- org.jhotdraw.contrib.zoom
    - ZoomDrawingView
        * centralize(Drawing d, Dimension bounds)
- org.jhotdraw.contrib
    - CTXContextMenu
        * enable(String name, boolean state)
        * checkEnabled()
    - CTXCommandMenu
        * actionPerformed(ActionEvent e)
    - CTXWindowMenu
        * buildChildMenus()
    - DesktopEventService
        * removeComponent(DrawingView dv)
        * getDrawingViews(Component[] comps)
        * fireDrawingViewAddedEvent(DrawingView)
        * fireDrawingViewRemovedEvent(DrawingView)
        * fireDrawingViewSelectedEvent(DrawingView, DrawingView)
        * Helper getDrawingView(Container container)
    - MDIDesktopPane
        * removeFromDesktop(DrawingView dv, int location)
        * removeAllFromDesktop(int location)
        * getAllFromDesktop(int location)
        * cascadeFrames()
        * tileFramesHorizontally()

    - tileFramesVertically()
    - arrangeFramesHorizontally()
    - arrangeFramesVertically()
    - resizeDesktop()
- SplitPaneDesktop
    - removeFromDesktop(DrawingView dv, int location)
- TextAreaFigure
    - drawText(Graphics g, Rectangle displayBox)
- TriangleFigure
    - rotate(double)
- WindowMenu
    - buildChildMenus()
- ClippingUpdateStrategy
    - draw(Graphics, DrawingView)
- CustomToolbar
    - activateTools()
- SimpleLayouter
    - calculateLayout(Point, Point)
- StandardLayouter
    - calculateLayout(Point, Point)
    - layout(Point, Point)
- TextAreaToolUndoActivity
    - undo()
    - redo()
    - setText(String)

- org.jhotdraw.figures
  - LineConnection
    - basicMoveBy(int, int)
  - PolyLineFigure
    - joinSegments(int, int)
    - displayBox()
    - basicMoveBy(int, int)
    - write(StorableOutput)
  - ConnectedTextTool.UndoActivity
    - undo()
    - redo()
  - ConnectedTextTool.DeleteUndoActivity
    - undo()
    - redo()
  - FigureAttributes
    - write(StorableOutput)
  - FontSizeHandle.UndoActivity
    - swapFont()
  - GroupCommand.UndoActivity
    - undo()
  - GroupFigure
    - displayBox()
    - setAttribute(String, Object)
    - setAttribute(FigureAttributeConstant, Object)
  - TextTool.UndoActivity

        * setText(String)
    - UngroupCommand
        * isExecutableWithView()
    - UngroupCommand.UndoActivity
        * undo()
        * ungroupFigures()

- org.jhotdraw.framework
    - FigureAttributeConstant
        * getConstant(String)

- org.jhotdraw.javadraw
    - JavaDrawApp
        * createImagesMenu()

- org.jhotdraw.standard
    - CompositeFigure
        * figures(Rectangle)
        * addAll(FigureEnumeration)
        * removeAll(FigureEnumeration)
        * removeAll()
        * orphanAll(FigureEnumeration)
        * draw(Graphics, FigureEnumeration)
        * findFigure(int, int)
        * findFigure(Rectangle)
        * findFigureWithout(int, int, Figure)
        * findFigure(Rectangle, Figure)

    - findFigureInside(int, int)
    - findFigureInsideWithout(int, int, Figure)
    - includes(Figure)
    - basicMoveBy(int, int)
    - release()
    - write(StorableOutput)
    - readObject(ObjectInputStream)
    - init(Rectangle)
- QuadTree
    - getAllWithin(Rectangle2D)
    - addFigureEnumerationToList(List, FigureEnumeration)
- StandardDrawing
    - figureInvalidated(FigureChangeEvent)
    - fireDrawingTitleChanged()
    - figureRequestUpdate(FigureChangeEvent)
    - displayBox()
- StandardDrawingView
    - fireSelectionChanged()
    - drawPainters(Graphics, List)
    - addAll(Collection)
    - figureExists(Figure, FigureEnumeration)
    - insertFigures(FigureEnumeration, int, int, boolean)
    - getConnectionFigures(Figure)
    - addToSelectionAll(FigureEnumeration)
    - clearSelection()
    - selectionHandles()
    - findHandle(int, int)
    - moveSelection(int, int)
    - checkDamage()

    - drawHandles(Graphics)
    - getDrawingSize()
    - getMinimumSize()
- ToolButton
    - ToolButton(PaletteListener, String, String, Tool)
- AbstractCommand.EventDispatcher
    - fireCommandExecutedEvent()
    - fireCommandExecutableEvent()
    - fireCommandNotExecutableEvent()
- AbstractFigure
    - visit(FigureVisitor)
- AbstractTool.EventDispatcher
    - fireToolUsableEvent()
    - fireToolUnusableEvent()
    - fireToolActivatedEvent()
    - fireToolDeactivatedEvent()
    - fireToolEnabledEvent()
    - fireToolDisabledEvent()
- AlignCommand.UndoActivity
    - undo()
    - alignAffectedFigures(Alignment)
    - setAffectedFigures(FigureEnumeration)
- BringToFrontCommand
    - execute()
- ChangeAttributeCommand
    - execute()

- ChangeAttributeCommand.UndoActivity
  - undo()
  - redo()
  - setAffectedFigures(FigureEnumeration)
- ChangeConnectionHandle
  - findConnectableFigure(int, int, Drawing)
- ConnectionHandle
  - findConnectableFigure(int, int, Drawing)
- ConnectionTool
  - findConnection(int, int, Drawing)
  - findConnectableFigure(int, int, Drawing)
- ConnectionTool.UndoActivity
  - undo()
- CutCommand
  - execute()
- CutCommand.UndoActivity
  - rememberSelectedFigures(FigureEnumeration)
  - release()
- DeleteCommand
  - execute()
- DragTracker
  - mouseDrag(MouseEvent, int, int)
- DragTracker.UndoActivity
  - moveAffectedFigures(Point, Point)

  - FigureTransferCommand
    - deleteFigures(FigureEnumeration)
  - PasteCommand
    - getBounds(FigureEnumeration)
  - PasteCommand.UndoActivity
    - undo()
  - SelectAreaTracker
    - selectGroup(boolean)
  - SendToBackCommand
    - execute()
  - SendToBackCommand.UndoActivity
    - undo()
    - redo()
    - setAffectedFigures(FigureEnumeration)
  - StandardFigureSelection
    - StandardFigureSelection(FigureEnumeration, int)

- org.jhotdraw.test.contrib
  - CommandTextBoxMenuItemTest
    - testSetGetCommand()
  - CommandMenuItemText
    - testSetGetCommand()
  - GraphicalCompositeFigureTest
    - testSetGetPresentationFigure()
    - testSetGetLayouter()

  - SimpleLayouterTest
    - testSetGetLayoutable()
    - testSetGetInsets()
  - TextAreaFigureTest
    - testSetGetText()
    - testSetIsReadOnly()
    - testSetIsSizeDirty()
    - testSetGetFont()
    - testSetIsFontDirty()

- org.jhotdraw.test.figures
  - BorderDecoratorTest
    - testSetGetBorderOffset()
  - NumberTextFigureTest
    - testSetGetValue()
  - PolyLineFigureTest
    - testSetGetStartDecoration()
    - testSetGetEndDecoration()
  - TextFigureTest
    - testSetGetFont()
    - testSetGetText()

- org.jhotdraw.test.standard
  - NullDrawingViewTest
    - testSetGetDisplayUpdate()
    - testSetGetBackground()

  - StandardDrawingText
    - testSetGetTitle()
  - StandardDrawingViewTest
    - testSetGetDisplayUpdate()
    - testSetGetConstrainer()

- org.jhotdraw.test.util.collections
  - BoundsTest
    - testSetGetCenter()
  - IconKit
    - loadRegisteredImages(Component component)

- org.jhotdraw.test.util.collections.jdk11
  - SetWrapper
    - SetWrapper(Set)

- org.jhotdraw.test.util
  - ClipboardTest
    - testSetGetContents()
  - PaletteIconTest
    - createInstance()
  - StandardStorageFormatTest
    - testSetGetFileExtension()
    - testSetGetFileDescription()
    - testSetGetFileFilter()

  - StandardFormatManagerTest
    - testSetGetDefaultStorageFormat()
  - UndoableAdapterTest
    - testSetIsUndoable()
    - testSetIsRedoable()
    - testSetGetAffectedFigures()
  - UndoableToolTest
    - testSetIsUsable()
    - testSetIsEnabled()
  - UndoRedoActivityTest
    - testSetIsUndoable()
    - testSetGetAffectedFigures()

- org.jhotdraw.util
  - ColorMap
    - color(String)
    - colorIndex(Color)
  - CommandMenu
    - enable(String, boolean)
    - checkEnabled()
    - actionPerformed(ActionEvent)
  - PaletteLayout
    - minimumLayoutSize(Container)
    - layoutContainer(Container)
  - StandardVersionControlStrategy
    - assertCompatibleVersion()

    - handleIncompatibleVersions()
  - StorableOutput
    - writeString(String)
  - StorageFormatManager
    - registerFileFilters(JFileChooser)
    - findStorageFormat(FileFilter)
    - findStorageFormat(File)
  - GraphLayout
    - relax()
    - remove()
  - JDOSStorageFormat
    - retrieveAll(PersistenceManager, Figure)
  - JDOSStorageFormat.DrawingListModel
    - DrawingListModel(Iterator)
  - UndoableAdapter
    - rememberFigures(FigureEnumeration)
    - release()
  - UndoManager
    - clearUndos(DrawingView checkDV)
    - clearRedos(DrawingView checkDV)
  - VersionManagement
    - readVersionFromFile(String, String)

- org.jhotdraw.samples.javadraw
  - BouncingDrawing

        * animationStep()

- org.jhotdraw.samples.net
    - NodeFigure
        * drawConnectors(Graphics)
        * findConnector(int, int)
- org.jhotdraw.samples.offsetConnectors
    - NodeFigure
        * drawConnectors(Graphics g)
        * findConnector(int, int)
- org.jhotdraw.samples.pert
    - PertFigure
        * start()
        * layout()
        * needsLayout()
        * notifyPostTasks()
        * hasCycle(Figure)
        * writeTasks(StorableOutput, List)