Contents

List of Figures

List of Tables

Abstract

USING EVOLUTION PATTERNS TO FIND DUPLICATED BUGS

by

Kai Pan

The widespread use of software configuration management repositories to record the evolution of a software project raises the possibility of mining these repositories to better understand how developers fix software defects.

In the repository that records a software project's change history, there are many changes where developers fix bugs (known as bug fix changes) as opposed to adding new features or refactoring source code. Bug fixes are interesting, since they not only provide the source code of a bug, but also the source code for how the bug is fixed. This dissertation defines 27 static bug fix patterns, which are automatically extractable, based on the syntax components and context of the source code involved in bug fix changes. Instances of static bug fix patterns are extracted from the configuration management repositories of seven open source projects, all written in Java (ArgoUML, Columba, Eclipse, JEdit, Scarab, Lucene, and MegaMek). Defined static bug fix patterns cover 45.7% to 63.6% of the total bug fix changes in these projects. Two classes of static bug fix pattern instances, those related to if statements and method calls, together account for 44.6% to 60.3% of all observed static bug fix patterns. Several analyses were performed on the extracted pattern instances on the projects.

This dissertation also presents a bug finding algorithm, BugMem, to find duplicated bugs using bug fix memories: a project-specific bug and fix knowledge base developed by analyzing the history of bug fixes. This approach is a learning process, hence identified bug patterns are project-specific, the number of bug patterns grows as the software evolves, and high-level project-specific bugs can be detected. The algorithm and tool were assessed by evaluating if duplicated bugs and fixes in project histories could be found in the bug fix memories. Analysis of five open source projects (ArgoUML, Columba, Eclipse, JEdit, and Scarab) shows that, for these projects, 17.5% - 32.4% of bugs appear repeatedly in the memories, and 7.5% - 13.0% of bug and fix pairs are found in memories. The results demonstrate that project-specific bug fix patterns occur frequently enough to be useful as a bug detection technique. Furthermore, for the bug and fix pairs, it is possible to both detect the bug and provide a strong suggestion for the fix.

Acknowledgements

I would like to thank the many people that have helped me on the path towards this dissertation.

I want to express my appreciation and thanks to my dear advisor, Prof. Jim Whitehead. I am very lucky to have Jim as my advisor, who has been a faithful source of wisdom, insight, support, and encouragement to me and to my work. I still remember the first time I met him in his office and he explained his NSF proposal to me. That was the start of my research experience. Not limited to doing researching, I learned a lot from him in the past five years, the attitude towards work and life, the experience of organizing a technical activity or a conference, how to enjoy American literature, the way to use humor to make a class lecture more attractive to students, and so on.

I am extremely grateful to my good friend and colleague, Sung Kim. Sung's help with my research on software evolution analysis is tremendous. Without him, this dissertation would not have been possible. We cooperated on many projects and I would love to work with him anytime in the future: it would be my honor. Sung is also a good friend to me in my life. He and his wife, Yeon Gyoung Gwack, are just like family to me.

My sincere thanks go to Guozheng Ge for being both a good friend and a supporter of my research. Guozheng always keeps me updated on all kinds of new technology in computer science. His suggestion to look into the patterns in software bugs led to my PhD work exploring bug fix patterns.

I also would like to thank the other members in the GRASE lab, Jennifer Bevan, Mark Slater, and Elias Sinderson. Jennifer's pioneering work on software evolution analysis in our lab offered me the great opportunity to work in the area. Mark's kindness, easy-going style, and humor make me feel that I were not living in a foreign country. Elias is also a nice and easy-going friend. I won't forget the quarter I worked on the Catacomb project with he and Sung. With his significant contribution, this project turned out to be a success.

At the same time, I would like to thank Professor Charlie McDowell for giving me the opportunity to TA two of his courses. While I was doing my TA work for him, I also was learning from his lectures, his teaching and the textbooks he wrote. It was a precious experience for me. I also thank Professor Yi Zhang for reviewing my dissertation. I audited her Information Retrieval class and obtained much useful knowledge from it. I wish I had had the chance to take her courses earlier.

I am thankful to my dear friends at Santa Cruz, Zhenjiang Li, Caixue Lin, Qin Xin, Xiaojin Shi, Fangyi Luo, Yue Zhou, Gang Wang, Yong Liu, Wei Wei, Zhongping Zhao, Ge Yang, Feng Wang, Yu Wang, Shoufeng Yang, Yan Zhang, and Yongtao Lu. My life would be tough without their friendship.

Finally, I thank my grandma, my parents and my brother for their love and support, without which I would never have arrived here in the first place.

To my grandma and my parents

1 Introduction

Change is inevitable in a large software system. Developers make changes to the source code for various reasons, such as performance improvement, code cleanup, adding new features, fixing bugs, code formatting, improving compatibility etc. Pressman classifies software changes into four categories, correction, adaptation, enhancement, and prevention [75]. Corrective changes are those performed to fix bugs in software. Adaptive changes are software modifications caused by changes in the external environment. Enhancement changes are made when additional requirements extend the functionality of the software. Preventive changes are performed when software reengineering is needed to prevent the deterioration of software.

Bug fix changes are a major class of program changes, accounting for 20 percent of all maintenance work [75]. Our analysis on the change history of several open source projects in this dissertation also shows that 18% to 47% of revisions are bug fix revisions. We are interested in bug fix changes, since they identify problematic code (the lines from the previous revision that were changed), and how to solve the problem (the modifications made to these lines).

A long-developed project usually has a software configuration management (SCM) repository that records a great number of bug fix changes. A typical bug fix process is as follows. A developer receives a bug report or modification request, or identifies a bug in the source code during code inspection. She changes the code to fix the bug, and submits the changes to the SCM repository with a change log message that describes the changes. The new code is then tested or inspected. The bug fix process repeats when new bugs are identified, the result being that bug fix changes and change logs keep accumulating in the repository. After developers submit a bug fix change, they may forget the previous mistakes made by themselves or peer developers, and commit the same mistakes again in the future.

Since bug fix changes in an SCM repository record change history information about previous development experience in the project, it would be nice if we could take advantage of this previous development experience.

In practical life, we learn lessons from previous mistakes and avoid committing the same mistakes again. A car accident due to a driver's speeding will teach the driver to drive more defensively in the future; last year's suffering from flu makes you take a flu shot this year; the experience of being burnt by hot water will make you more cautious when handling it. In software development, it would be nice if we could learn from our mistakes in the same way.

If a developer could look back at the bug fix changes she or other developers have made, she may learn from this previous bug fix experience that would allow her to avoid making similar mistakes again, and find the correct way to write the code. But, since there are so many bug fix changes in the repository and the size of the source code is huge, it is not possible for developers to manually inspect all previous bug fix changes to find something useful. So, if we can find a way to automatically summarize some commonality in the previous bug fix changes, it will help developers understand existing bugs made by themselves or other developers, and learn from previous mistakes.

There are two major goals of this dissertation. First, to explore patterns in bug fixes by investigating the bug fix changes in a software project's change history. Second, to detect duplicated software bugs by mining bug fix changes in software change repositories. Achieving these goals can help developers understand previous bugs in a project and take advantage of previous bug fix experience. In particular, inexperienced developers and new project members can quickly learn from the bug fix change history to avoid committing similar mistakes, and find a correct way to write code by learning from the examples in the fixed code.

1.1 Understanding of Bug Fix Changes and Bug Fix Patterns

A software configure management (SCM) system, such as CVS [20] or Subversion [81], uses GNU diff [27] to determine the textual difference between two versions of text files. GNU diff calculates the longest common subsequence (LCS) of two text files at the line level. The text lines that are not in the longest common subsequence are the difference of the two file versions. There are many advanced approaches [2, 30, 35, 43, 84, 87] that compute program differences in ways that carry more structural or semantic information, but GNU diff is favored in practical usage, since it is widely used, free, cheap in computation, and language neutral. In fact, diff has become a de facto standard to compute and visualize program differences, and deliver software patches.

The language-neutral semantic-free properties of textual differences are not favorable to software evolution analysis, since many textual differences look random for source code change understanding. These facts prevent people from understanding properties in previous changes, especially bug fix changes, since manual inspection of bug fix changes can only allow a limited number of revisions to be inspected and analyzed. There are a number of research efforts [13, 54, 69, 73] that investigate software maintenance activities to classify software faults, and explore their properties and causality. But these approaches usually employ manual inspection of software change activities, and hence do not scale.

One goal of this work is to perform software evolution analysis on a project's bug fix change history to reveal the properties of bug fix changes in projects and explore patterns in bug fixes. To overcome the shortcomings of textual difference, we developed an analyzer to parse and analyze source code changes between two versions and extract syntactic or structural information from them.

In order to learn from bug fix changes, it is necessary to abstract away the project-specific details of each change, yielding more general bug fix patterns. This permits cross-project comparison of bug fix data. For example, the specific variables involved in the modification of an *if* statement condition are unimportant, since these will vary substantially within and across projects. By abstracting away the details concerning individual variables and recording just that there was an instance of the if-condition pattern, it is now possible to characterize and compare if-condition bug fix patterns both within and across projects.

We defined 27 automatically extractable bug fix patterns based on the syntax components and context of the source code involved in bug fix changes. These patterns are based on the syntax components and context of the source code involved in bug fix changes. Patterns were initially identified from a manual analysis of the bug fixes in open source Java projects. Subsequently, a bug fix pattern extraction tool was developed that could automatically identify the bug fix patterns.

Bug fix patterns are syntax-driven, and hence are very close to the source code. The resulting categories are highly tied to language features. This characteristic of the bug fix patterns allows for automatic extraction. Results from the analysis can answer questions like, what kinds of bugs does a project have the most, how to classify bug fixes, are there any meaningful patterns in the bug fixes, what kinds of bugs did a developer introduce the most, etc.

1.2 Finding Duplicated Bugs using Bug Fix Change History

Since bug fix changes in the software change repository record previous faults (mistakes) in the source code, they can be used to prevent similar mistakes. A software change repository is generally accessible to every developer, but several factors prevent bug fix changes from being fully exploited for automatic fault prevention.

1. It takes effort to differentiate bug fix changes from non-fix changes.
2. Too many revisions in the change repository.
3. Too many files in a project.
4. Textual difference representation is not favorable for conveying syntax and semantic information, i.e. changes look random.
5. Noise in bug fix changes, such as comments, blanks, format changes, etc.

Due to these factors, developers fail to make use of them fully: they may omit their previous mistakes or mistakes made by peer developers, and commit the same mistakes again.

It would be nice to have a good method to collect mistakes so that developers can avoid repeating these bugs in the future. Static bug finding tools achieve this goal in a horizontal way. Many automatic bug finding tools have been proposed, including Bandera [17], ESC/Java [26], FindBugs [39], JLint [4, 5], and PMD [16]. They use a range of techniques to detect bugs and suggest fixes, including pre-defined bug patterns [5, 39], type checking [26], and model-checking [17]. These bug finding tools adopt a horizontal approach, using techniques that are applicable across all projects. To date, there are very few tools using the vertical approach of leveraging patterns in a specific project and performing project-specific bug finding.

This dissertation presents a vertical bug finding approach, BugMem, which extracts and memorizes a broad range of patterns in buggy code and uses the previous bug patterns of a specific project to find project-specific bugs in new changes or other parts of the source code. We provide a series of solutions to address issues in making use of bug fix changes. These solutions include automatically discovering bug fix changes, eliminating noise in textual differences, generating normalized syntax information for source code, extracting syntax components from textual differences between bug and fix versions, and performing pattern matching on syntax components to discover duplicated bugs.

The BugMem approach has an advantage over existing bug prediction approaches [33, 46, 47, 53, 67, 68, 70, 71] using software quality metrics or machine learning in that it can locate bugs at very small granularity, the source code line level, as compared with the function level, file level, or subsystem level supported in other bug prediction approaches. Another advantage of BugMem is that it provides a fix suggestion for each bug warning, which allows users to reject or accept a bug warning easily. Bug prediction approaches using machine learning only provide a possibility that a section of code or a file may contain a bug or a possible number of bugs, with the drawback that developers can not easily figure out why the code is suspicious and how it may be corrected.

Generally, the BugMem approach aims to find high-level project-specific bugs with a vertical approach. It can serve as a nice complement to the bug finding tools using static analysis approaches and bug prediction tools using machine learning.

2 Terminology

2.1 Version Control and Program Change

This dissertation uses several terms related to version control and program changes, defined as follows.

Version. A version is a state of an object at one point in the history of its evolution.

Revision. Revision is interchangeable with version, i.e. revision *n* indicates the n_{th} version. The term *revision* additionally puts more emphasis on the difference between a version and its previous version. That is, a revision indicates all the changes made to the previous version to generate the current version. The version before the changes is called an *old version*, and the version after a revision is called the *new version*.

Version Control. Version control is a basic function supported by a software configuration management (SCM) system. Generally, the goal of version control is to keep track of all the changes made to files.

Hunk, deleted hunk, added hunk. The textual difference of a file between an old version and its new version as computed by a diff tool is represented by a list of text regions, called *hunks*. A hunk is a series of contiguous changed lines and can be empty. A hunk in the old version is called a *deleted hunk*, and a hunk in the new version is called an *added hunk*.

Hunk Pair or delta. A deleted hunk always has its corresponding added hunk, and we call a deleted hunk and its corresponding added hunk a hunk pair (or delta). There are three kinds of hunk pairs: modification, addition, or deletion. If neither the deleted hunk nor the added hunk of a hunk pair is empty, we call this a modification hunk pair. A deletion hunk pair has an empty added hunk, and an addition hunk pair has an empty deleted hunk. Figure 2-1 illustrates the three kinds of hunk pairs.

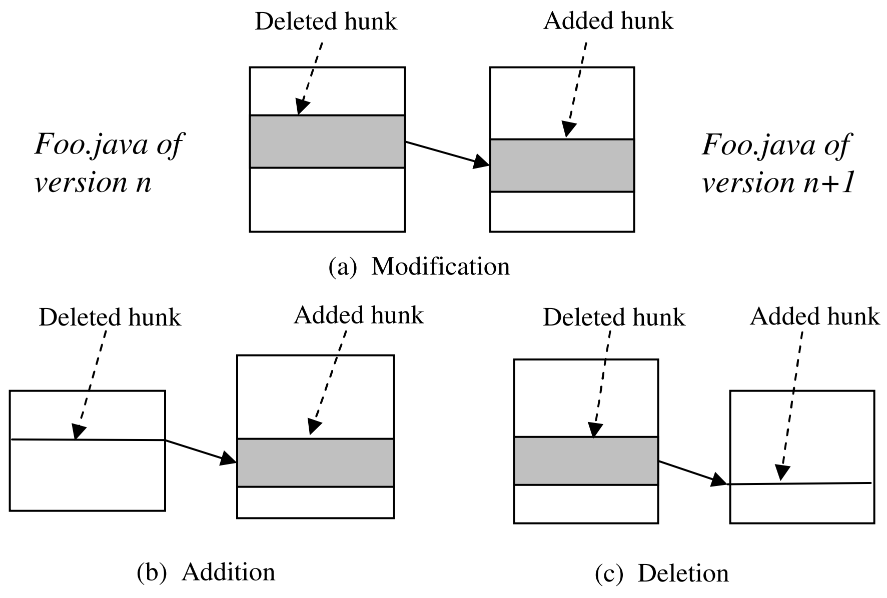


Figure 2-1. Three kinds of hunk pairs.

2.2 Bug and Fix

Bug. In this dissertation, the term *bug* describes a mistake made in software source code. It is a programmer error that manifests itself in the form of incorrect source code. Bug is used instead of the equivalent term fault since it is more colorful, and is in widespread professional use.

Fix or Bug Fix. A change to the source code to eliminate a bug.

Bug Fix Revision. A project revision containing changes to repair buggy code is a *bug fix revision*. A revision can be either a bug fix revision or a *non-fix revision*.

Bug Version and Fix Version. The old version of a bug fix revision is called a *bug version*, and the new version of a bug fix revision is called a *fix version*.

Bug Hunk and Fix Hunk. A hunk in a bug version is called a *bug hunk*, and a hunk in a fix version is called a *fix hunk*.

Bug Fix Hunk Pair. A hunk pair in a bug fix revision is a *bug fix hunk pair*. That is, a bug fix hunk pair consists of a bug hunk and its corresponding fix hunk.

Buggy code. The code in a bug hunk is called buggy code.

Fix Code. The code in a fix hunk is called fix code.

In general software engineering usage, the concept of bug or fault has broader meaning, which can represent an abnormality in specification, design, code, test-case, etc. Since this dissertation only focuses on bugs that appear in source code, we narrow the definition of bug.

2.3 Source Code Parsing and Analysis

Hunk pairs or deltas computed by diff only show the textual difference of a file between two versions. To understand the changes represented by these hunk pairs, we have to parse the source code, and analyze the code in the hunks and the code nearby. The following terms are frequently used in describing the source code parsing and analysis.

Define, defined variable. If a variable in a statement may receive a new value after the statement executes, we say this statement *define*s the variable, and the variable is a *defined variable* in the statement. For example, the statement *foo = bar + 1;* defines the variable *foo*, and *foo* is a defined variable in the statement.

Use, used variable. If a variable appears in a statement but is not defined, or a variable is involved in computing the value of a defined variable in the statement, we say this statement *uses* the variable, and the variable is a *used variable* in the statement. For example, in the statement *foo = bar + i;*, the used variables include *bar* and *i*. More about variable definition and use and data dependence can be found in [45, 58].

Syntax line. A syntax line represents a line of source code after we format a source code file in a uniform way. The reason that we format a source code file in a uniform way is to get rid of the noise in the textual difference for a file between different versions, which will be discussed in Section 3.2. Generally, to perform the formatting process, we begin by preprocessing the code to remove all whitespace and blank lines. We differentiate between composite statements such as *if*, *for*, *while*, etc. and those that are simple such as method calls, assignment, etc. We further process the code to concatenate all multi-line simple statements into a single line. Additionally, we process the code to ensure that the conditional predicates of *if*, *for*, *while*, etc. all lie on a single line. This yields basic *syntax lines*.

Syntax component. A syntax component is a fine-grained syntax segment in a syntax line. More specifically, a syntax component corresponds to a non-leaf node in a syntax line's abstract syntax tree. A syntax component can be further decomposed into raw syntax components and normalized syntax components. We use syntax component and normalized syntax component interchangeably in places where there is no confusion. The detailed description of syntax component can be found in Section 5.1.2.

Context or syntax context. We use context to represent the environment a statement is in, i.e. the nearby statements that are related to this statement, and the syntax construct, such as *if, while, try*, etc., the statement is in. The context information will help us to find the patterns in bug fix hunk pairs.

3 Fact Extraction from Software Change Repository

In both the bug fix pattern analysis and the BugMem analysis, we need to extract versions and changes from the change repository of a project, and perform analysis on every file and change in a revision. To assist the analysis, we pre-process every revision in the software change repository for a project, extract facts from it, and transfer the extracted facts to a database. In this chapter, we describe the architecture of the fact extraction framework, the steps to pre-process the revision data, and the schema of the database that stores the extracted facts.

3.1 Architecture

To analyze software evolution, we need to extract revisions from the SCM system that stores the change history of a project. In our project, we use the Kenyon infrastructure to extract revisions from SCM repositories [9]. Kenyon is a system that provides access facilities to various SCM repositories such as CVS [20], SubVersion [81] and ClearCase [40], and automates the process of extracting revisions from a software configuration management system repository. The architecture and data flow in the fact extraction framework is depicted in Figure 3-1.

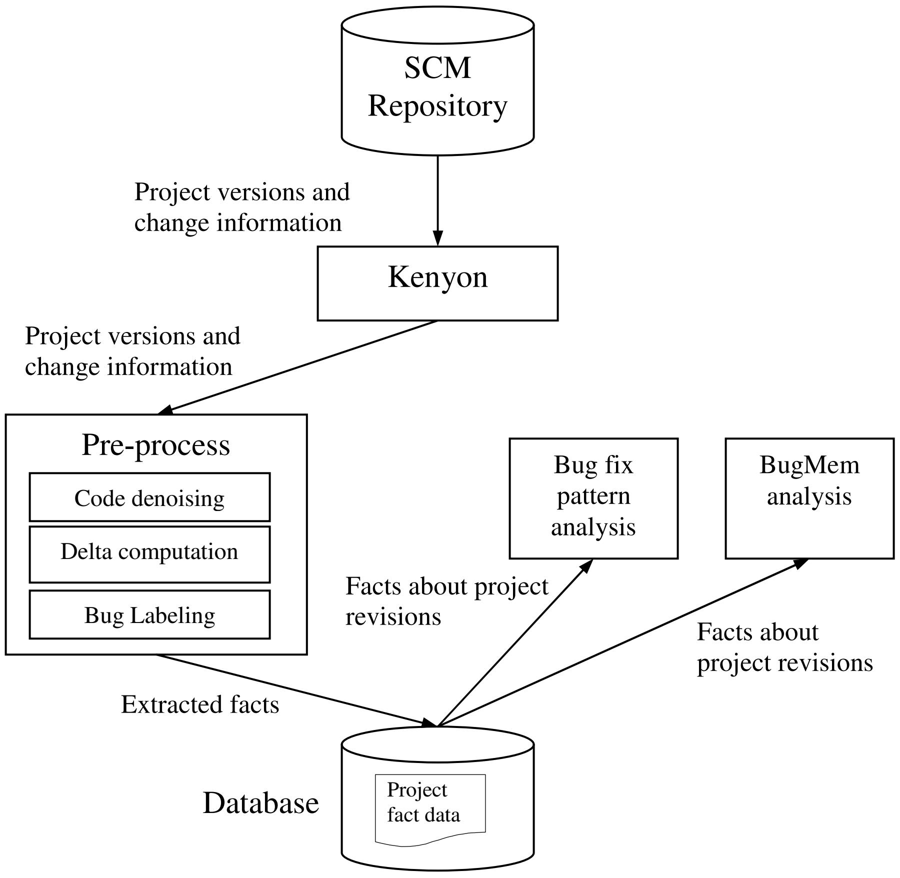


Figure 3-1. Architecture and data flow for fact extraction.

In our project, Kenyon is responsible for extracting project source versions from a repository of a project under analysis. Kenyon automatically checks out the source code of each revision and extracts change information such as the change log message, author, change date, source code, etc.

The extracted source code versions and change information are fed to a pre-processing module, which contains three sub-modules for code denoising, delta computation, and bug labeling. The code denoising module transforms the source code into a uniform format to eliminate formatting and whitespace noise in the source code, such as blanks, blank lines, comments, different code formats, etc. The denoised source code is favorable for computing meaningful textual difference between two versions of a source code file, since many textual changes that cause no behavioral changes will be eliminated from the results of diff. The bug labeling module differentiates bug fix revisions from non-fix revisions. These three modules are explained in detail in the following subsections. Finally, the extracted facts for the project revisions are saved to the *project fact data* database.

3.2 Code Denoising

In our analysis, we only care about those changes in the source code that will cause the program to behave differently. We note that there are many source code changes that will not modify the behavior at all, for example, addition/removal of blanks, addition/removal of blank lines, changes to comments, and different source code formatting. Figure 3-2 shows an example of two source code segments that have the same behavior but different source code format. This kind of textual difference should be ignored in our analysis. The existence of the non-behavior altering changes adds noise to the results of textual difference between two versions, and affects the quality of the bug fix pattern analysis and the BugMem analysis.

```
- if (foo != null)   { foo.bar(); }
+ if (foo != null) {
+    foo.bar();
+}
```

Figure 3-2. An example of different source code format with the same behavior.

To address this issue, we reformat the source code file versions before textual difference computation. More specifically, before we compute the textual difference of a Java source code file (the projects analyzed for this dissertation are all in Java) between different versions, we use a JDT (Java Development Tools) flattener [22] to transform the source code files into a uniform format.

```
if (  foo != null
        && foo.ready)
{
    /* This is an
        example. */
    foo.ret = foo.bar(
                1, 2);

        foo.start();
}
```

```
if (foo != null && foo.ready) {
    foo.ret = foo.bar(1,2);
    foo.start();
}
```

Figure 3-3. An example of code segment and its denoised version.

The transformation works in this way: first it parses the entire source code file and obtains an abstract syntax tree (AST) for it, and then it outputs the AST in a uniform format. This process eliminates all the unnecessary blanks, removes blank lines and comments, and indents the source code in a uniform way. Figure 3-3 shows an example code segment and the corresponding code after the denoising process.

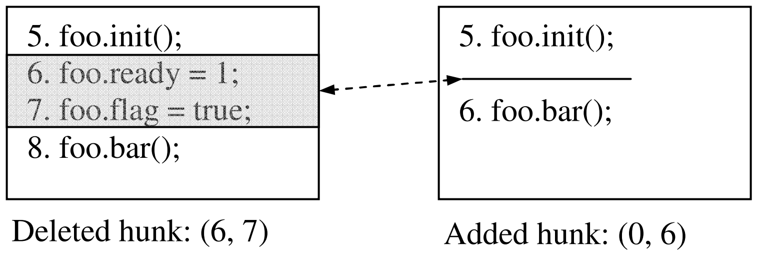
After we perform the denoising process on the source code files, we save them to the database. The delta computation of the files in two versions will be carried out on the denoised source code files.

3.3 Delta Computation

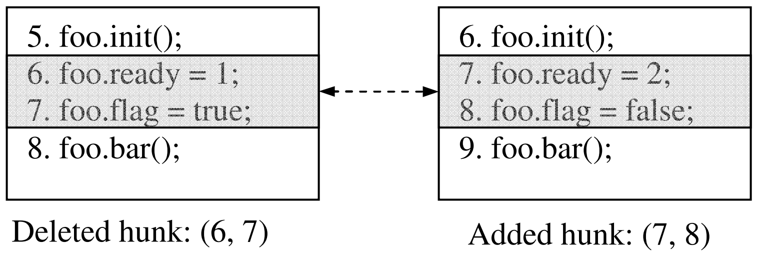
To explore how software evolves, we need to compute the changes to project files across versions. The delta computation module is used for this purpose, which computes deltas (hunk pairs) of a file between two versions. Since delta computation is performed on denoised source code versions, most of the noise in textual differences has already been removed and only meaningful deltas remain.

The deltas are computed using the diff tool [27]. A list of hunk pairs, which represents the regions that differ in a file in two versions, is returned from the delta computation module. We record a hunk for a file by storing the starting line number and the ending line number of the hunk in the file. When a hunk is empty, we set the starting line of the empty hunk to be 0 and set the ending line of the hunk to be the line number before which the hunk is deleted from the other version. Figure 3-4 shows an example of how to record hunk information, e.g. Figure 3-4

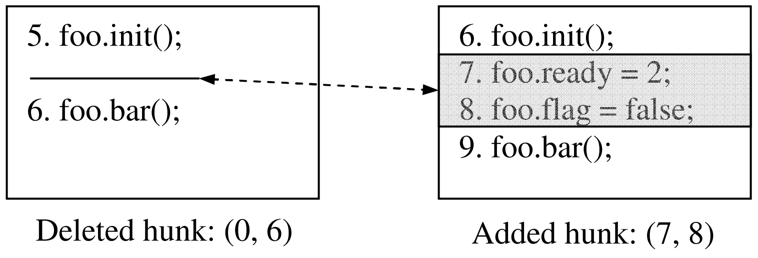
(a) shows a deletion hunk pair, where the deleted hunk is not empty, and the added hunk is empty. The hunk pair information is saved into the database.



(a) deletion hunk pair



(b) modification hunk pair



(c) addition hunk pair

Figure 3-4. An example of how to record hunk information.

3.4 Bug Labeling

Since our analysis focuses on bug fix revisions, we need to differentiate bug fix revisions from non-fix revisions.

Traditionally, bugs are identified in software by examining test executions for incorrect output, performing software inspections, or running static analysis tools. Our method for bug identification is somewhat different, in that we assume that developers have been using these traditional methods for bug identification throughout a project's evolution, and have been fixing the buggy code. Hence, our task is to retrospectively discover which changes involved bug fixes, and which ones performed other kinds of software update.

A project revision containing changes to repair buggy code is a bug fix revision. The version before the bug fix revision is the bug version, and the version after the bug fix revision is the fix version. We identify bug fix revisions based on the log messages that are supplied with a revision. There are two approaches for identifying bug fix revisions. The first one is to look for keywords like "fix" or "bug" in the change log, a technique introduced by Mockus and Votta [63]; the second approach is looking for references to bug reports like "#42233" as introduced by Fischer et al. [25] and by Cubranic and Murphy [18]. We use the first approach in this work. That is, if a change log message contains "bug", "fix", or "patch", the revision is found to be a bug fix revision. In a bug fix revision, a hunk pair is labeled as a bug fix hunk pair, a hunk in the bug version is labeled as a bug hunk, and a hunk in the fix version is labeled as a fix hunk.

3.5 Database Schema

We store the facts extracted from a project's SCM repository into a MySQL database. After that, software evolution analysis can be performed by searching and retrieving data from the database instead of extracting data directly from the SCM repository, which only obtains raw data and is slow for data retrieval.

There are four database tables for the project fact data, Revision, File, Delta, and Content. Each record in the Revision table stores the information for a revision in a project, such as the author that committed the changes and the log message attached to the changes. The File table records the information of the files contained in a revision such as the name of the file, *file_name*, and whether the file is changed in this revision, *is_modified*. We record the information for all the files including unmodified files and modified files in the File table. The design choice of keeping the information for unmodified files is in order to fast locate the content of a file in two consecutive versions and compute their difference. There is not much redundancy caused by this design since the content of the file is not stored in the File table, but a reference to a record in the Content table. The Delta table stores the hunk pairs of a file of this revision computed against the file in the previous revision. These four tables represent the hierarchy relationship of the entities in the source code changes in a project's change history. Figure 3-5 illustrates the entity-relationship (ER) model of the four database tables: one revision (Revision) of a project contains many files, and one file of a version may contain many deltas with respect to the previous version, and the content of the file. Tables 3-1 to 3-4 provide the detailed schema information for the four database tables.

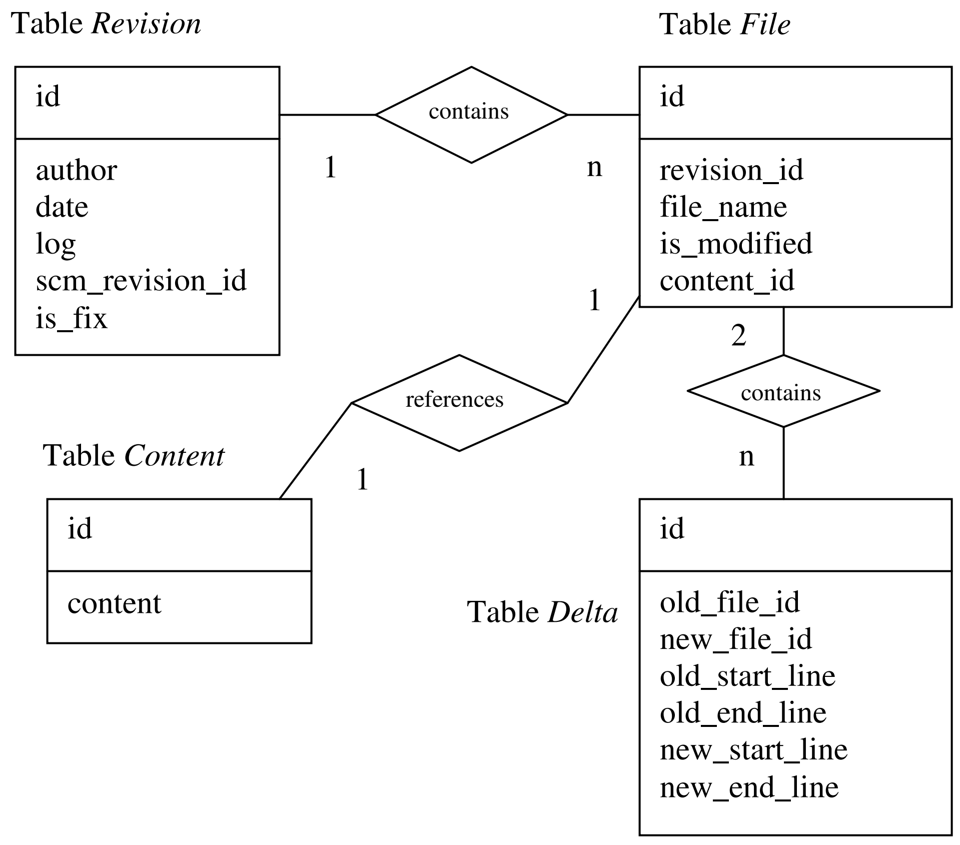


Figure 3-5. ER diagram of the database schema for the project fact data.

Field	Type	Description
id	integer, not null, PRIMARY KEY, AUTO_INCREMENT	The serial number of the revision record in the table.
author	varchar(32), not null	The username of the developer that commits the changes for this revision.
date	datetime	The date and time the revision is submitted
log	text	The change log supplied with the revision.
scm_revision_id	varchar(32)	The version number in the SCM repository
is_fix	boolean	Is this revision a bug fix revision?

Table 3-1. Database Schema of the *Revision* table

Field	Type	Description
id	integer, not null, PRIMARY KEY, AUTO_INCREMENT	The serial number of the file record in the table.
revision_id	integer, not null	The id of the revision record that contains this file.
file_name	varchar(255), not null	The file name of the file in the project.
is_modified	boolean	Is this file modified in the revision?
content_id	Integer,	The id of the content record that contains the text of the file.

Table 3-2. Database schema of the *File* table

Field	Type	Description
id	integer, not null, PRIMARY KEY, AUTO_INCREMENT	The serial number of the delta record in the table.
old_file_id	integer, not null	The id of the file record of the old version.
new_file_id	integer, not null	The id of the file record of the new version.
old_start_line	integer, not null	The line number of the start line of the deleted hunk in the old version.
old_end_line	integer, not null	The line number of the end line of the deleted hunk in the old version.
new_start_line	integer, not null	The line number of the start line of the added hunk in the new version.
new_end_line	integer, not null	The line number of the end line of the added hunk in the new version.

Table 3-3. Database schema of the *delta* table

Field	Type	Description
id	integer, not null, PRIMARY KEY, AUTO_INCREMENT	The serial number of the content record in the table.
content	longtext	The text content of the source code file after the denoising process.

Table 3-4. Database schema of the *Content* table

3.6 Analyzed Projects

We built a project fact database for seven open source projects written in Java, ArgoUML [3], Columba [15], Eclipse [21], JEdit [44], Scarab [77], Lucene [1], and MegaMek [61]. The seven projects are studied in the bug fix pattern analysis, while only the first five projects are studied in the BugMem analysis. Table 3-5 summarizes the seven projects, and we briefly describe them as follows.

Project	**Software type**	**Period**	**# of revisions**	**# bug fix revisions**
ArgoUML	UML editor	01/1998 ~ 09/2005	4,685	1,310
Columba	Mail client	11/2002 ~ 12/2005	2,362	797
Eclipse	IDE	06/2001 ~ 01/2006	6,394	2,807
JEdit	Editor	09/2001 ~ 01/2006	1,190	557
Scarab	Issue tracker	12/2000 ~ 02/2006	2,962	535
Lucene	Search engine	09/2001 ~ 02/2006	1042	266
MegaMek	Online game	02/2002 ~ 09/2006	2825	706

Table 3-5. Analyzed projects (For the Eclipse project we use only the core.jdt module due to the large size of the entire project).

ArgoUML (http://argouml.tigris.org/). ArgoUML is a UML diagramming application that includes support for all standard UML 1.4 diagrams such as Class, State, Use Case, Activity, Collaboration, Deployment and Sequence.

Columba (http://columba.sourceforge.net/). Columba is a mail user agent. Columba provides a user-friendly graphical interface with wizards and internationalization support.

Eclipse (http://www.eclipse.org/). Eclipse is an open source platform-independent integrated development environment (IDE) for Java. Eclipse framework has a design that allows third-party plugins to be easily integrated into the environment. Separate plugins for C/C++, FORTRAN, PHP, and Perl have been created. The Eclipse framework also supports adding any desired extensions to the environment, such as Subversion support.

JEdit (http://www.jedit.org/). JEdit is a text editor for programmers. JEdit has an extensible plugin architecture, and supports macros written in BeanShell, Jython and JavaScript.

Scarab (http://scarab.tigris.org/). Scarab is a highly customizable bug tracking application. The major features supported by Scarab include data entry, queries, reports, notifications to interested parties, collaborative accumulation of comments, and dependency tracking.

Lucene (http://lucene.apache.org/). Lucene is a text search engine library.

MegaMek (http://megamek.sourceforge.net/). MegaMek is an open-source, online version of the Classic BattleTech board game. Currently, MegaMek supports nearly all level 1 BattleTech rules, most level 2 rules, and a number of level 3 and house rules.

4 Patterns in Bug Fixes

In current practice, an analysis of the frequency of various error types in a software project involves software engineers manually identifying errors and assigning them to categories that are organized around error causes. This is a labor-intensive practice that requires a disciplined development team. As a result, this data is unavailable for most software projects. To perform software error analysis on these less disciplined projects requires some form of automated technique for identifying and classifying software errors.

This dissertation defines 27 static bug fix patterns in Java software. These patterns are based on the syntax components and context of the source code involved in bug fix changes. Patterns were initially identified from a manual analysis of the bug fixes in five open source Java projects. Subsequently, a bug fix pattern extraction tool was developed that could automatically identify the static bug fix patterns. This was run on the change history of seven open source projects, the original five used in the manual analysis, plus two more.

Static bug fix patterns are syntax-driven, and hence allow for automatic extraction. As an analytical tool, static bug fix patterns have several beneficial qualities. Since they can be automatically detected, it is possible to gather bug fix statistics from multiple projects. With this data in hand, it is possible to characterize the frequency of bug fixes made to specific programming language features (such as an *if* statement), fine grain program structures (loops, switch statements), or small sequences of operations.

Static bug fix patterns provide improved visibility into how bugs are being repaired, a visibility that takes the form of a frequency distribution of static bug fix patterns. This allows us to clearly see which programming language features are the most bug prone.

When we identify buggy code areas by mining bug fix revisions, we declare the source code that falls into bug hunks to be buggy code. But, this approach has limitations, since bug fix hunk computation is based on textual differences, while the dependence among statements is not considered. The bug fix pattern approach provides a way to address this problem by discovering buggy code missed by diff. For example, the statement *foo.bar = 1* in Figure 4-1 is not discovered as buggy code using diff, since the bug hunk is empty. But this statement is in fact buggy code and can be discovered by bug fix patterns.

Static bug fix patterns assign syntax context information to buggy code and fix code to assist in finding bugs using evolution patterns, discussed in Section 5. In the rest of this dissertation, we use the term static bug fix pattern and bug fix pattern interchangeably when it will not lead to confusion.

foo.bar = 1;
	if (foo != null)
	    foo.bar = 1;

Figure 4-1. An example bug fix in which the bug hunk is empty.

4.1 Bug Fix Patterns

Bug fix hunk pairs contain just the text difference between a section of code in the bug version and the corresponding code in the fix version. Some of them look random to us, but the others are not. We can sense some patterns in the changes from the changed syntax components in the bug fix hunk pair, the context the hunk code is in, and the code near the hunks.

To define a set of bug fix patterns, we manually analyzed part of the bug fix change history of five open source projects in Table 3-5, ArgoUML, Columba, Eclipse, JEdit, and Scarab. The analysis involved inspecting the bug hunks and the corresponding fix hunks in the bug fix revisions, and classifying bug fix changes into different patterns based on the syntax component kinds in the hunk pairs and the program context the syntax components are in. These identified bug fix patterns are grouped into several categories including If-related (IF), Method Call (MC), Loop (LP), Assignment (AS), Switch (SW), Try (TY), Method Declaration (MD), Sequence (SQ), and Class Field (CF).

Table 4-1 presents an overview list of bug fix patterns. In the following subsections, we present a catalog of observed bug fix patterns. Each pattern begins with a title, and an abbreviation code. A brief description is given of the pattern, followed by one or more source code examples taken from the examined projects. Example code with a leading "–" is from the bug hunk, while code with leading

"+" is in the fix hunk. Unmarked code (without leading "–" or "+") is found in both bug and fix versions.

Category	Pattern Name	Short Name
Assignment (AS)	Change of assignment expression	AS-CE
Class Field (CF)	Addition of a class field	CF-ADD
	Change of class field declaration	CF-CHG
	Removal of a class field	CF-RMV
If-related (IF)	Addition of an else branch	IF-ABR
	Addition of precondition check	IF-APC
	Addition of precondition check with jump	IF-APCJ
	Addition of post-condition check	IF-APTC
	Change of if condition expression	IF-CC
	Removal of an else branch	IF-RBR
	Removal of an if predicate	IF-RMV
Loop (LP)	Change of loop condition	LP-CC
	Change of the expression that modifies the loop variable	LP-CE
Method Call (MC)	Method call with different actual parameter values	MC-DAP
	Different method call to a class instance	MC-DM
	Method call with different number or types of parameters	MC-DNP
Method Declaration (MD)	Addition of a method declaration	MD-ADD
	Change of method declaration	MD-CHG
	Removal of a method declaration	MD-RMV
Sequence (SQ)	Addition of operations in an operation sequence of field settings	SQ-AFO
	Addition of operations in an operation sequence of method calls to an object	SQ-AMO
	Addition or removal method call operations in a short construct body	SQ-AROB
	Removal of operations from an operation sequence of field settings	SQ-RFO
	Removal of operations from an operation sequence of method calls to an object	SQ-RMO
Switch (SW)	Addition/removal of switch branch	SW-ARSB
Try (TY)	Addition/removal of a catch block	TY-ARCB
	Addition/removal of a try statement	TY-ARTC

Table 4-1. Bug fix patterns.

4.1.1 If-related (IF)

1. Addition of precondition check (IF-APC)

This bug fix adds an *if* predicate to ensure a precondition is met before an object is accessed or an operation is performed. Without the precondition check, there may be a NullPonterException error or an invalid operation execution caused by the buggy code. This kind of bug occurs when the developer did not consider the precondition before an operation is performed.

Example:

```
– lastChunk.init(seg,expander,x,styles,
–           fontRenderContext, context.rules.getDefault());
+ if (!lastChunk.initialized)
+       lastChunk.init(seg,expander,x,styles,
+           fontRenderContext,  context.rules.getDefault());
```

2. Addition of precondition check with jump (IF-APCJ)

This bug fix pattern is similar to the previous one, IF-APC, except it adds an *if* statement that encloses a jump statement, such as return, continue, or break. This causes the fixed code to skip the remaining code in the block if the precondition is not satisfied.

Example:

```
+ if (!comp.isShowing())     return true;
for (; ; ) {
     if (comp instanceof View) {
          ((View)comp).processKeyEvent(evt);
```

3. Addition of post-condition check (IF-APTC)

The fix code adds an *if* statement after an operation to check the result from the operation. One example is error checking, as shown in the example below. This kind of bug occurs when a developer fails to consider different return results from an operation.

Example:

```
– parentFolder.addFolder(name);
+ FolderTreeNode folder = parentFolder.addFolder(name);
+ if (folder==null) success = false;
```

4. Removal of an if predicate (IF-RMV)

The fix removes an *if* predicate from the code. This bug occurs when there is an unnecessary condition check. There are many fewer instances of this bug fix pattern than of IF-APC.

Example:

```
– if (seg.array == null || seg.array.length < len)
      seg.array=new char[len];
```

5. Addition of an else branch (IF-ABR)

The bug fix adds an *else* branch to an *if* statement to cover a condition not previously considered.

Example:

```
  else  if (aname == "PLUGIN")     depPlugin=value;
+ else  if (aname == "SIZE")     size=Integer.parseInt(value);
```

6. Removal of an else branch (IF-RBR)

This bug fix pattern is the opposite of IF-ABR. This bug fix removes an *else* branch, thereby freeing the code in the *else* body from the constraint of the *if* condition.

Example:

```
– else     addOptionGroup(pluginsGroup,rootGroup);
+ addOptionGroup(pluginsGroup,rootGroup);
```

There are many fewer instances of this bug fix pattern than of IF-ABR.

7. Change of if condition expression (IF-CC)

This bug fix change fixes the bug by changing the condition expression of an *if* condition. The previous code has a bug in the *if* condition logic. This pattern is further explored in Section 4.3.3, which presents sub-patterns describing common changes within the conditional.

Example:

```
– if (getView().countSelected() == 0) {
+ if (getView().countSelected() <= 1) {
```

4.1.2 Method Call (MC)

8. Method call with different number of parameters or different types of parameters (MC-DNP)

The bug fix changes a method call in the code by using a method with the same method name but different number of parameters or types of parameters.

This change may be caused by the change of method interface, or use of an overloaded method.

Example:

```
– query = getLuceneQuery(filter.getFilterRule());
+ query = getLuceneQuery(filter.getFilterRule(), analyzer);
```

9. Method call with different actual parameter values (MC-DAP)

The bug fix changes the expression passed into one or more parameters of a method call.

Example:

```
– tree.putClientProperty("JTree.lineStyle","Horizontal");
+ tree.putClientProperty("JTree.lineStyle","Angled");
```

10. Change of method call to a class instance (MC-DM)

The fix code calls a different member method of a class instance. This new method may have some name similarity to the one used in the bug version. This bug fix may be caused by a method being renamed or by a developer using an incorrect member method.

Example:

```
– Enumeration enum=windows.keys();
+ Enumeration enum=windows.elements();
```

4.1.3 Sequence (SQ)

11. Addition of operations in an operation sequence of method calls to an object (SQ-AMO)

The bug fix adds one or more method calls into a sequence of method calls to the same object. This kind of bug occurs when the developer was missing one or more method calls in a sequence of method calls to the same object.

Example:

```
  importDeclaration.setSourceRange();
– importDeclaration.setOnDemand(importReference.onDemand);
+ importDeclaration.setName(name);
+ importDeclaration.setOnDemand(onDemand);
```

12. Removal of operations from an operation sequence of method calls to an object (SQ-RMO)

The opposite of SQ-AMO, this bug fix pattern removes one or more method calls in a sequence of method calls to the same object.

Example:

```
   pathField.setPreferredSize(prefSize);
–  pathField.addFocusListener(new FocusHandler());
```

13. Addition of operations in a field setting sequence (SQ-AFO)

This bug fix pattern is similar to SQ-AMO, except that the operation sequence involves setting object fields, not method calls.

Example:

```
cd.sourceEnd=sourceEnd;
cd.modifiers=modifiers & AccVisibilityMASK;
```

```
+ cd.isDefaultConstructor=true;
```

14. Removal of operations from a field setting sequence (SQ-RFO)

This bug fix pattern is the opposite of SQ-AFO. The code in the bug version contains an unnecessary field setting operation.

Example:

```
– ref.sourceEnd=intStack[intPtr--];
  ref.sourceStart=intStack[intPtr--];
```

15. Addition or removal of method calls in a short construct body (SQ-AROB)

This bug fix adds or removes method calls from a construct body, such as a method body, if body, while body, etc., that only contains two or three statements. Unlike the SQ-AMO and SQ-RMO patterns, SQ-AROB does not require the method calls involved to be to the same object variables. This kind of bug fix occurs when the developer was missing some operations or included unnecessary operations in an operation sequence.

Example:

```
if (evt.getClickCount() == 2) {
      jEdit.showMemoryDialog(view);
   + memory.repaint();
}
```

4.1.4 Loop (SQ)

16. Change of loop predicate (LP-CC)

The bug fix changes the loop condition of a loop statement. The previous code has a bug in the loop condition logic.

Example:

```
–while (!buffer._isLineVisible(line,index)) line--;
+while (!buffer._isLineVisible(line,index) && line > 0) line--;
```

17. Change of the expression that modifies the loop variable (LP-CE)

The bug fix changes the expression that modifies the loop variable or adds a statement that modifies the loop variable.

Example:

```
while (comp != null) {
    ……
  +comp=comp.getParent();
}
```

4.1.5 Assignment (AS)

18. Change of assignment expression (AS-CE)

The bug fix changes the expression on the right hand side of an assignment statement. The expression on the left-hand side is the same in both the bug and the fix versions.

Example:

```
– interfacesRange[1]=bodyStart - 1;
+interfacesRange[1]=superinterfaceEnds[
+                                     superinterfaces.length - 1];
```

4.1.6 Switch (SW)

19. Addition/removal of switch branch (SW-ARSB)

The bug fix adds or removes a case from a *switch* statement. The previous code was missing a case or includes an unnecessary case condition.

Example:

```
+ case ClassFileStruct.ClassTag:
+     name=extractClassReference(
+                 constantPoolOffsets,reader,i);
```

4.1.7 Try (TY)

20. Addition/removal of try statement (TY-ARTC)

The bug fix adds a try/catch statement to enclose a section of code, or removes a try/catch construct from the code in the bug hunk.

Example:

```
+ try {
    mimePartTree = srcFolder.getMimePartTree(uid, wsc);
+ }  catch ( FileNotFoundException ex) {
+     return;
+ }
```

21. Addition/removal of a catch block (TY-ARCB)

The bug fix adds a catch block to a try statement, or removes a catch block from a try statement in the bug hunk. The previous code failed to capture a kind of exception caused by the code in the try block.

Example:

```
+ } catch ( InvocationTargetException ex ) {
+     ex.getCause().printStackTrace();
+     throw ex;
```

4.1.8 Method Definition (MD)

22. Change of method declaration (MD-CHG)

The bug fix changes the declared interface for a method. The interface change may increase or decrease the number of parameters, change parameter types, change the return type, or change a method access modifier. This kind of change usually also leads to changes at call sites to this method.

Example:

```
– public int fetchMessageCount() throws Exception
+ public int fetchMessageCount(WorkerStatusController
+                                 worker) throws Exception
```

23. Addition of a method definition (MD-ADD)

A method definition is added in the fix version.

Example:

```
+ public void removeNotify(){
+    jEdit.setIntegerProperty("vfs.browser.splitter",
+            splitPane.getDividerLocation());
+    super.removeNotify();
+ }
```

24. Removal of a method definition (MD-RMV)

A method definition is deleted from the bug version.

Example:

```
– public static Image getEditorIcon(){
–     return ((ImageIcon)EDITOR_WINDOW_ICON).getImage();
–   }
```

4.1.9 Class Field (CF)

25. Addition of a class field (CF-ADD)

A class field is added in the fix version.

Example:

```
+ private NameReference[] unknownRefs;
+ private int unknownRefsCounter;
```

26. Removal of a class field (CF-RMV)

A class field is removed from the bug version.

Example:

```
– private MarkerHighlight markerHighlight;
```

27. Change of class field definition (CF-CHG)

A class field definition is changed in the bug fix revision.

Example:

– JPanel content=new JPanel(new BorderLayout());
+ JPanel content=new JPanel(new BorderLayout(12,12));

4.1.10 Ignored patterns

Besides the bug fix patterns identified above, there are still some bug fixes that have an obvious pattern, but are ignored because those bug fixes are trivial code changes and cause little semantic difference. Examples of the ignorable bug fixes include addition or removal of debug information, addition or removal of output statements, and changes to import statements. Some other patterns appear very rarely in the source code we examined, such as change of inherited super class, or changes related to the *synchronized* construct. We ignored these patterns too.

Bug fix patterns that require expensive program analysis to detect are also omitted, including statement permutation, variable renaming, and removal of dead code.

4.2 Tools for Extracting Bug Fix Pattern Instances

The bug fix pattern extractor tool consists of two major parts, a parser module and a pattern discovery module. The parser module is responsible for parsing the Java source code and collecting syntax information for each statement. The syntax information for a statement includes the AST tree of this statement, and the structure context of the statement. The parser module uses JDT [22].

The pattern discovery module is responsible for recognizing the bug fix pattern from each bug fix hunk pair. The syntax information for the code in the bug hunk and the fix hunk is used by the pattern discovery module to determine what bug fix patterns this bug fix hunk pair contains. Figure 4-2 shows the pseudo-code of the bug fix pattern extractor.

```
Bug_Fix_Pattern_Extractor(BugFile, FixFile)
Input: a source code file of bug version, BugFile, and the fix version of the source
        code file, FixFile. Both BugFile and FixFile have completed the denoising
        process.
Output: BugFixPatternInstanceList, a list of bug fix pattern instances, each of which
        is the identifier of a bug fix hunk pair and the name of a bug fix pattern
        identified for it.
Begin
    BugFixPatternInstanceList is empty.
    Parse BugFile to generate AST for each statement and create the symbol table.
    Parse FixFile to generate AST for each statement and create the symbol table.
    Search in the Delta table and obtain a list of bug fix hunk pairs for BugFile and
            FixFile.
    For each bug fix hunk pair, the pattern discovery module uses rules to check
        whether there is a bug fix pattern in the bug fix pair. If there is, add the
        discovered bug fix pattern instance in BugFixPatternInstanceList.
    Return BugFixPatternInstanceList.
End
```

Figure 4-2. Pseudo-code of bug fix pattern extractor.

In the pattern discovery module, each bug fix pattern has a rule to determine whether a bug fix hunk pair has an instance of this pattern. These rules take into account the statements in the bug fix hunk pair, the statements around the bug hunk and the fix hunk, and the construct that encloses the code in the bug hunk or the fix hunk. We describe the pattern discovery rule for each bug fix pattern below.

1. Addition of precondition check (IF-APC)
   (1) There is an *if* predicate that is in the fix hunk, but does not exist in the bug hunk.
   (2) The code (not in the fix hunk) enclosed by the *if* predicate in the fix version has corresponding code in the bug version.

If these conditions are met, the pattern discovery module reports that this bug fix hunk pair contains an IF-APC pattern instance.

2. Addition of precondition check with jump (IF-APCJ)
   (1) There is an *if* predicate that is in the fix hunk, but does not exist in the bug hunk.
   (2) There is a return, continue, or break statement in the fix hunk that is enclosed by the *if* predicate.

3. Addition of post-condition check (IF-APTC)

Case 1:

   (1) There is an *if* predicate that is in the fix hunk, but does not exist in the bug hunk.
   (2) There is an assignment statement one or several lines before the *if* statement in the fix version. The assignment statement may or may not be

in the fix hunk, and the assignment defines a variable used in the *if* predicate.

An example of case 1 is:

```
 t = foo.bar();
+if (t) {
```

Case 2:

(1) There is an *if* predicate that is in the fix hunk, but does not exist in the bug hunk.

(2) The *if* predicate contains a method call or assignment statement, and this method call or assignment statement exists in the bug hunk.

An example of case 2 is:

```
- foo.bar();
+ if (foo.bar())
```

4. Removal of an if predicate (IF-RMV)

(1) There is an *if* predicate that is in the bug hunk, but does not exist in the fix hunk.

(2) The code (not in the bug hunk) enclosed by the *if* predicate in the bug version has corresponding code in the fix version.

5. Addition of an else branch (IF-ABR)

(1) There is an *else* branch that is in the fix hunk, but does not exist in the bug hunk.

6.Removal of an else branch (IF-RBR)

(1) There is an *else* branch that is in the bug hunk, but does not exist in the fix hunk.

7. Change of if condition expression (IF-CC)

(1) There is an *if* predicate in both the bug hunk and the fix hunk, and the *if* condition in the bug hunk is different from that in the fix hunk.

Note that we only consider the first *if* predicate in both the bug hunk and the fix hunk when identifying this pattern. The possibility that there is more than one *if* predicate in a bug hunk or a fix hunk is low, since we only consider small hunks.

8. Method call with different number of parameters or different types of parameters (MC-DNP)

(1) There is the same method call in the bug hunk and in the fix hunk.

(2) The method call in the bug hunk has a different number of actual parameters, or types of actual parameters from all the method calls with the same method name in the fix hunk.

9. Method call with different actual parameter values (MC-DAP)

(1) There is the same method call with the same number and type of parameters.

(2) At least one actual parameter expression passed into the method call in the bug hunk is different from the corresponding one in the fix hunk.

10. Change of method call to a class instance (MC-DM)

    (1) There is a method call to a class or a class instance in the bug hunk.

    (2) There is a method call in the fix hunk that corresponds to the method call in the bug hunk. The method call in the fix hunk invokes the same class or class object as in the bug hunk, but the method name is different from that in the bug hunk.

11. Addition of operations in an operation sequence of method calls to an object (SQ-AMO)

    (1) There is a method call to an object variable in the fix hunk.

    (2) This method call is not made in the bug hunk (the bug hunk can be empty).

    (3) In code close to the method call in the fix version, there are other method calls to the same object variable that are found in both the bug version and the fix version. We have a threshold *h*, typically 20, to specify the meaning of "close to the method call". That is, the code in the range of *h* lines from the method call both before and after is the code *close to* the method call. In the rest of this subsection, "close to" has the same definition.

12. Removal of operations from an operation sequence of method calls to an object (SQ-RMO)

    (1) There is a method call to an object variable in the bug hunk.

    (2) This method call is not made in the fix hunk (the fix hunk can be empty).

(3) In code close to the method call in the bug version, there are other method calls to the same object variable that are found in both the bug version and the fix version.

13. Addition of operations in a field setting sequence (SQ-AFO)

(1) There is an assignment to a field of an object variable in the fix hunk.

(2) This field assignment is not made in the bug hunk (the bug hunk can be empty).

(3) In code close to the field setting statement in the fix version, there are other field assignments to the same object variable that are found in both the bug version and the fix version.

14. Removal of operations from a field setting sequence (SQ-RFO)

(1) There is an assignment to a field of an object variable in the bug hunk.

(2) This field assignment is not made in the fix hunk (the fix hunk can be empty).

(3) In code close to the field setting statement in the bug version, there are other field assignments to the same object variable that are found in both the bug version and the fix version.

15. Addition or removal of method calls in a short construct body (SQ-AROB)

Case of addition

(1) There is a method call to an object variable in the fix hunk.

(2) This method call is not made in the bug hunk (the bug hunk can be empty).

(3) The method call is in a construct body, such as method body, *if* body, *do* body, *while* body, and etc., that only contains less than or equal to three statements.

Case of removal

(1) There is a method call to an object variable in the bug hunk.

(2) This method call is not made in the fix hunk (the fix hunk can be empty).

(3) The method call is in a construct body, such as method body, *if* body, *do* body, *while* body, and etc., that only contains two or three statements.

16. Change of loop predicate (LP-CC)

(1) There is a loop predicate in both the bug hunk and the fix hunk, and the loop condition in the bug hunk is different from that in the fix hunk.

17. Change of the expression that modifies the loop variable (LP-CE)

(1) There is an assignment to a variable in the bug hunk and the fix hunk.

(2) This assignment is in a loop body.

(3) The variable defined in the assignment is used in the loop predicate.

18. Change of assignment expression (AS-CE)

(1) There is an assignment statement to a variable in the bug hunk and the fix hunk respectively.

(2) This left hand side of the assignment statement in the bug hunk is the same as the one in the fix hunk.

(3) This right hand side of the assignment statement in the bug hunk is different from the one in the fix hunk.

19. Addition/removal of switch branch (SW-ARSB)

Case of addition

(1) There is a *case* branch that is in the fix hunk which does not exist in the bug hunk.

Case of deletion

(1) There is a *case* branch that is in the bug hunk which does not exist in the fix hunk.

20. Addition/removal of try statement (TY-ARTC)

Case of addition

(1) There is a *try* statement that is in the fix hunk, but does not exist in the bug hunk.

(2) The code (not in the fix hunk) enclosed by the *try* construct in the fix version has corresponding code in the bug version.

Case of deletion

(1) There is a *try* statement that is in the bug hunk, but does not exist in the fix hunk.

(2) The code (not in the bug hunk) enclosed by the *try* construct in the bug version has corresponding code in the fix version.

21. Addition/removal of a catch block (TY-ARCB)

Case of addition

(1) There is a *catch* statement that is in the fix hunk, but does not exist in the bug hunk.

Case of deletion

(1) There is a *catch* statement that is in the bug hunk, but does not exist in the fix hunk.

22. Change of method declaration (MD-CHG)

(1) There is a method declaration with the same method name but different signature between the bug version and the fix version.

23. Addition of a method definition (MD-ADD)

(1) There is a method definition that is in the fix hunk, but does not exist in the bug hunk.

24. Removal of a method definition (MD-RMV)

(1) There is a method definition that is in the bug hunk, but does not exist in the fix hunk.

Note that the bug fix pattern analysis does not address the method renaming situation, i.e. it treats method renaming as removal of a method and addition of a new method. Identifying method renaming requires deeper analysis of the source code, such as origin analysis [28, 49].

25. Addition of a class field (CF-ADD)

(1) There is a class field definition that is in the fix hunk, but does not exist in the bug hunk.

26. Removal of a class field (CF-RMV)

(1) There is a class field definition that is in the bug hunk, but does not exist in the fix hunk.

27. Change of class field declaration (CF-CHG)

(1) There is a class field definition in both the bug hunk and the fix hunk, but either the field name or the initialization expression is different between the bug hunk and the fix hunk.

4.3 Characteristics of Bug Fix Patterns

Having described the bug fix patterns and the tool used to automatically detect them, we shift to a characterization of bug fix patterns in several Java projects.

4.3.1 Distribution and Frequency of Pattern Types

The pattern extractor tool was used to analyze the change history of seven open source projects, ArgoUML, Columba, Eclipse, JEdit, Scarab, Lucene, and MegaMek, summarized in Table 3-5. The analysis ignores large bug fix hunk pairs. The rationale for ignoring large hunk pairs is that large bug fix hunk pairs generally look random and do not contain meaningful bug fix patterns. In particular, addition of a new file usually results in large bug fix hunk pairs, which are ignored by the extractor. We defined large hunks as those containing more than seven lines of code, based on a rule of thumb. That is, during the process of manually examining the bug fix hunk pairs in the analyzed projects, we observed that the bug fix hunk pairs generally appeared to be random when a hunk contained more than seven lines of code. This rule of thumb may not work for other projects, and we will test other thresholds in future. In the projects we studied, most bug fix hunk pairs (91% to 96%) are small ones, and hence ignoring large hunk pairs has low impact on the analysis.

We examined the pattern coverage for the bug fix changes in the change history of the projects. The results in Figure 4-3 show that 45.7% to 63.6% of the small bug fix hunk pairs contain a bug fix pattern. In the remaining discussion, we examine characteristics of bug fix patterns, and it is worth remembering that these patterns encompass only about half of the bug fixes observed in these projects.

With patterns extracted, we count how many instances of each pattern type were observed over the lifetime of each project. The frequency of each pattern among all observed patterns is also computed. These results are listed in Table 4-2.

Somewhat surprisingly, the results show two clear spikes in frequency. The Method Call (MC) and If-Related (IF) categories are the most prevalent bug fix patterns. Together they account for 44.6% to 60.3% of all bug fix pattern instances. In the Method Call category, most of the bug fixes to method calls are changes to the actual parameter expressions (MC-DAP). Getting parameter lists correct is the single largest source of programmer error observed.

Category	Short Name	ArgoUML (#, %)		Columba (#, %)		Eclipse (#, %)		JEdit (#, %)		Scarab (#, %)		Lucene (#, %)		MegaMek (#, %)	
Assignment (AS)	AS-CE	685	8.5	159	6.0	1208	6.3	404	8.3	169	7.2	255	8.3	1120	14.2
		685	8.5	159	6.0	1208	6.3	404	8.3	169	7.2	255	8.3	1120	14.2
Class Field (CF)	CF-ADD	246	3	105	4.0	707	3.7	216	4.4	66	2.8	102	3.3	269	3.4
	CF-CHG	230	2.9	85	3.2	422	2.2	94	1.9	44	1.9	144	4.7	227	2.9
	CF-RMV	132	1.6	42	1.6	208	1.1	108	2.2	42	1.8	106	3.4	61	0.8
		608	7.5	232	8.7	1337	7.0	418	8.6	152	6.5	352	11.4	557	7.1
If-related (IF)	IF-ABR	78	1	29	1.1	208	1.1	88	1.8	22	0.9	17	0.6	103	1.3
	IF-APC	318	3.9	148	5.6	1142	6.0	266	5.5	124	5.3	68	2.2	199	2.5
	IF-APCJ	295	3.7	90	3.4	726	3.8	164	3.4	34	1.5	60	1.9	196	2.5
	IF-APTC	131	1.6	102	3.8	285	1.5	131	2.7	37	1.6	16	0.5	89	1.1
	IF-CC	868	10.8	149	5.6	3553	18.6	624	12.9	250	10.7	370	12.0	1157	14.7
	IF-RBR	25	0.3	15	0.6	96	0.5	51	1.1	8	0.3	10	0.3	18	0.2
	IF-RMV	139	1.7	56	2.1	441	2.3	140	2.9	51	2.2	67	2.2	78	1.0
		1854	23	589	22.2	6451	33.9	1464	30.2	526	22.5	608	19.7	1840	23.4
Loop (LP)	LP-CC	114	1.4	23	0.9	297	1.6	59	1.2	31	1.3	84	2.7	230	2.9
	LP-CE	23	0.3	3	0.1	34	0.2	11	0.2	2	0.1	7	0.2	5	0.1
		137	1.7	26	1.0	331	1.7	70	1.4	33	1.4	91	3.0	235	3.0
Method Call (MC)	MC-DAP	1921	23.8	432	16.3	3416	17.9	725	14.9	596	25.5	515	16.7	1500	19.1
	MC-DM	75	0.9	59	2.2	256	1.3	105	2.2	27	1.2	105	3.4	224	2.8
	MC-DNP	428	5.3	104	3.9	1365	7.2	233	4.8	151	6.5	166	5.4	181	2.3
		2424	30	595	22.4	5037	26.4	1063	21.9	774	33.1	786	25.5	1905	24.2
Method Declaration (MD)	MD-CHG	511	6.3	204	7.7	1108	5.8	289	6.0	149	6.4	176	5.7	275	3.5
	MD-ADD	540	6.7	254	9.6	1036	5.4	256	5.3	172	7.4	358	11.6	309	3.9
	MD-RMV	243	3	90	3.4	358	1.9	98	2.0	58	2.5	165	5.4	56	0.7
		1294	16	548	20.6	2502	13.1	643	13.2	379	16.2	699	22.7	640	8.1
Sequence (SQ)	SQ-AFO	18	0.2	3	0.1	132	0.7	31	0.6	1	0	5	0.2	707	9.0
	SQ-AMO	418	5.2	203	7.6	562	2.9	267	5.5	103	4.4	62	2.0	458	5.8
	SQ-AROB	255	3.2	99	3.7	376	2.0	203	4.2	77	3.3	24	0.8	67	0.9
	SQ-RFO	26	0.3	1	0.0	59	0.3	13	0.3	0	0	8	0.3	143	1.8
	SQ-RMO	265	3.3	105	4.0	263	1.4	204	4.2	90	3.9	66	2.1	128	1.6
		982	12.2	411	15.5	1392	7.3	718	14.8	271	11.6	165	5.4	1503	19.1
Switch (SW)	SW-ARSB	3	0	9	0.3	305	1.6	28	0.6	0	0	72	2.3	55	0.7
		3	0	9	0.3	305	1.6	28	0.6	0	0	72	2.3	55	0.7
Try (TY)	TY-ARCB	41	0.5	51	1.9	450	2.4	23	0.5	23	1	44	1.4	15	0.2
	TY-ARTC	42	0.5	36	1.4	44	0.2	24	0.5	8	0.3	11	0.4	3	0.0
		83	1	87	3.3	494	2.6	47	1.0	31	1.3	55	1.8	18	0.2

Table 4-2. Count and frequency of extracted pattern instances for analyzed projects.

Within the If-Related category, the If-Conditional Change (IF-CC) pattern has many more instances than other patterns in the IF category. Viewing this data in a different way, it is possible to lump together the loop condition pattern (LP-CC) with the Switch (SW) and If-Related (IF) categories to develop an aggregate sense of bug fixes that repair logic errors. This composite category accounts for 23.5% to

37.2% of all bug fix pattern instances. At least for the observed projects, program logic is a notable source of error.

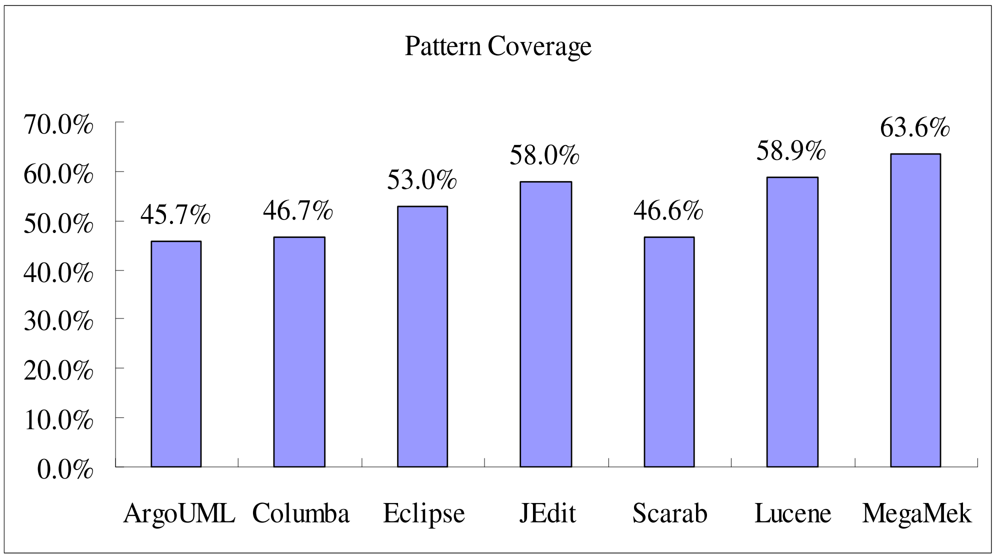


Figure 4-3. Ratio of the bug fix hunk pairs that has at least one pattern to the total number of bug fix hunk pairs in the analyzed projects.

Bug fixes in the Switch (SW) category are quite rare, but we note that the Eclipse project has a much higher ratio of Switch bug fixes than the other projects. The reason for this phenomenon is that that there are many *case* expressions in the Eclipse project. For example, the file 'jdt/internal/compiler/parser/Parser.java' in Eclipse contains hundreds of *case* expressions.

4.3.2 Cross-project Similarity

Table 4-3 presents the Pearson's correlations between the ratios of pattern instances of bug fix patterns in different projects. The correlation values in the results are high (most of them are greater than 0.85) among all the projects except for MegaMek, and all correlations are significant (p-value < 0.001). The quantitative inspection of the results shows that most of these projects have a very similar bug fix pattern distribution.

We manually examined the bug fix changes in MegaMek to find out why it has such different bug fix pattern distributions from other projects. MegaMek is an online BattleTech board game. In its design, MegaMek defines Java classes for many kinds of weapons, ammunition, and music. Classes in the same group, e.g. weapons, are very similar to each other in design and have the same list of class fields. So, when there is a bug in one class, such as a bug in the field setting statement or addition of a field setting statement, the same bug will occur in many other classes in the same group. That is the reason MegaMek has a much higher ratio of AS-CE and SQ-AFO bug fix pattern instances than other projects.

All in all, most of the projects we observed have similar bug fix pattern distributions, and the special nature and design of the MegaMek project make it have different pattern distributions from other projects observed.

	ArgoUML	Columba	Eclipse	JEdit	Scarab	Lucene	MegaMek
ArgoUML	1	0.91	0.89	0.93	0.99	0.89	0.70
Columba		1	0.75	0.85	0.91	0.82	0.56
Eclipse			1	0.94	0.89	0.86	0.65
JEdit				1	0.92	0.85	0.66
Scarab					1	0.88	0.66
Lucene						1	0.71
MegaMek							1

Table 4-3. Pearson's correlation between the ratios of pattern instances of patterns in different projects.

4.3.3 If Conditionals

The language keyword that is the single greatest individual source of bug fixes is *if*, with the If-conditional (IF-CC) pattern accounting for 5.6%-18.6% of all bug fix patterns. To better characterize this type of change, several sub-patterns were developed.

Changes to *if* conditions are a mixture of regular and random changes. Regular changes include addition of a condition clause, removal of a condition clause, addition of a variable in the condition expression, etc. Random changes involve complete turnover of the conditional (all variables and operators changed), and change to a function call in the *if* condition (changed method parameters, changed method name, etc.).

We consider three factors, condition clauses, as well as variables and operators in an *if* condition, and list the finer-grained bug fix patterns found under the IF-CC pattern below. Note that the five sub-patterns may have overlaps.

1. IF-SUB-AC: a new condition clause is added. For example, *if (flag > 5)* is changed to *if (flag > 5 && flag < 10)*.
2. IF-SUB-RC: a condition clause is removed. For example, *if (flag > 5 && flag < 10)* is changed to *if (flag > 5)*.
3. IF-SUB-AV: a new variable is used in the condition. For example, *if (flag > 5)* is changed to *if (flag > 5 && length > 0)*.
4. IF-SUB-RV: there is a decrease of the number of variables used in the condition. For example, *if (flag > 5 && flag < 10)* is changed to *if (flag > 5)*.
5. IF-SUB-AO: there is an increase of the number of operators in the condition. The operator factor indicates the complexity of a condition. For example, *if (len > start)* is changed to *if (len > start + 1)*.
6. IF-SUB-RO: there is a decrease of the number of operators in the condition. For, example, *if (len > start + 1)* is changed to *if (len > start)*.

An extractor tool was developed to extract instances of the finer-grained IF-CC patterns, and run on the seven projects. The distribution of these patterns is shown in Table 4-4. The results show that 25% of IF-CC bug fixes on average involve addition or removal of condition clauses from an *if* condition (IF-SUB-AC and IF-SUB-RC). Generally, there are more bug fixes that add new clauses than remove clauses from an *if* condition, which means that it is more typical for developers to miss conditions in the *if* logic.

In five of the projects (columba, Eclipse, JEdit, Lucene, and MegaMek), the number of variables in an *if* condition tends to increase (more IF-SUB-AV instances than IF-SUB-RV ones) when fixing a bug, whereas two projects (ArgoUML and Scarab) have the contrary trend. In all seven projects, the complexity based on the number of operators of *if* conditions tends to increase in bug fixes, since there are many more IF-SUB-AO instances than IF-SUB-RO ones.

	ArgoUML	**Columba**	**Eclipse**	**JEdit**	**Scarab**	**Lucene**	**MegaMek**
IF-SUB-AC	13.1%	28.9%	20.8%	23.1%	20.8%	4.3%	16.1%
IF-SUB-RC	11.5%	8.7%	6.9%	11.2%	3.6%	4.3%	2.7%
IF-SUB-AV	8.3%	14.1%	14.3%	23.7%	16.4%	11.4%	19.5%
IF-SUB-RV	12.0%	5.4%	9.7%	15.1%	16.8%	10.5%	11.6%
IF-SUB-AO	22.4%	37.6%	22.3%	38.0%	26.8%	13.2%	27.8%
IF-SUB-RO	14.6%	11.4%	15.1%	21.0%	10.4%	11.6%	11.8%

Table 4-4. Bug fix distribution on finer-grained IF-CC pattern.

4.3.4 Bug Fix Pattern Distribution by Developer

It is interesting to know the pattern distribution of bugs for each developer. That is, we can know what kinds of bugs a developer tends to introduce the most. To obtain this data, we apply a bug introduction analysis [50] which is a refinement to the SZZ algorithm [79], on buggy code that is covered by an instance of bug fix pattern. Bug introduction analysis traces backwards from a bug fix revision through a file's revision history to find the origin of each line containing a bug, so we can know who introduced the problematic line, and in which revision. For example, if we find that the line *if (foo.flag > 1)* is covered by an IF-CC pattern instance in a bug hunk in revision 200, and we use the bug introduction analysis to find that this line is introduced by the developer *Bob* in revision 50, we say that the developer *Bob* introduced a buggy line in revision 50 that is solved by an IF-CC bug fix in revision 200.

We analyzed the bug fix pattern distribution through bug introduction for four major development members in the Eclipse project. Table 4-5 presents the Pearson's correlations between the pattern distributions for different developers, and Figure 4-4 shows the detailed pattern distribution by developer. In Figure 4-4, the X axis represents the bug fix patterns, and the Y axis represents the percentage, i.e. each bar for a developer and a pattern indicates the ratio of bug fixes of this pattern this developer's code has caused to all the bug fixes this developer's code has caused. The 'Expected Ratio' bar for a bug fix pattern indicates the average ratio of the number of instances of the pattern to the number of instances of all kinds of bug fix pattern.

Though Table 4-5 shows that the bug fix patterns have similar pattern distributions between different developers (most of the Pearson's correlations are greater than 0.85), we can still see that some developers have an obvious tendency in introducing more or less bugs of some individual patterns than other developers. The results in Figure 4-4 show that the *ffusier*'s code caused less than average problems that are caused by missing *if* condition checks (IF-APC), but she/he tends to introduce more problems in assignment statements (AS-CE). The developer *jlanneluc*'s code caused higher ratio of fixes of adding precondition checks with jump (IF-APCJ) than average, and her/his code caused relatively more problems in method call definition (MD-CHG). *Othomann* tended to make more mistakes in method call sequences (SQ-AMO, SQ-RMO), since her/his code introduced a high ratio of bugs in method call sequences that were fixed by SQ-AMO and SQ-RMO patterns. The project member *pmulet* tended to introduce a high ratio of bugs that misused of catch branches (TY-ARCB).

One thing needs to be noted in this analysis. This analysis reflects what kinds of bugs a developer tends to produce, but besides the developer themselves, other factors such as the nature of the project parts assigned to that developer may also affect the specific error production tendency of developers.

	ptff	jlanneluc	ffusier	pmulet	othomann
ptff	1	0.92	0.94	0.96	0.88
jlanneluc		1	0.92	0.88	0.84
ffusier			1	0.91	0.82
pmulet				1	0.87
othomann					1

Table 4-5. Pearson's correlation between the pattern distributions for different developers.

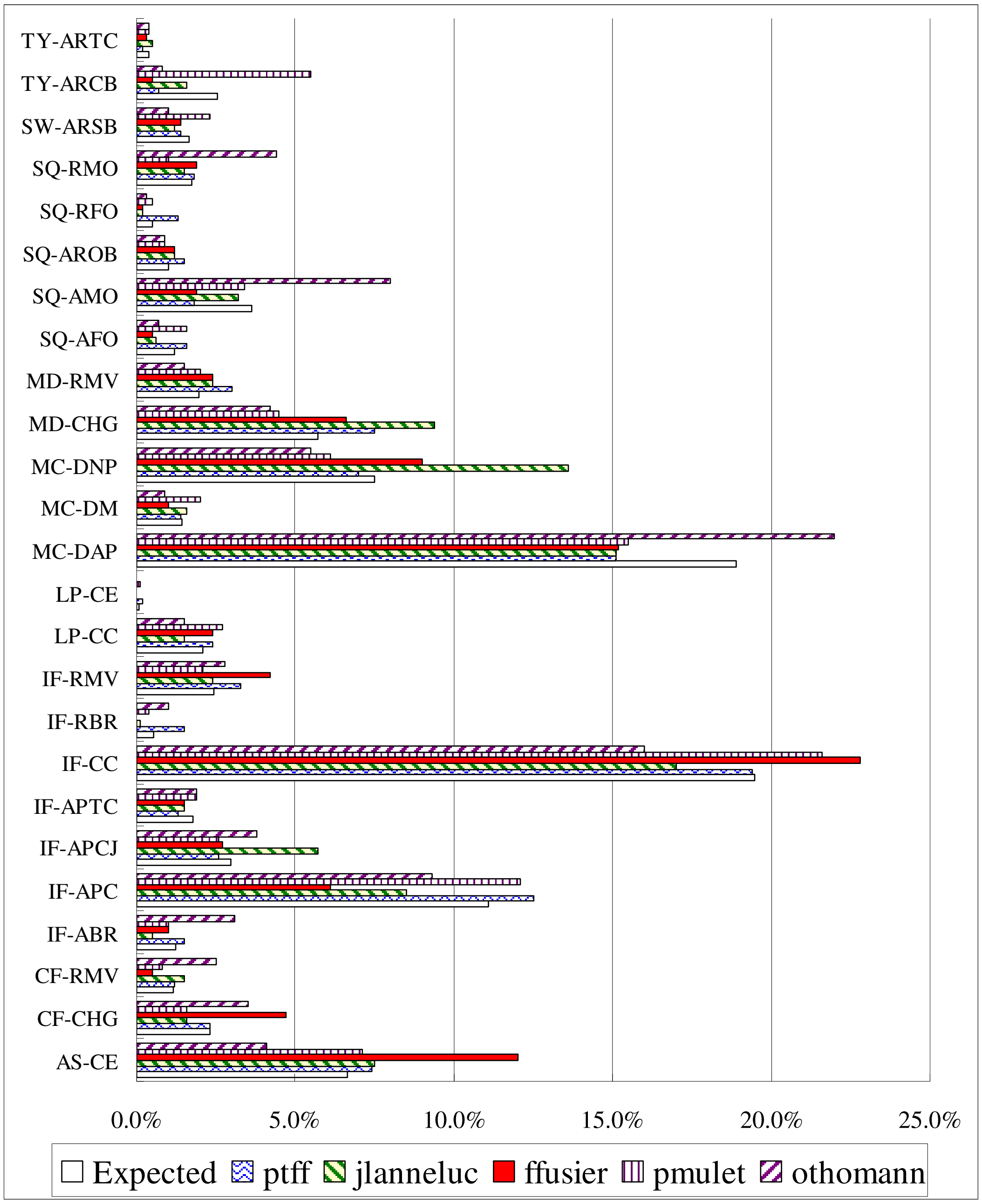


Figure 4-4. Pattern distribution of bug introduction for four major developers in the Eclipse project.

4.3.5 Bug Fix Pattern Distribution by Module

It is also interesting to know what kinds of bugs each project module tends to contain. This kind of information may be helpful for the developers working on this module, since it raises awareness about the relative riskiness to commit the more prevalent kinds of bugs. This information may be helpful for other bug prediction approaches as well.

We analyzed the bug fix pattern distribution for five major modules in the Eclipse project. Here, a module represents a dictionary of files. Table 4-6 presents the Pearson's correlations between the pattern distributions in different modules, and Figure 4-5 shows the data. In Figure 4-5, the X axis represents the bug fix patterns, and the Y axis represents the percentage, i.e. each bar for a module and a pattern indicates the ratio of bug fixes of this pattern made on this module to all the bug fixes made on this module. The 'Expected Ratio' bar for a bug fix pattern indicates the average ratio of the number of instances of the pattern to the number of instances of all kinds of bug fix patterns.

	compiler-internal	search-internal	dom-core	model-internal	search-core	dom-internal
compiler-internal	1	0.89	0.86	0.80	0.60	0.87
search-internal		1	0.90	0.93	0.68	0.95
dom-core			1	0.95	0.56	0.92
model-internal				1	0.60	0.95
search-core					1	0.62
dom-internal						1

Table 4-6. Pearson's correlation between the pattern distributions in different modules.

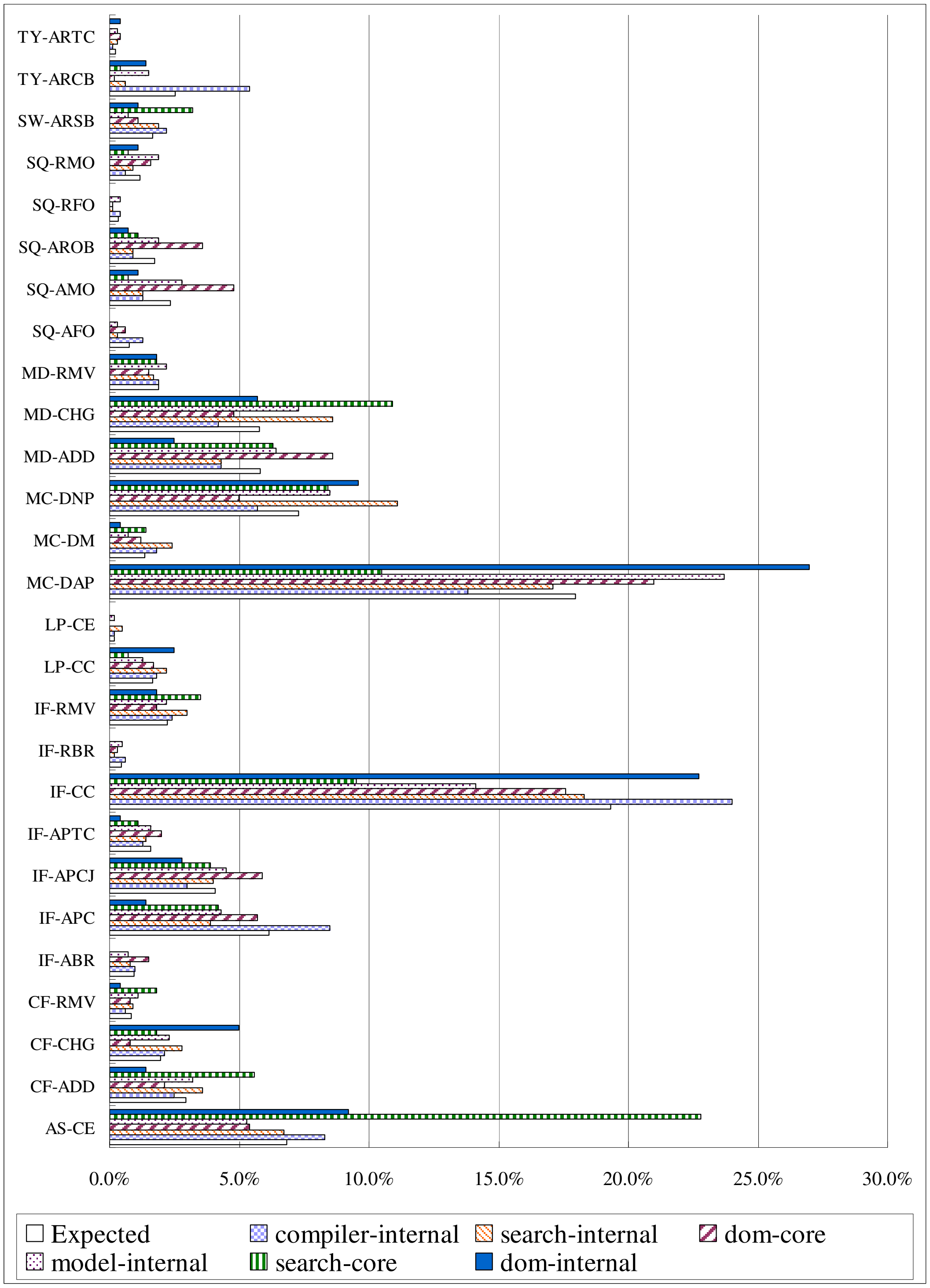


Figure 4-5. Pattern distribution by module in the Eclipse project.

As shown in Table 4-6, all the modules except for the *search-core* module have similar bug fix pattern distributions between each other (most of the Pearson's correlations are greater than 0.85). But when examining individual patterns and modules, we can still see some exceptional bug fix pattern distributions for some individual patterns and modules in the Eclipse project. The results in Figure 4-5 show that the *search-core* module has a smaller ratio of IF-CC bug fixes than average, but it has a tendency to have much more bugs in the assignment statements (AS-CE). The *dom-core* module has a high ratio of problems in method call sequence (SQ-AMO and SQ-AROB), and the *compiler-internal* module has a lot of problems in *catch* statements (TY-ARCB). Most of the bug fixes on *if* conditionals in the *dom-internal* module fall into the IF-CC pattern, while it has less problems with precondition checks or post-condition checks (IF-APC, IF-APCJ, IF-APTC).

4.4 Discovering Buggy Code using Bug Fix Patterns

In bug-related software evolution analysis, it is critical to locate buggy code areas in source code versions. For example, bug introduction analysis [50, 79] explores the origin of bugs, finding the author that injected a bug and the version in which it was injected, by tracing the source code changes along the revision history. If the buggy code area cannot be accurately identified, the results of bug introduction analysis will be in doubt.

Traditionally, there are two steps involved in automatically locating buggy code using software change history. First, change log messages are used to classify source code revisions as either bug fix revisions or non-fix revisions, as described in Section 3.4. Second, diff computes deltas between the bug version and the fix version of a file in a bug fix revision. The resulting bug hunks of the deltas indicate the areas of buggy code, since code in the bug hunks has been changed to fix a bug. Figure 4-6 shows an example of locating buggy code using bug fix hunk pair. The gray box indicates the bug hunk, and the white box indicates the fix hunk. In this example, the code in the gray box, lines 63 and 64 of the file in the bug version, is identified as buggy code, and the code in the white box, lines 64 and 65 in the fix version, is identified as fix code.

```
63: – size+=installer.getIntProperty("comp." + i + ".real-size");
64: – components.addElement(installer.getProperty("comp." + i + ".fileset"));
```

```
64: + size+=installer.getIntProperty("comp." + ids.elementAt(i) + ".real-size");
65: + components.addElement(installer.getProperty("comp." + ids.elementAt(i) + ".fileset"));
```

Figure 4-6. Example of locating buggy code using bug fix hunk pair.

Bug fix hunk pairs can identify most of the buggy code in the change history, but some bugs are missed, especially those bug fix hunk pairs whose bug hunk is empty. Figure 4-7 shows an example of missed buggy code. In this example, the fix version added an *if* condition before the statement, *setMessage(null)* to fix a bug. Since *setMessage(null)* exists in both the bug version and the fix version, *setMessage(null)* is not in the bug hunk, i.e. the bug hunk is empty, while the fix hunk contains the line *if (isShowing())*. In this case, *setMessage(null)* is not identified as buggy code in the traditional approach, since it is not in a bug hunk, but actually this statement has a bug since it is missing the precondition check *if (isShowing())*. There are still other cases where buggy code is not captured by bug fix hunk pairs. Since bug fix patterns consider the context of statements in the bug hunk and the fix hunk, we can use some of the bug fix patterns to discover the buggy code missed by bug fix hunk pairs.

Bug Version	Fix Version
 setMessage(null);	+ if (isShowing()) setMessage(null);

Figure 4-7. Example of missed buggy code.

4.4.1 Buggy Code Discovery using Bug Fix Patterns

In this section, we discuss the bug fix patterns that can discover additional buggy code and its corresponding fix code. These bug fix patterns include IF-APC, IF-APCJ, IF-APTC, IF-RMV, IF-ABR, IF-RBR, SQ-AMO, SQ-RMO, SQ-AFO, SQ-RFO, SW-ARSB, TY-ARTC, and TY-ARCB, most of which have an empty bug hunk in their pattern instances.

4.4.1.1 Context for Discovered Buggy Code or Fix Code

Before we discuss each bug fix pattern that can be used to discover additional buggy code and fix code, we first explain the concept of context for a statement. The context for a statement indicates the environment the statement is in, i.e. the nearby statements that are related to this statement, and the syntax construct, such as if, while, try, and etc., the statement is in. A statement can have multiple contexts. The buggy code discovered by patterns is the same as its corresponding fix code, but they have different context in the program. The context strings will also be used in the BugMem approach for pattern matching, as described in Section 5.1.3.

In this subsection, we discuss the context expressions that are applicable to the buggy code and fix code discovered by static bug fix patterns. The context expressions are used to differentiate the statements in buggy code from those in fix code, since they have the same program text between the bug version and the fix version. The context expressions are defined based on the summary of the context difference between the bug code and the fix code in the bug fix pattern. Though the projects we examined are all written in Java, these contexts are also applicable to C++ too, since C++ and Java share many common language constructs.

In the context expression, there are several special notations used to express a context for a statement. These notations are explained below.

1. "< >". An angle bracket pair enclosing a construct or an operation sequence kind. The construct or operation sequence kind indicates that the statement is enclosed by the construct or an operation sequence, or the statement is following the construct and has a control dependency relationship with it in the program. For example, the context <*if*> for a statement represents that the statement is enclosed by an *if* precondition

check, and the context <*jump*> for a statement indicates that the statement follows a jump statement in the program and is enclosed by a branch created by the jump statement.

2. "()". A construct enclosed by a parenthesis pair indicates that the construct follows the statement in the program text and has a dependency relationship with the statement. For example, the context *(if)* for a statement represents that there is an *if* statement following the statement and the *if* statement performs a post-condition check for the statement.
3. "^" indicates a logical negation. For example, ^<if> for a statement indicates that there is not an *if* precondition check for the statement.
4. "{ }" encloses an identifier list, which is the variable part of a context.

In the rest of this section, we describe each context expression and its meaning.

1. <if>

Context Expression: <if>

Context String Example: <if>

The <*if*> context for a statement represents that this statement is enclosed in an *if* predicate. For example, if the code is like:

```
if (foo != null)
    foo.bar();
```

We say the context for *foo.bar()* is <*if*>.

2. ^<if>

Context Expression: ^<if>

Context String Example: ^<if>

The ^<*if*> context for a statement represents that this statement is not enclosed in an *if* predicate. For example, the context for the code below is ^*(if)*.

```
foo.bar();
```

3. (if)

Context Expression: (if)

Context String Example: (if)

The *(if)* context for a statement indicates that there is an *if* statement that follows this statement and the *if* statement uses the variable defined by this statement. For example, in the following code:

```
ret = foo.bar();
if (ret != null)
  ……
```

We say *ret = foo.bar()* has context *(if)*.

4. ^(if)

Context Expression: ^(if)

Context String Example: ^(if)

The ^*(if)* context for a statement indicates that there is not an *if* statement that follows this statement within a few lines and the *if* statement uses the variable defined by this statement.

5. (else)

Context Expression: (else)

Context String Example: (else)

The *(else)* context for an *if* predicate indicates that there is an *else* branch corresponding to this *if* predicate. For example, in the following code:

```
if (ret != null)
  ……
else
  ……
```

We say *if (ret != null)* has context *(else)*.

6. ^(else)

Context Expression: ^(else)

Context String Example: ^(else)

The ^*(else)* context for an *if* predicate indicates that there is not an *else* branch corresponding to this *if* predicate.

7. <calls>{list_of_method_calls}

Context Expression: <calls>{*list_of_method_calls*}

Context String Example: <calls>{init,setReady,startWork,endWork}

<calls>{*list_of_method_calls*} context for a method call statement to an object represents a sequence of method calls to the same object, including the method call statement. For example, in the code sequence:

```
foo.init();
foo.setReady();
foo.startWork();
foo.endWork();
```

We say the context for *foo.startWork()* is <calls>{init, setReady, startWork, endWork}.

Note that we do not include the object variable, and only include the method names in a *list_of_method_calls* for two reasons. First, we want to approximate the information of the method call list to increase the possibility of pattern matching. Second, since we only perform a one-pass parse for the source code files in our implementation, the types of some variables are unknown. For the same reason, the object variables are not included in the <fields> context and the <ShortBody> context.

8. <fields>{list_of_fields}

Context Expression: <fields>{*list_of_fields*}

Context String Example: <fields>{flag1,flag2,flag3}

<fields>*{list_of_method_calls}* context for a field setting statement to an object represents the sequence of field setting statements to the same object and the field setting statement is one of them. For example, in the following code:

```
foo.flag1 = 5;
foo.flag2 = 10;
foo.flag3 = 15;
```

We say *foo.flag2 = 10;* has context <fields>{flag1,flag2,flag3}.

9. <ShortBody>{list_of_method_calls}

Context Expression: <ShortBody>{*list_of_method_calls*}

Context String Example: <ShortBody>{init, setReady, startWork}

The <ShortBody>{*list_of_method_calls*} context for a method call statement in a short construct body represents a sequence of method calls in the same short construct body. For example, in the following code:

```
void startFooWork() {
  foo.init();
  setReady();
  foo.startWork();
}
```

We say the context for *foo.startWork()* is <ShortBody>{init, setReady, startWork}.

10. <jump>

Context Expression: <jump>

Context String Example: <jump>

The *<jump>* context for a statement represents that the statement before this statement is a jump statement, such as continue, return, or break, enclosed in an *if* predicate. For example, in the following code:

```
if (foo == null)
      break;
foo.bar();
```

We say *foo.bar()* has context *<jump>*.

11. ^<jump>

Context Expression: ^<jump>

Context String Example: ^<jump>

The ^*<jump>* context for a statement represents that the statement before this statement is not a jump statement enclosed in an *if* predicate.

12. (case){case_expression_list}

Context Expression: (case){*case_expression_list*}

Context String Example: (case){1,2,3}

The *(case)* context for a *switch* predicate represents that the *switch* statement has the *case* branches specified in *case_expression_list*. For example, in the following code:

```
switch (foo) {
      case 1: ……
      case 2: ……
      case 3: ……
}
```

We say *switch(foo)* has context *(case)*{*1,2,3*}.

13. <try>

Context Expression: <try>

Context String Example: <try>

The <*try*> context for a statement represents that this statement is enclosed in a *try* construct. For example, in the following code:

```
 try {
    foo.bar();
} catch (FileNotFoundException ex) {
    return;
}
```

We say *foo.bar()* has context <*try*>.

14. ^<try>

Context Expression: ^<try>

Context String Example: ^<try>

The ^<*try*> context for a statement indicates that this statement is not enclosed in a *try* construct.

15. (catch){catch_expression_list}

Context Expression: (catch){*catch_expression_list*}

Context String Example: (catch){FileOperationException,MySearchException}

(catch){*catch_expression_list*} context for a method call statement in a *try* construct indicates that the *try* statement enclosing this statement has a corresponding *catch* expression in *catch_expression_list*. For example, in the following code:

```
try {
   foo.bar();
} catch (FileNotFoundException e) {
   return;
} catch (MySearchException e) {
   reportSQLError();
}
```

The context for statement *foo.bar()* is *(catch){FileOperationException, MySearchException}*.

4.4.1.2 Bug Fix Patterns that Discover Additional Buggy code and Fix Code

1. Addition of precondition check (IF-APC)

In this bug fix pattern, an *if* predicate is added to enclose one or multiple statements to perform a precondition check before these statements execute. In textual difference analysis, the statements enclosed in the *if* exist in both the bug version and the fix version, so they are not in a bug hunk and hence are not considered as buggy code. But, in fact, these statements do contain a bug, since they miss a precondition check. Figure 4-8 illustrates this point. The code in the gray box in the bug version and in the fix version indicates the additional buggy code and the corresponding fix code discovered by IF-APC pattern, but missed by textual difference (bug fix hunk) analysis. Note that, although the discovered buggy code is the same as the corresponding fix code, they have different contexts, i.e. *foo.bar()* in the bug version does not have an *if* predicate enclosing it, while *foo.bar()* in the fix version does.

The context string for the discovered buggy code in the example is ^<*if*>, and the context string for the discovered fix code is <*if*>.

Bug Version	Fix Version
	+ if (foo != null)
foo.bar();	foo.bar();

Figure 4-8. Additional buggy code and fix code discovered using IF-APC pattern.

2. Addition of precondition check with jump (IF-APCJ)

IF-APCJ is a variant of IF-APC. In IF-APCJ, the fix version adds a precondition check with a jump statement before some statements. Figure 4-9 shows an example. In this example *foo.bar()* is discovered as buggy code. Though *foo.bar()* is not enclosed in the newly added *if* predicate in the fix version, the added *return* statement generates an alternative control flow for *foo.bar()* based on the resulting value of the *if* condition. *foo.bar()* is recognized as buggy code due to the lack of precondition check before it in the bug version.

The context string for the discovered buggy code in the example is ^<*jump*>, and the context string for the discovered fix code is <*jump*>.

Bug Version	Fix Version
	+ if (foo == null)
	+ return;
foo.bar();	foo.bar();

Figure 4-9. Additional buggy code and fix code discovered using IF-APCJ pattern.

3. Addition of post-condition check (IF-APTC)

The IF-APCJ pattern can discover buggy code that is missing a required post-condition check. The example in Figure 4-10 shows that the statement *int res = foo.bar()* is discovered as buggy code, though it exists in both the bug version and the fix version. In the bug version, the return value from *foo.bar()* is used directly in the following code without any check, and hence could cause abnormal program behavior. The bug is fixed in the fix version by adding a post-condition check.

The context string for the discovered buggy code in the example is ^*(if)*, and the context string for the discovered fix code is *(if)*.

Bug Version	Fix Version
int res = foo.bar();	int res = foo.bar();
	+ if (res < 0) return;
car(res)	car(res)

Figure 4-10. Additional buggy code and fix code discovered using IF-APTC pattern.

4. Removal of an if predicate (IF-RMV)

This is the opposite case to IF-APC. As illustrated in Figure 4-11, *foo.start()* is recognized as buggy code, since it has an unnecessary precondition check.

The context string for the discovered buggy code in the example is <*if*>, and the context string for the discovered fix code is ^<*if*>.

Bug Version	Fix Version
– if (bar.ready())	
foo.startIt();	foo.startIt();

Figure 4-11. Additional buggy code and fix code discovered using IF-RMV pattern.

5. Addition of an else branch (IF-ABR)

The IF-ABR pattern discovers an *if* predicate that is missing an *else* branch. As the *if* predicate exists in both the bug version and the fix version, it is not marked as buggy code using textual difference analysis. But, the IF-ABR pattern recognizes it as buggy code, since the other branch is not covered by this *if* logic. Figure 4-12 illustrates an example, in which *if (foo.ready())* is discovered to be additional buggy code.

The context string for the discovered buggy code in the example is ^*(else)*, and the context string for the discovered fix code is *(else)*.

Bug Version	Fix Version
if (foo.ready())	if (foo.ready())

foo.startIt();	foo.startIt();
	+else
	+ foo.start();

Figure 4-12. Additional buggy code and fix code discovered using IF-ABR pattern.

6. Removal of an else branch (IF-RBR)

This is the case opposite to IF-ABR. As shown in Figure 4-13, *if (foo.ready())* is recognized as buggy code, since it has an unnecessary *else* branch.

The context string for the discovered buggy code in the example is *(else)*, and the context string for the discovered fix code is ^*(else)*.

Bug Version	Fix Version
if (foo.ready())	if (foo.ready())
foo.startIt();	foo.startIt();
–else	
– foo.init();	

Figure 4-13. Additional buggy code and fix code discovered using IF-RBR pattern.

7. Addition of operations in an operation sequence of method calls to an object (SQ-AMO)

In the SQ-AMO pattern, a missing method call to an object is added to fix a bug. In this case, we have reason to treat the method calls to the same object in the bug version as buggy code. As shown in Figure 4-14, all the lines in the sequence in the bug version, *foo.init()* and *foo.startIt()* are recognized as buggy code, since another method call, *foo.setReady()*, is missing in the bug version.

The context string for the discovered buggy code in the example is <*calls*>*{init,startIt}*, and the context string for the discovered fix code is <*calls*>*{init,setReady,startIt}*..

Bug Version	Fix Version
foo.init();	foo.init();
	+foo.setReady();
foo.startIt();	foo.startIt();

Figure 4-14. Additional buggy code and fix code discovered using SQ-AMO pattern.

8. Removal of operations from an operation sequence of method calls to an object (SQ-RMO)

The opposite of SQ-AMO, SQ-RMO removes an unnecessary method call to an object to fix a bug. In this case, we treat the method calls in the method call sequence in the bug version as buggy code. As shown in Figure 4-14, *foo.init()* and *foo.startIt()* are recognized as buggy code by the SQ-RMO pattern, while *foo.lock()*, which is in the bug hunk, is recognized as buggy code by diff.

The context string for the discovered buggy code in the example is *<calls>{init,lock,startIt}*, and the context string for the discovered fix code is *<calls>{init,startIt}*.

Bug Version	Fix Version
foo.init();	foo.init();
–foo.lock();	
foo.startIt();	foo.startIt();

Figure 4-15. Additional buggy code discovered using SQ-RMO pattern.

9. Addition of operations in a field setting sequence (SQ-AFO)

Similar to SQ-AMO, SQ-AFO recognizes the sequence of field assignment statements to the same object in the bug version as buggy code (See Figure 4-16).

The context string for the discovered buggy code in the example is *<fields>{flag1,flag3}*, and the context string for the discovered fix code is *<fields>{flag1,flag2,flag3}*.

Bug Version	Fix Version
foo.flag1 = 1;	foo.flag1 = 1;
	+foo.flag2 = 1;
foo.flag3 = 1;	foo.flag3 = 1;

Figure 4-16. Additional buggy code and fix code discovered using SQ-AFO pattern.

10. Removal of operations from a field setting sequence (SQ-RFO)

As shown in Figure 4-17, the SQ-RFO pattern recognizes *foo.flag1 = 1* and *foo.flag3 = 1* as buggy code, while *foo.flag2 = 1*, which is in the bug hunk, is recognized as buggy code by diff.

The context string for the discovered buggy code in the example is *<fields>{flag1,flag2,flag3}*, and the context string for the discovered fix code is *<fields>{flag1,flag3}*.

Bug Version	Fix Version
foo.flag1 = 1	foo.flag1 = 1;
–foo.flag2 = 1;	
foo.flag3 = 1;	foo.flag3 = 1;

Figure 4-17. Additional buggy code and fix code discovered using SQ-RFO pattern.

11. Addition or removal of method calls in a short construct body (SQ-AROB)

In this bug fix pattern, a short construct body forms a special context for the method calls in it, and implies that these method calls should be grouped together.

In the example in Figure 4-18, we mark *foo.startIt()* as buggy code, since we believe that it is supposed to exist together with *setReady()* as shown in the fix version.

The construct body types we consider include method body, *if* body, *else* body, *for* body, *do* body, *while* body, *try* body, *catch* body, and *finally* body. We only consider short bodies, i.e. the body that only contains no more than 3 simple statements.

The context string for the discovered buggy code in the example is <*ShortBody*>*{startIt}*, and the context string for the discovered fix code is <*ShortBody*>*{setReady,startIt}*.

Bug Version	Fix Version
void startToWork() {	void startToWork() {

foo.startIt();	setReady();
	bar.startIt();

}	}

Figure 4-18. Additional buggy code and fix code discovered using SQ-AROB pattern.

12. Addition/removal of switch branch (SW-ARSB)

In the SW-ARSB pattern, a *case* branch is removed or added to fix a bug. In this case, we believe that the developer did not consider all the cases of the switch condition when she/he introduced the *switch* predicate in the source code. So, we mark the *switch* predicate as buggy code. Figure 4-19 illustrates an example. There are two reasons that the *case* statements are not treated as buggy code in this bug fix pattern. First, each *case* statement carries little syntactic information, i.e. it usually only contains a constant in a *case* expression, so the *case* statements are not useful in the BugMem approach as described in Section 5. Second, some projects we examined contain *switch* statements that enclose hundreds of *case* statements, so it is not reasonable to regard all these *case* statements as buggy code.

The context string for the discovered buggy code in the example is *(case){1,2,default}*, and the context string for the discovered fix code is *(case){1,2,3,default}*.

Bug Version	Fix Version
switch (result) {	switch (result) {
case 1: startIt(1); break;	case 1: startIt(1); break;
case 2: process(2); break;	case 2: process(2); break;
	+case 3: output(3); break
default: break;	default: break;
}	}

Figure 4-19. Additional buggy code and fix code discovered using SW-ARSB pattern.

13. Addition/removal of try statement (TY-ARTC)

In the TY-ARTC pattern, a try/catch statement is added to enclose a section of code, or a try/catch construct is removed from a section of code. In both cases, we recognized the method calls in the code section as buggy code, since when these method calls were introduced in the source code, no proper exception handling was employed on them. The reason that we only consider method calls is that most exceptions are caused by method calls. In the example in Figure 4-20, *foo.openTheFile()* is recognized as buggy code, since a try/catch statement is missing for it.

There is one problem in the strategy of using TY-ARTC to discover buggy code when the *try* block includes many method calls. In this case, the rule explained above will treat each method call as buggy code. However this will lead to many false positives, since the addition or removal of the *try* statement may be only caused by one method call in the enclosed code. To avoid too many false positives, we defined a threshold value. If the number of method calls is greater than three, we will ignore the buggy code discovered by the bug fix pattern instance.

The context string for the discovered buggy code in the example is ^<*try*>, and the context string for the discovered fix code is <*try*>.

Bug Version	Fix Version
	+ try {
foo.openTheFile();	foo.openTheFile();

	+ } catch (FileOperationException e) {
	+ return;
	+ }

Figure 4-20. Additional buggy code and fix code discovered using TY-ARTC pattern.

14. Addition/removal of a catch block (TY-ARCB)

The TY-ARCB pattern also locates buggy code that has improper exception handling. In the example in Figure 4-21, *foo.doSearch()* is discovered as buggy code since one exception handling catch block is missing. As with TY-ARTC, we ignore buggy code discovered by the bug fix pattern if there are more than three method calls in the *try* block. The number three here is chosen based on a rule of thumb.

The context string for the discovered buggy code in the example is *<catch>{FileOperationException}*, and the context string for the discovered fix code is *<catch>{FileOperationException, MySearchException}*.

Bug Version	Fix Version
try {	try {
foo.doSearch();	foo.doSearch();

} catch (FileOperationException e) {
return;
}
	} catch (FileOperationException e) {
	return;
	+ } catch (MySearchException e) {
	+ reportSQLError();
	}

Figure 4-21. Additional buggy code and fix code discovered using TY-ARCB pattern.

4.4.2 Additional Buggy code Discovery on Open Source Projects

To evaluate how much additional buggy code the bug fix patterns can discover in a software change history, an analysis of additional buggy code discovery was performed on the seven projects listed in Table 3-5. Below is pseudo-code for the process of using bug fix patterns to discover additional buggy code in a project's software change history.

The results of the additional buggy code discovery analysis are shown in Figure 4-23. Bug fix patterns discover 5.0% to 17.4% more buggy code in addition to that computed by diff.

```
Buggy_Code_Discovery(project_revisions)
Input: all the revisions of a project(project_revisions)
Output: a list of buggy statements discovered by bug fix patterns
        (additional_buggy_statement_list)
Begin
   foreach revision r in the project_revisions
     if r is a bug fix revision
       foreach changed file f in revision r
         pre_f = file f in revision r - 1
         delta_list=compute the deltas between pre_f and f by searching the
                  database containing project fact data
         foreach delta d in delta_list
           s = buggy statements discovered by bug fix patterns in the delta d
           Add the statements in s but not in the bug hunk of d
                into additional_buggy_statement_list
         endfor
       endfor
     endif
   endfor
   output additional_buggy_statement_list
End
```

Figure 4-22. Pseudo-code of discovering additional buggy code using bug fix patterns for a project.

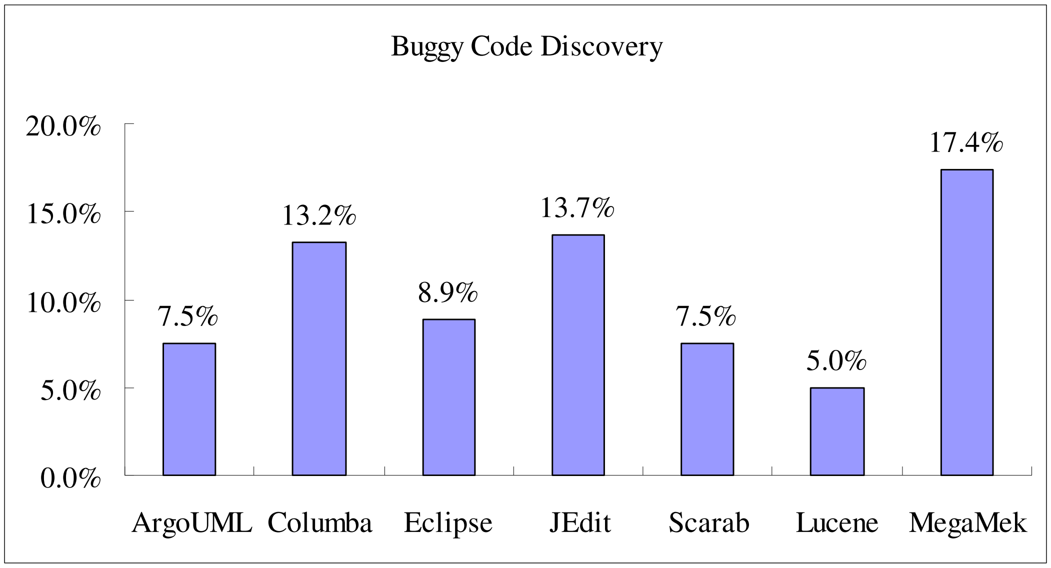


Figure 4-23. Ratio of the buggy LOC discovered by bug fix patterns to the total number of LOC of bug hunks for the analyzed projects.

5 Finding Duplicated Bugs using Change History

Bugs are prevalent in software. As a result, any technique that can automatically detect software bugs and suggest fixes will lead to fewer delivered bugs and improved software quality. Many automatic bug finding tools have been proposed, including Bandera [17], ESC/Java [26], FindBugs [39], JLint [5], and PMD [16]. They use a range of techniques to detect bugs and suggest fixes, including pre-defined bug patterns [5, 39], theorem proving [26], and model-checking [17]. These bug finding tools adopt a horizontal approach, using techniques that are applicable across all projects. To date, there are very few tools using the vertical approach of leveraging patterns in a specific project and performing project-specific bug finding. Recent work using this vertical approach includes DynaMine [57], which focuses on detecting bugs in method usage pairs, and HistoryAware [85] which focuses on return value checking. In this chapter, we present a vertical bug finding approach that extracts and memorizes a broad range of patterns in buggy code, using the previous bug patterns of a specific project to find project-specific duplicated bugs in new changes or other parts of the source code.

One of the common bugs detectable by horizontal bug finding tools is the null dereferencing bug, shown in Figure 5-1. The code tries to reference ‘bar’, when it is null. The correct behavior of the code is to check if ‘bar’ is null, printing the ‘foo’ field only if this is not the case.

```
if (bar==null) {
  System.out.println(bar.foo);
}
```

Figure 5-1. Example null dereferencing bug.

The bug in Figure 5-1 is easily detected using horizontal bug finding techniques, and the null dereferencing bug is one of many kinds of bugs that exist across software projects. However, we believe that there are many project-specific bugs, since different projects have different requirements, business logic, and semantics. Consider two bug fix examples from the Eclipse project, shown in Figure 5-2.

```
JavaProject.java at revision 2024 (Fix for bug
28434)
- if (requiredProjectRsc.exists() &&
-     requiredProjectRsc.isOpen()) {;
+if(JavaProject.hasJavaNature(requiredProjectRsc))
```

```
DeltaProcessor.java at revision 1945 (Fix for bug
27499)
- boolean isOpened=proj.isOpen();
- if (isOpened && this.hasJavaNature(proj))
+ if (JavaProject.hasJavaNature(proj))
```

Figure 5-2. A duplicated bug fix in Eclipse.

The example shows two separate instances in the history of two different files where an incorrect condition check, *isOpen()*, is removed and replaced with the correct condition check, *hasJavaNature()*.

Figure 5-3 shows an example of a duplicated bug fix change in the JEdit project. This example shows two separate instances in the history where the method call *setSelectedText("\t")* is replaced by one to *insertTab()*. The *insertTab()* method has more complex logic than *setSelectedText("\t")*, and contains two *if* branches: one *if* branch just calls *setSelectedText("\t")*, while the other performs some text processing before calling *setSelectedText()*.

```
JEditTextArea.java at revision 86
- setSelectedText("\t");
+ insertTab();
JEditTextArea.java at revision 114
- setSelectedText("\t");
+ insertTab();
```

Figure 5-3. A duplicated bug fix in JEdit.

These examples are representative of a large class of bugs that are project-specific and involve the use of high-level abstractions. These bugs cannot be detected by existing horizontal bug finding tools [5, 16, 17, 26, 76], since these kinds of design or implementation details are usually not formally described, and change over time. The project-specific design and implementation details used to perform the bug fix shown in Figure 5-2 and Figure 5-3 are common knowledge among a project's core developers, and this knowledge remains in their collective memory. For new developers, it is not easy to learn such knowledge, and even core developers sometimes forget, committing the same mistakes over again.

We can learn from previous mistakes to keep project-specific bugs from recurring. A long-developed project usually has a software configuration management (SCM) repository that records a great number of bug fix changes. These bug fix changes record the location of bugs as well as their fixes, the solutions to the bugs. By extracting and saving the code patterns found in buggy code, we can detect potential bugs in new changes or in other parts of the program. Based on this assumption, we build project-specific bug and fix memories from project change histories.

In this dissertation, we use the term “memory” or “memories” to describe a database that stores bug and fix pattern instances extracted from bug fix changes in a project’s development history. An algorithm extracts pattern instances from bug fix changes by parsing, normalizing and filtering the code in the bug or fix area. The parsing step extracts syntax components from the code, the normalization process generalizes the syntax structure for matching similar code, and the filtering step eliminates noise in component matching. These extracted pattern instances are stored in the memory database for matching future bugs. We also develop a bug finding tool, BugMem, for detecting project-specific bugs and suggesting fixes for the bugs using the memories of bug fixes.

After applying our approach on five open source projects, we found that 17.5% - 32.4% of the bugs and 7.5% - 13.0% of bug and fix pairs were duplicated in the history. The results demonstrate that project-specific bug fix patterns occur frequently enough to be useful as a bug detection technique. Furthermore, for the bug and fix pairs, it is possible to both detect the bug and provide a strong suggestion for the fix.

We compared BugMem with a bug finding tool based on a static syntax checker, PMD, and found that the bug sets identified by PMD and by BugMem are mostly exclusive. This indicates that BugMem is not meant to replace conventional bug finding tools, and can be used with other bug finding tools to maximize bug detection.

The remainder of this chapter begins by presenting algorithms to build bug fix memories from project change histories (Section 5.1) and then proceeds to evaluate our approach by checking how well the memories match real bug fixes in a project's change history (Section 5.2). We next describe the BugMem tool (Section 5.3). We discuss limitations of BugMem and ways to improve it in Section 5.4.

5.1 Building Bug Fix Memories

5.1.1 General Description

Building bug fix memories involves a process of extracting syntax components from bug and fix code identified in a project's change history, then storing them in a database. The extracted syntax components will be used to find duplicated bugs in new revisions or in other parts of the source code.

To build memories of bug fixes, we must identify those changes in a software project history where a bug was fixed. We first need to extract project fact data such as source code, change logs, and source code changes (deltas) from a project's SCM repository. This step has been introduced in Section 3 and the architecture is depicted in Figure 3-1. After this, we have all the project fact data for a project's change history in the database. The work of building bug fix memories builds on the project fact data. Figure 5-4 depicts the architecture of the process to build bug fix memories.

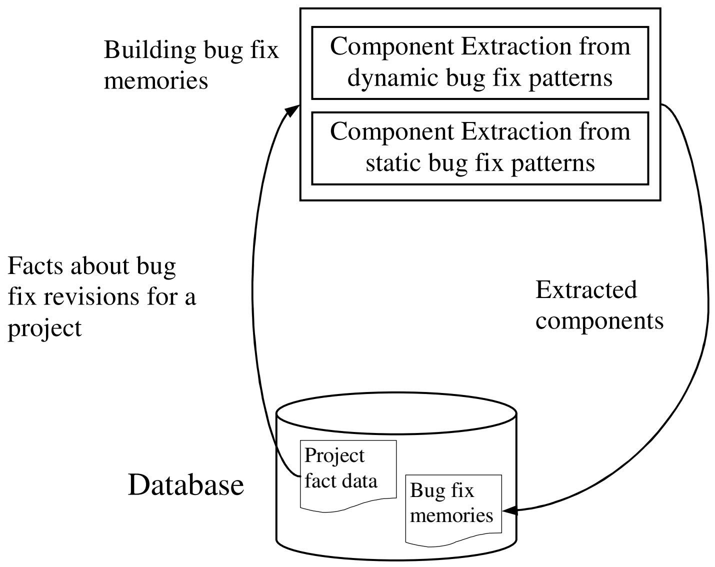

Building bug fix memories
Component Extraction from dynamic bug fix patterns
Component Extraction from static bug fix patterns
Facts about bug fix revisions for a project
Extracted components
Database
Project fact data
Bug fix memories

Figure 5-4. Overview of the process of building bug fix memories.

Generally, to build the bug fix memories, each bug fix hunk pair in a bug fix revision is scanned. A hunk (***H***) consists of a set of contiguous syntax lines (***SL***), i.e. ***H*** = {***SL***}. A hunk pair consists of a deleted hunk (***DH***), representing lines deleted from the prior version, and the corresponding added hunk (***AH***) with lines added in the new version, i.e. ***HP*** = (***DH***, ***AH***). We exclude hunks that include only import statements, since most of these changes do not affect program behavior. We also exclude large hunks that contain more than seven lines of code, since we observed that large hunks will bring noise to the memories.

In bug fix changes, we start by assuming that deleted hunks (***DH***) are bug hunks (***BH***), and added hunks (***AH***) are fix hunks (***FH***), since by deleting the lines in ***DH*** a bug was removed, and by adding the lines in ***AH*** a bug was fixed. Formally, ***BH*** = ***DH*** and ***FH*** = ***AH*** if it is a bug fix change.

Central to the approach is an algorithm for extracting syntax patterns, called components, from hunks, which are saved to the memory database. To extract components from hunks, we parse the entire source code file and extract components. Then we collect only those components that fall into hunks or additional code lines discovered by static bug fix patterns. The extraction process can be expressed as:

extract(***H***) = *extract_dynamic*(***H***) ∪ *extract_static*(***H***),

extract_dynamic(***H***) → {c | c is a component in ***H***}

extract_static(***H***) → {c | c is a component in the additional code lines discovered by the static bug fix patterns for ***H***}.

We currently support only the Java programming language for building bug fix memories, due to the ease of parsing this language. The extraction process consists of two steps. The first one is to extract syntax components from instances of dynamic bug fix patterns, i.e. extracting components from code covered by bug fix hunk pairs. The second step is to extract components from the buggy code and fix code discovered by static bug fix patterns. The component extraction process is detailed in Section 5.1.2 and Section 5.1.3.

The outcome of this process is a set of bug fix memories for a project. The bug fix memories (***M***) are the union of the bug memories (***BM***) and fix memories (***FM***): $\boldsymbol{M} = \boldsymbol{BM} \cup \boldsymbol{FM}$. Bug memories (***BM***) and fix memories (***FM***) are the union of components extracted from the bug hunks or fix hunks in a project:

$$\boldsymbol{BM} = (\bigcup extract\_dynamic(BH)) \cup (\bigcup extract\_static(BH)),$$

$$\boldsymbol{FM} = (\bigcup extract\_dynamic(FH)) \cup (\bigcup extract\_static(FH)).$$

Put another way, to build the memory database for a specific project, we iterate over each bug fix revision and apply the component extraction algorithm. We save the extracted components in the memory database. The overall process for building the bug fix memory database is sketched in Figure 5-5.

```
$\boldsymbol{M}$ = null
for ($r$ = 1 to $N$) {
  get bug and fix hunks in revision $r$
  $\boldsymbol{M} = \boldsymbol{M} \cup extract\_dynamic(\boldsymbol{H}) \cup extract\_static(\boldsymbol{H})$ for
       each bug and fix hunk $\boldsymbol{H}$ in revision $r$
}
```

Figure 5-5. The process for building bug fix memories. The total number of revisions is *N*.

Once constructed, the bug fix memories can be used to detect potential bugs in new or existing source code and provide corresponding fix examples. We define the *find* function, which takes components and memories as inputs and returns a set of matched hunk pairs (***MatchedHP***):

$find(component, M) \rightarrow \{MatchedHP\}$
$find(extract(BH), M) \rightarrow \{MatchedHP\}$
$find(extract(FH), M) \rightarrow \{MatchedHP\}$

The next sections detail the process of building a project's bug fix memory database.

5.1.2 Extracting Components from Dynamic Bug Fix Patterns

The code in bug hunks represents the mistakes developers have made throughout a project's history, while the code in fix hunks contains solutions to these mistakes. We want to learn from the project's history so we can prevent mistakes similar to those we have already observed and provide useful suggestions for how to repair these errors. Intuitively, to achieve this goal we must record the code found in bug hunks as well as the corresponding fixes. But how?

Naively, we can directly save all the code found in a bug hunk for use in future bug detection. Unfortunately, code in a new change typically does not have an exact match with the buggy code in the history. As a result, this approach would miss cases in which new code has a similar, but not equivalent, structure as the stored buggy code. In order to take a full advantage of the buggy code in the history, it is necessary to perform a series of steps that break the code down into its constituent parts, and then abstract these parts into more general patterns.

We developed an algorithm that extracts syntax patterns (components) from the source code lines in the bug hunks or fix hunks. We call these syntax components extracted from bug hunks or fix hunks *dynamic bug fix patterns* in contrast to the pre-defined static bug fix patterns described in Section 4.1. We save the components generated by the algorithm to a database, thereby creating bug fix "memories" that can be used for future bug detection and change suggestions.

In the rest of this section, we explain the component extraction algorithm, which is an implementation of the function, *extract_dynamic*(***H***). The algorithm consists of four steps, raw component extraction, normalization, information filtering, and diff filtering.

5.1.2.1 Raw Component Extraction

In this step, we parse the source code inside a hunk, and burst out the individual syntactic elements we find there.

```
if (foo.flag>=5 && foo.ready()) {
   i=1;
   foo.create("example");
   initiate(5,bar);
}
```

Figure 5-6. Example code in a bug hunk (the type of the variable *foo* is *Foo*, the type of *i* is *int*, and the type of *bar* is unknown).

Figure 5-6 shows an example of code in a bug hunk, which we use as a running example. We extract four basic syntax lines from this code: (1) *if (foo.flag>=5 && foo.ready())*; (2) *i=1*; (3) *foo.create ("example")*; and (4) *initiate(5,bar).*

In our implementation of the component extractor, we developed a parser based on JDT [22] to generate the abstract syntax tree for Java code and the symbol table for variables. This parser only performs a single-pass scan of individual Java files, so types of some variables are unknown to the parser, such as the field variables of external objects. This is the reason why the type of a variable or an expression is sometimes unknown, as with *bar* in Figure 5-6 above.

We extract raw components based on the abstract syntax tree of a basic syntax line. More specifically, we extract the set of non-leaf nodes in a syntax line's abstract syntax tree as its raw components. For example, from the syntax line *if (foo.flag>=5 && foo.ready())* in Figure 5-6, we extract six raw components, which are (1) *foo.flag*, (2) *foo.flag>=5,* (3) *ready()*, (4) *foo.ready()*, (5) *foo.flag>=5 && foo.ready()*, and (6) *if (foo.flag>=5 && foo.ready())*. Note that *foo.flag* is the parent node of leaf nodes, *foo* and *flag*, and we treat string literals as non-leaf nodes.

A raw component can be one of four high-level kinds: *static Java call*, *Java call*, *user-defined call*, or *non-call*. If a component does not represent a method call, it is of the non-call kind. If a component represents a method call, and the method call is an invocation of a Java core class or a static field of a Java core class, it is a *static Java call*. For example, the components

System.out.println("Hello") and *Integer. parseInt("12")* are static Java calls. If a component represents a method call to a user-defined object whose type is in the Java core classes, we call it a *Java call*. For example, given a *String* object *str,* the component *str.length()* is considered to be a *Java call*. A method call to a user-defined method is a *user-defined call*. This classification of components is used in setting component search options, described in Section 5.1.4.

foo.flag	*Foo.flag,*
foo.flag>=5	*Foo.flag>=5*
	Foo.flag>=int
foo.ready()	*Foo.ready()*
ready()	*ready()*
foo.flag>=5 && foo.ready()	*Foo.flag>=5 && Foo.ready()*
	Foo.flag>=int && Foo.ready()
if (foo.flag>=5 && foo.ready())	*if (Foo.flag>=5 && Foo.ready())*
	if (Foo.flag>=int && Foo.ready())
i=1	*int=1*
	int=int
"example"	*"example"*
	String
foo.create.() "example"	*Foo.create(.) String*
create(.)"example"	*create(.) String*
initiate(,) 5, bar	*initiate(,) int, **

Figure 5-7. Raw components (left) and components after normalization (right).

The data for a component consists of a component string and an actual parameter list. A non-call component has only component string data. For example, the component string for *foo.flag* is "*foo.flag*". For a method call component, we represent the method name and actual parameter list separately. The component string for a method call component carries method name and parameter number information. For example, the component string for method calls *initiate()*,

initiate(5), and *initiate(5, 9)* are *initiate()*, *initiate(.)*, and *initiate(,)*respectively. The actual parameter list for *initiate(5, 9)* contains “5” and “9”. We represent method call components in this way to support component matching, discussed further in Section 5.1.4. All the raw components extracted from the example code in Figure 5-6 are shown in the left part of Figure 5-7. Note that for each method call component there is an actual parameter list following the component string.

5.1.2.2 Normalization

The purpose of extracting components is to discover characteristics from the code in a bug or fix hunk so they can be used to match similar characteristics in new changes. For example, suppose we have *foo.execute()* in the component database. The component *bar.execute()* in a new change should match if variables *foo* and *bar* are of the same type.

To extend the possibility of matching similar code, we perform normalization on the extracted raw components. The normalization process follows the rules below.

1. If we know the type of a variable in a raw component, we normalize this variable to its type in the resulting component. For example, given an object *foo* of type *Foo*, the raw component *foo.flag* will be normalized to *Foo.flag*. If the type is numeric (i.e. *int*, *float*, *double*, etc.), we further normalize it to *int*. Variables with unknown types are not normalized.

2. For a raw component that contains a numeric, boolean, or char literal, we generate two components: one without normalization for the literal and one with the literal replaced by *int*, *boolean*, or *char* accordingly. For example, two components, *int=5* and *int=int*, will be generated for the raw component *i=5*.

3. For a raw component that contains a string literal, we generate two components: one without normalization that includes the original string literal and one with normalization where the string literal is replaced with *String*. For example, two components, *String="example"* and *String=String* will be generated for the raw component *str="example"*.

4. For a method call, we normalize each actual parameter to the type of the parameter. If we are not able to figure out the type of a parameter value, we use * to indicate that this parameter can be any type. For example, the component string for component *initiate(5,bar)* is *initiate(,)* and the parameter list of this component is *int* and *, supposing the type of *bar* is unknown.

Figure 5-7 shows the resulting components after normalizing the raw component list in the left hand side of the figure.

5.1.2.3 Information Filtering

After normalization, the resulting components are candidate components for storage in the database. One problem that arises after the normalization step is that commonly occurring statement types are normalized to commonly occurring components. For example, integer assignment statements will generate *int=int,* which is very common. We do not want these common components to be added to the database, since they will cause many false alarms. Therefore, we filter out components such as these that carry little unique information.

Detailed Element	**Condition**	**Information Value**	**Example**
if predicate	Construct	1	if ()
do predicate	Construct	1	while ()
while predicate	Construct	1	while ()
for expression	Construct	1	for ()
Conditional expression	Construct	1	i>0? i: 1
return statement	Construct	1	return i
case expression	Construct	1	case 5:
switch expression	Construct	1	switch ()
synchronized expression	Construct	1	synchronized ()
throw statement	Construct	1	throw new Exception()
string literal	Length>8	2	"compiler.problem.Messages"
string literal	Length between 3 and 8	1	"example"
numeric literal		1	10
method call		2	initiate()
class name or variable type	User-defined class	1	Foo
variable name or field name		1	flag
Other	Does not match any other category	0	int=int

Table 5-1. The information value for detailed elements in components.

A component's *information value* indicates how much unique information it carries. The information value for a component is determined by summing the information values of its constituent elements. Table 1 lists the information value for different kinds of syntax constructs, identifiers and literals. For example, the information value for the component *if (Foo.flag>=int && Foo.ready())* is 6: the *if* construct counts for 1, the *Foo* before the field access *flag* counts for 1, *flag* counts for 1, the *Foo* before *ready()* counts for 1, and the method call *ready()* counts for 2.

We filter out components possessing little unique information by defining an information value threshold, that is, we only keep components whose information value is greater than or equal to 2. Following this rule, we filter out the four components *int=1* (information value of 1, from the numeric literal 1), *int=int* (information value 0, an "other"), *"example"* (information value 1, string literal), and *String* (information value 0, an "other") from the resulting components listed in the right hand side of Figure 5-7, since their individual information values are less than 2.

5.1.2.4 Diff Filtering

After the information filtering step, we obtain a list of components that pass the threshold for carrying sufficient unique information from the code in the bug hunk. A further filtering step is to determine the components that exist in the bug hunk but not in the fix hunk. That is, we do not save the code characteristics that are common between the bug hunk and the fix hunk, since they are unchanged.

Bug Hunk

```
-if (foo.flag>=5 && foo.ready()) {
-   i=1;
-   foo.create("example");
-   initiate(5,bar);
```

Fix Hunk

```
+if (foo.flag>=7 || foo.ready()) {
+   create("example");
+   initiate(5);
```

Figure 5-8. Example code in a bug hunk and the corresponding code in the fix hunk.

```
Foo.flag>=5
Foo.flag>=5 && Foo.ready()
Foo.flag>=int && Foo.ready()
if (Foo.flag>=5 && Foo.ready())
if (Foo.flag>=int && Foo.ready())
Foo.create(.)    String
initiate(,)       int, *
```

Figure 5-9. Components in the bug hunk but not in the fix funk for the example in Figure 5-8.

Figure 5-8 shows example code in a bug hunk and its corresponding code in a fix hunk. Figure 5-9 shows the resulting components after diff filtering the components from the code in Figure 5-8. We can see that several components have been filtered out in this step. One example is the component *foo.ready()*, which exists in both the bug hunk and the fix hunk. The components remaining at the end of this step are saved to the database. For the example in Figure 5-8, the components listed in Figure 5-9 will be saved to the memory database.

5.1.3 Extracting Components from Static Bug Fix Patterns

The procedure described in Section 5.1.2 extracts components from bug fix hunks, but not all buggy code or fix code is covered by this process. As we have discussed in Section 4.4, some of the static bug fix patterns can be used to discover additional buggy code and fix code. So, in this section, we describe a process for extracting components from buggy code and fix code discovered by bug fix patterns, and then saving them to the memory database.

The process used to extract components using bug fix patterns is described as below.

1. For a bug fix hunk pair, use the rules described in Section 4.2 to check whether it contains instances of any of the following bug fix patterns: IF-APC, IF-APCJ, IF-APTC, IF-RMV, IF-ABR, IF-RBR, SQ-AMO, SQ-RMO, SQ-AFO, SQ-RFO, SW-ARSB, TY-ARTC, and TY-ARCB. If no instances are found, stop and return, since there is no additional buggy code or fix code for this bug fix hunk pair.
2. For each bug fix pattern instance identified from Step 1, use the approach described in Section 4.4.1 to discover additional buggy code and fix code.
3. For the additional discovered code, perform raw component extraction (Section 5.1.2.1), normalization (Section 5.1.2.2), and information filtering (Section 5.1.2.3) to extract components that carry context information corresponding to the bug fix pattern.

4. Save the extracted components to the memory database.

Note that the components extracted during this stage carry context information for the corresponding bug fix pattern, while the components extracted from dynamic bug fix patterns (Section 5.1.2) do not. The context expression for corresponding bug fix patterns is described in Section 4.4.1.1.

5.1.4 Storing and Searching Memories

Section 5.1.2 and 5.1.3 describe the algorithm for extracting components from the bug fix hunk pairs. We apply this algorithm to all of the bug fix hunk pairs and save the resulting components to the memory database. Each component recorded in the database table contains the fields listed in Table 5-2.

Field	**Description**
Component_string	The component string.
Kind	The kind of the component: static Java call, Java call, user-defined call, or non-call.
Parameter_list	Parameters for method calls. Components of non-call kind leave this field empty.
Information _value	The information value of this component.
In_bug_hunk	If the component is extracted from a bug hunk, this value is true; otherwise, false.
File_name	The path of the file containing this component.
Revision	Revision number containing the component.
Delta_id	The id for the delta that contains this component.
Context	The context string for the components extracted from static bug fix patterns.

Table 5-2. Database table for recording components.

Using a populated memory database, it is possible to perform bug detection and change suggestion. Duplicated bugs are found by searching for matching patterns in bug hunks, while change suggestions are made by returning the code in the corresponding fix hunk. Figure 5-10 sketches the component searching algorithm. The input is a given component and its context list, and the output is a list of deltas that contain the component matching the given one. Component matching is mainly based on the *Component_string*, *Parameter_list*, and *Context* information in the database.

From the pseudo-code in Figure 5-10, component searching against the components that are extracted from dynamic bug fix pattern is straightforward: only the component string and parameter list are needed for comparison. For example, suppose the component *if (Foo.flag>=int && Foo.ready())* is found in a new change. We search the database for records whose *Component_string* value is also *if (Foo.flag>=ubt && Foo.ready())* and whose *In_bug_hunk* value is *true*. If we find any matching records, we can alert the developer that the new change may contain a bug. We can additionally provide developers with suggestions on how to fix the bug by presenting the fix hunk code.

```
Component_Matching(component, context_list)
Input: a component to search against the memory database, and the context list for
        the statement containing the component.
Output: a list of deltas that contain a component matching the given one.
Begin
    components_in_memories = A list of all records in the database whose
        Component_string value is the same as the component string property of
        component and In_bug_hunk value is the same as component's In_bug_hunk
        property.
    resulting_delta_list is empty.
    foreach component c in components_in_memories
        parameters_match = true;
        context_match = true;

        if c is a method call type and c's Parameter_list value does not match
                component's parameter list
            parameters_match = false;
        endif

        if c is a component discovered by static bug fix pattern, and c's Context value
                does not match component's context_list
            context_match = false;
        endif

        if parameters_match is true, and context_match is true
            Add the delta that contains c to resulting_delta_list;
        endif
    endfor
    return resulting_delta_list;
End
```

Figure 5-10. Pseudo-code for component searching.

We note that to match method call components, we need to compare the parameter list as well as the component string. We additionally need special handling to match parameter types recorded as * in the parameter list, indicating that they match any parameter type.

Searching against the components extracted from static bug fix patterns needs the context list information for the component to be searched and one more step to check context match. For example, suppose that the component *Foo.bar=int* is from the code:

```
void doSomeThing() {
    if (foo != null)
        return;
    foo.bar=5;
}
```

and we want to search the component *Foo.bar=int* in the memory database. By checking the source code around the statement *foo.bar=5*, we can see that the context list for the component *Foo.bar=int* includes the following context strings: <if>, (if), ^<try>, and <jump>. Suppose there is a component in the database, whose Component_string value is *Foo.bar=int* and Context value is ^<jump>. Then it is not a match, since the component *Foo.bar=int* in the memory database was originally extracted from a statement that does not have a jump statement enclosed by an *if* condition before it. Figure 5-11 shows another example of component matching with context information. The statement *buffer.endCompoundEdit()* in both revisions 71 and 145 is buggy code discovered by the IF-APC pattern. In revisions 71 and 145, this statement's context strings are both ^<if>, since there is no precondition check. In versions 72 and 146, this statement's context strings both become <if>. So, we say the component *buffer.endCompoundEdit()* in the both the bug hunk and the fix hunks in revision 146 match those in revision 72.

```
Macros.java at revision 146, project jEdit
+ if (buffer.insideCompoundEdit())
      buffer.endCompoundEdit();
```

```
EditPane.java at revision 72, project jEdit
+ if (buffer.insideCompoundEdit())
      buffer.endCompoundEdit();
```

Figure 5-11. Example of component matching with context.

There are several options for component searching, which adjust the degree of exact/close matching and omission of very common components. The options are listed in Table 5-3.

Option	Description
0	Exclude all static Java call or Java call component kinds.
1	Exclude all static Java component kinds.
2	Search in all the components.

Table 5-3. Options for component searching.

Option 0 has the strictest matching rule for component searching. This option searches components that are not of *static Java call* kind or *Java call* kind. The component *System.out.println()* is a commonly occurring example of a static Java call component. The component *String.length(),* normalized from *str.length(),* is an example of a Java call kind, also a common pattern. Option 0 ignores components such as *System.out.println()* and *String.length()* to avoid the false positives they cause, since we believe developers typically do not make mistakes in these kinds of components. Option 1 is less strict than option 0, since it does not ignore Java call components, such as *String.length()*. Option 2 provides the most permissive component search, searching all components in the database. Option 2 yields the highest hit rates for component searching, but at the cost of more false positives.

5.2 Evaluation

Before diving into the details of how we evaluate the effectiveness of bug fix memories, it is important to understand how they are intended to be used within a software project. A developer working on a project in their favorite development environment will receive feedback whenever the code they are developing matches one of the stored bug patterns. Our tool for performing this matching is called BugMem. The memory database they are querying is *always up to date*. This is due to the inclusion, at checkin time, of new bug fix information. Since the component extraction process is computationally inexpensive, and requires no manual intervention, it is integrated into post-checkin processing for a project. As a result, the bug fix memories can be viewed as a kind of on-line learning algorithm.

Since the intended use of the memories is to find bugs and suggest changes in the current revision by leveraging the information in *all* prior revisions, traditional approaches for evaluation are ill suited for this situation. In a project with *n* revisions, a typical approach would be to train on 90% of the revisions and then evaluate on the remaining 10%. It is also possible to pick different parts of the project to be the 90% by cycling the 10% evaluation set through different portions of the revision history – k-fold cross-validation [65]. The drawback is that they don't reflect the actual use conditions of the memory database, since in normal use the revisions in the evaluation set would contribute to training the memories.

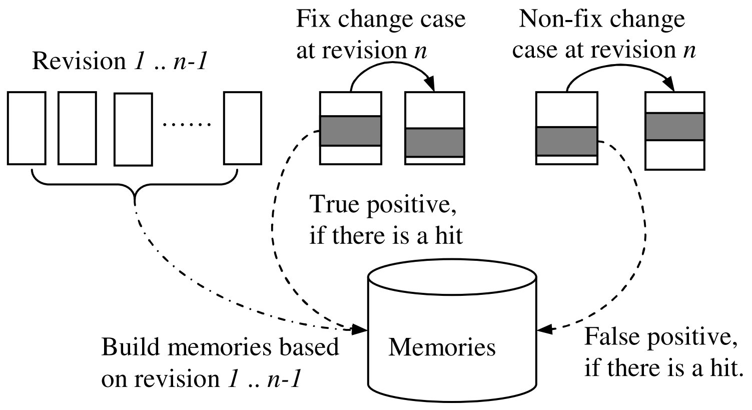

Revision *1* .. *n-1*
Fix change case at revision *n*
Non-fix change case at revision *n*
True positive, if there is a hit
Memories
Build memories based on revision *1* .. *n-1*
False positive, if there is a hit.

Figure 5-12. Evaluation of true positives and false positives.

Another common approach for evaluating bug finding techniques is to train a model on one project, then evaluate it on another. This is also ill-suited for evaluating the bug fix memories. Since a project's memories are comprised of source code patterns, it is inherently specific to that project. In general, evaluation of vertical bug finding techniques involves training on one project and then assessing performance on that same project.

The approach we have chosen for our evaluation is sketched in Figure 5-12. We walk through the revision history of a project, evaluating at each revision how well the approach works when *using only the information available as of that revision*. To evaluate the bug memories at revision *n*, we build the memories using the bug fix hunks from revisions *1* to *n-1*. We then determine whether revision *n* is a bug fix. If so, we check if a component in the bug hunk at revision *n* is found in the memory database. If found, we call it a *true positive hit*, which means that the bug is found in the previous memories. If revision *n* is a non-fix change, and a component in the non-bug hunk is found in the memories, we call it a *false positive hit*, since code that does not contain a bug matches the bug memories. Hit rates are used to evaluate the bug finding and suggestion generation capabilities of BugMem.

We built bug fix memories for five open source projects, ArgoUML, Columba, Eclipse, JEdit, and Scarab. Table 5-4 summarizes the analyzed projects.

Project	**Period**	**Number of revisions**	**Number of deltas**	**Number of bug fix deltas (%)**	**Number of components in bug memories**	**Number of components in fix memories**
ArgoUML	01/1998 ~ 09/2005	4,685	63,505	11,542 (18.2%)	46,960	58,980
Columba	11/2002 ~ 12/2005	2,362	26,370	3,684 (14.0%)	11,977	15,816
Eclipse	06/2001 ~ 01/2006	6,394	83,582	28,138 (33.7%)	88,762	109240
JEdit	09/2001 ~ 01/2006	1,190	22,694	6,797 (30.0%)	22,876	29,714
Scarab	12/2000 ~ 02/2006	2,962	18,254	3,118 (17.1%)	12,561	14,631

Table 5-4. Analyzed open source projects.

We also compare our approach to a horizontal bug finding tool, PMD [16]. This permits us to evaluate how well BugMem and PMD perform at finding the actual bugs in our projects, and whether they find the same kinds of bugs. This is described in Section 5.2.2.

5.2.1 Bug Fix Memory Hit Rates

As described above, we compute hit rates at revision *n* by searching for matches in the memories built from revision *1* to *n-1*. To precisely describe this process, we add notations to the bug memories, *M*, indicating which revisions have contributed information to the memories. Specifically, we introduce the notion of per-revision memories, M_i, representing only those components extracted from revision *i*. We then use this to define the memories as of a given revision, *n*:

$$M^n = \bigcup_{i=1}^{n} M_i$$

This allows us to refine the *find* function to describe searches at a given revision, *n*:

$$find_n(extract(H), M^{n-1}) \rightarrow \{MatchedHP\}$$

This states that the find function for revision *n* only uses memories built using components from revisions *1* to *n-1*.

We extract all bug and fix hunks at revision *n*, and check if the hunks match against M^{n-1}. We iterate the process from revision 1 to the end revision, *N*. The overall process is sketched in Figure 5-13.

To determine half and full hit rates, we start by extracting matched hunk pairs from the memories:

$$Half\_hit\_MatchedHP_n(HP) \ = \ find_n(extract(DH), M^{n-1})$$

```
For ($n = 1$ to $N$) {
    If (revision is a bug fix) {
        Get bug or fix hunks in revision $n$
        Perform $find_n(extract(H), M^{n-1})$ for each bug and fix hunk in revision $n$
    }
}
```

Figure 5-13. Process for evaluating hit rates. The total number of revisions is *N*.

If a component in a bug hunk is found in the memories, we define it as a half hit. A half hit indicates that we have seen the same kind of bug in previous revisions (memories):

$$Half\_hit_n(HP) \ \ iff \ \ Half\_hit\_MatchedHP_n(HP) \neq \Phi$$

If a component in a fix hunk is also found in *Half_hit_MatchedHP$_n$(HP)*, it is a full hit. A full hit indicates that we have seen the exact bug and fix pair in previous revisions (memories).

$$\begin{aligned} &Full\_hit_n(HP) \ iff \ Half\_hit_n(HP) \wedge \\ &\qquad (find(extract(AH), Half\_hit\_MatchedHP_n(HP)) \neq \Phi \vee \\ &\qquad type(HP) = deletion) \end{aligned}$$

More precisely, in order to have a full hit, a half hit must have occurred first. Additionally, the added (fix) hunk must be found in the bug and fix hunk pairs returned by the half hit database query (*find(…)*). In the case where there is no added hunk (code was deleted, but not added when fixing a bug), the hunk pair is of type deletion (*type(HP)=deletion*). Since it is not possible to find an empty added hunk in the half hit hunk pairs, we assume that a half hit is a full hit in this case.

Table 5-5 shows the full and half hit rates of analyzed projects. Using different component search options yields different hit rates. We use option 2 to yield the results in Figure 5-14, which represent the highest possible hit rates. Full hits vary from 7.5% - 13.0%, indicating that this many bug and fix pairs repeat over the project's history. Half hit rates vary from 17.5% - 32.4%, indicating that this many bug hunks are found in previous revisions (memories).

We also computed the average number of matching bug hunks for a half hit or full hit in the Scarab project and the JEdit project. The results show that there are about 9.8 matching bug hunks for a half hit and 2.6 suggested fixes for a full hit in the Scarab project, and about 3.3 matching bug hunks for a half hit and 1.6 suggested fixes for a full hit in the JEdit project.

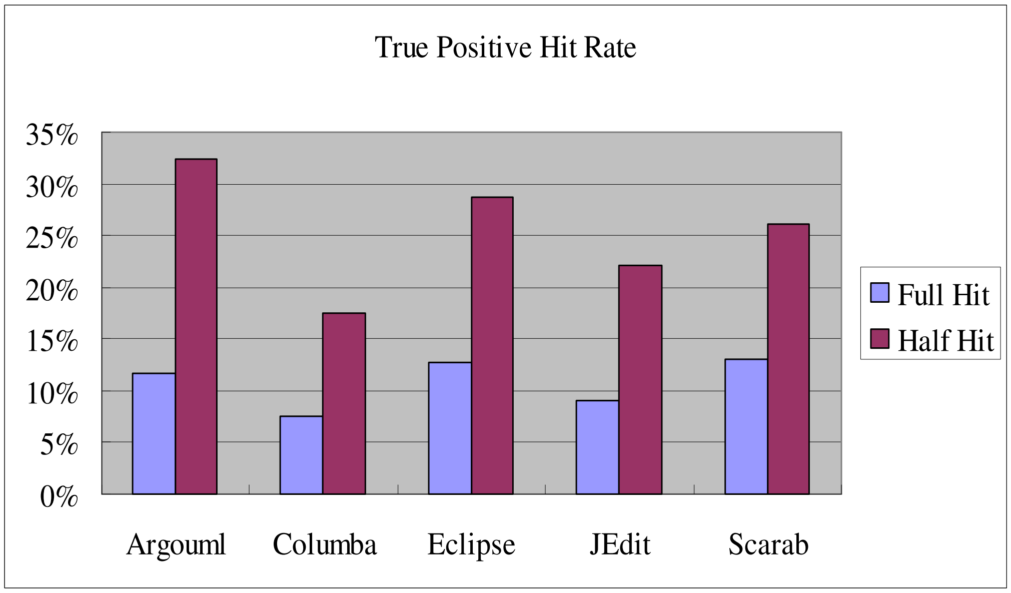


Figure 5-14. Full and half hit rates, search option 2.

From the developer's perspective, the half hit rate indicates that in 17.5% - 32.4% of bug fix changes, the BugMem tool can find code in the change that matches an existing pattern. These matching lines can be highlighted for the developer as code that is likely to be buggy. Additionally, the full hit rates indicate that for 7.5%-13.0% of the changes, it is possible for the BugMem tool to provide suggested changes, observed from the development history, to fix the identified bug.

To get a sense of the impact of different search options, we evaluated the hit rates for all search options. Table 5-5 shows detailed full and half hit rates for search options 0 - 2. We can see that option 0 has the smallest hit rates, and option 2 secures the largest hit rates. But we cannot say option 2 is preferable to option 0 in practice. A user study to evaluate the best option for different projects remains future work.

Option / Projects	0	1	2
Argouml	7.7% / 26.1%	8.0% / 26.8%	11.7% / 32.4%
Columba	5.1% / 13.1%	5.5% / 14.1%	7.5% / 17.5%
Eclipse	10.4% / 23.8%	10.8% / 24.4%	12.8% / 28.7%
JEdit	5.3% / 15.4%	5.7% / 16.6%	9.0% / 22.1%
Scarab	5.5% / 20.0%	5.9% / 21.0%	13.0% / 26.1%

Table 5-5. True positive hit rates.

Hit rates in Table 5-5 are average hit rates for all the bug fix changes in a project's entire change history. But we can expect that the hit rates in the initial stage of a project should be lower than in the late development stage of the project. To examine how hit rate varies over the life of a project, we observed true positive half hits (option 2) over revisions as shown Figure 5-15. The hit rates dramatically increase around revision 200-900 and then continue growing slowly as revisions accumulate.

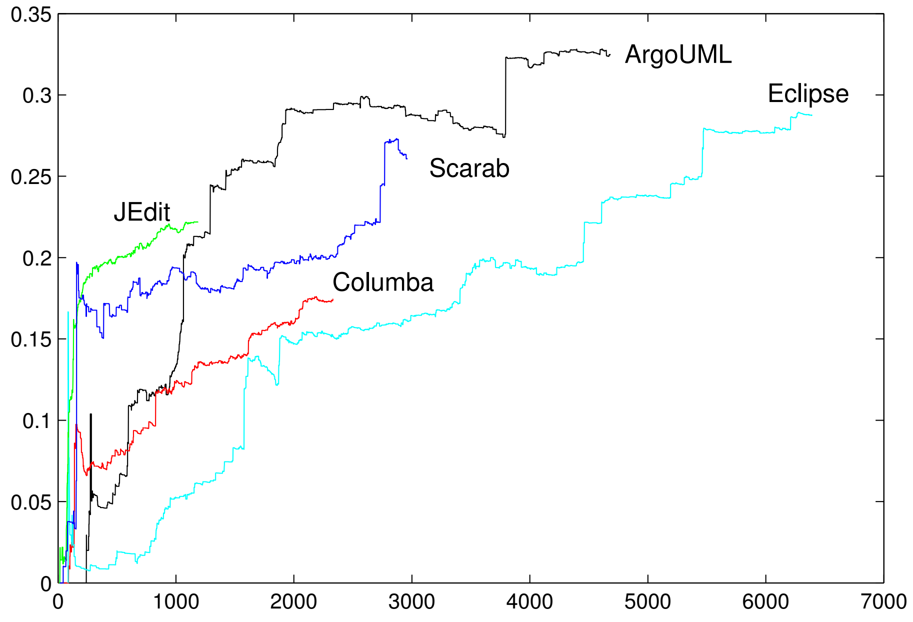


Figure 5-15. True positive half hit rates (option 2) over revisions.

To get a sense of the false positive rates of BugMem, we repeated the analysis shown in Figure 5-13, this time using non bug fix hunks. If components in the non bug fix hunks are found in the memories, such components are false positives, since those hunks are not bug fixes, and hence not supposed to match any components in the memories. Figure 5-16 shows an overview of the hit rates.

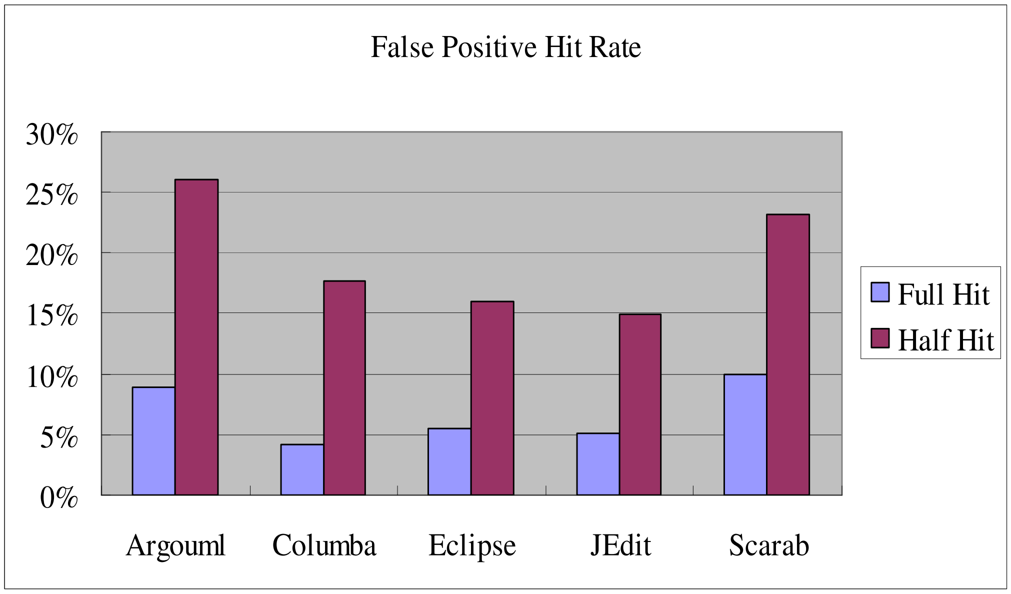


Figure 5-16. False positive hit rates, search option 2.

Table 5-6 shows detailed false positive full and half hit rates for search options 0-2.

Option / Projects	0	1	2
Argouml	5.4% / 19.8%	5.6% / 20.4%	8.9% / 26.1%
Columba	2.2% / 10.9%	2.5% / 11.6%	4.2% / 17.7%
Eclipse	4.2% / 12.9%	4.4% / 13.2%	5.5% / 16.0%
JEdit	3.0% / 10.2%	3.2% / 10.8%	5.1% / 15.0%
Scarab	6.0% / 16.6%	6.5% / 17.2%	10.0% / 23.2%

Table 5-6. False positive hit rates.

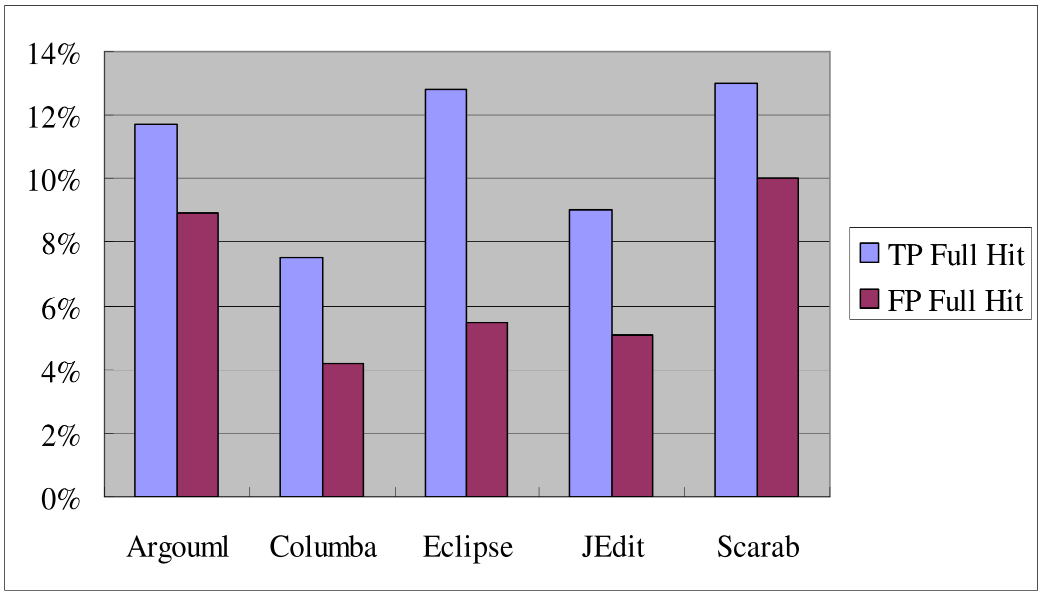


Figure 5-17. True positive (TP) and false positive (FP) full hit rates of analyzed projects.

We compared the true positive (TP) full hit rates and false positive (FP) full hit rates, with results in Figure 5-17. Overall, the false positive full hit rates range from 4.2%-10.0%, which is 2.8% - 7.3% lower than the true positive hit rates. Even though the false positive hit rates seem relatively high, BugMem is still useful, since it provides not only warning flags, but also bug fix examples. Developers can quickly decide if they want to accept or reject the warnings by examining the provided examples. Holmes et al. [34] and Mandelin et al. [59] show that providing code examples is beneficial for understanding software. BugMem always provides bug fix examples along with warnings.

5.2.2 Comparison with PMD

To compare a horizontal bug finding tool with the vertical approach used by BugMem, we identify bugs using a bug finding tool, PMD [16], comparing them with those identified by BugMem. We choose PMD because it does not require annotation and only needs Java source code as its input. Most other bug finding tools such as FindBugs or JLint take Java class files as their input, which would require source code compilation for every revision. This is computationally very expensive.

PMD can identify potential bugs using pre-defined syntactic error patterns such as 'empty if statement', 'misplaced null check', or 'no null check in the equal method' [16]. We locate potential bugs using PMD and check if the bugs are fixed in the project's change histories to compute its hit rate.

To compute full and half hit rates, we ran PMD and obtained detected violations that fall into a hunk. We define the violation count of hunks as *VC(**H**)*. For example, if a hunk includes 5 violations, *VC(**H**)* returns 5. We extracted all bug and fix hunks and measured violation counts for each hunk. The process is sketched in Figure 5-18.

```
For ($r$ = 1 to $N$) {
   Get bug fix hunks in revision $r$
   Compute $VC(\boldsymbol{H})$ for each bug and fix hunk in revision $r$
}
```

Figure 5-18. Process for evaluating PMD hit rates. The total number of revisions is *N*.

A half hit occurs if the violation count of a bug hunk is greater than 0, since PMD correctly identifies some bugs in the bug hunks. To be a full hit, the violation number of a bug hunk must be reduced in the fix hunk. A full hit indicates that there are violations in the bug hunk, but the violations are removed in the fix hunk. If the hunk pair type is deletion and there is a hit on the bug hunk, we assume it as a full hit, since the violated code is removed. Formally, half and full hits are defined as:

$$\begin{aligned}&Half\_hit(HP) \text{ iff } VC(AH) > 0\\&Full\_hit(HP) \text{ iff } Half\_hit(HP) \wedge\\&\qquad (VC(DH) < VC(AH) \vee type(HP) = deletion)\end{aligned}$$

Identified PMD violations are given priorities, ranging from 1 to 3. A priority of 1 indicates a serious warning, while a priority of 3 reflects less important warnings. We observe hit rate variances by using warnings at a specific priority level. Detailed full and half hit rates for each project with priority combinations are shown in Table 5-7.

Priority / Projects	1	2 or less	all
ArgoUML	0.09% / 0.23%	1% / 4.1%	1% / 4.1%
Columba	0.05% / 0.22%	0.9% / 3.6%	0.9% / 3.6%
Eclipse	0.09% / 0.21%	0.5% / 3.9%	0.5% / 3.9%
JEdit	0% / 0.19%	0.07% / 3.7%	0.7% / 3.7%
Scarab	0.06% / 0.22%	1.6% / 4.8%	1.6% / 4.8%

Table 5-7. PMD hit rates of analyzed projects. The first number indicates the full hit rate, the second indicates the half hit rate.

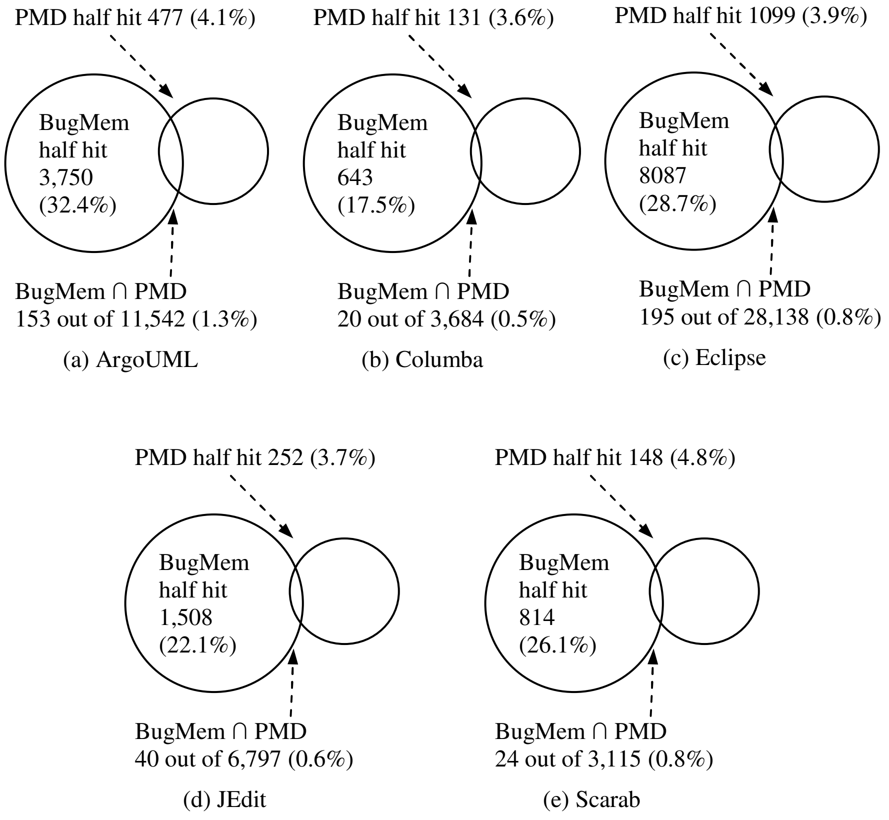

PMD half hit 477 (4.1%)
BugMem half hit 3,750 (32.4%)
BugMem ∩ PMD 153 out of 11,542 (1.3%)
(a) ArgoUML
PMD half hit 131 (3.6%)
BugMem half hit 643 (17.5%)
BugMem ∩ PMD 20 out of 3,684 (0.5%)
(b) Columba
PMD half hit 1099 (3.9%)
BugMem half hit 8087 (28.7%)
BugMem ∩ PMD 195 out of 28,138 (0.8%)
(c) Eclipse
PMD half hit 252 (3.7%)
BugMem half hit 1,508 (22.1%)
BugMem ∩ PMD 40 out of 6,797 (0.6%)
(d) JEdit
PMD half hit 148 (4.8%)
BugMem half hit 814 (26.1%)
BugMem ∩ PMD 24 out of 3,115 (0.8%)
(e) Scarab

Figure 5-19. Identified bug hunk sets using PMD and BugMem.

We compared the bug sets correctly identified by PMD and BugMem (half hits) to see if the two sets are exclusive. The results in Figure 5-19 show that the bugs identified by PMD and those by BugMem are very exclusive. We explain the results by examining ArgoUML and Eclipse as examples.

Figure 5-19 (a) shows the entire bug hunk space of ArgoUML, comprised of 11,542 bug hunks. Among them, 3,750 (32.4%) of the bug hunks were correctly identified by BugMem (see Table 5-5, half hit at option 2), and 477 (4.1%) were correctly identified by PMD (see Table 5-7, half hit at priority 3 or less). For a fair comparison, we use the options in BugMem (option 2) and PMD (priority 3 or less) that can achieve the most hits. We then observed the intersection of the two correctly identified sets to see how these two tools can complement each other. Surprisingly, only 1.3% of the total identified hunks were common between PMD and BugMem.

Figure 5-19 (b) shows identified bug hunk sets by BugMem and PMD in Eclipse. Similarly, only 0.8% of the identified hunks were common in Eclipse. Intersections for other projects are shown in Table 5-8. The intersections of the identified bugs are about 0.5~1.3% of the total bug hunks. These results indicate that the identified hunk sets by PMD and BugMem are nearly mutually exclusive. We conclude that BugMem is not meant to replace prior bug finding tools. BugMem can find bugs that cannot be identified by PMD and vice-versa. There is considerable synergy in using a combination of vertical and horizontal bug finding tools together.

	BugMem hit (%)	**PMD hit (%)**	**BugMem∩PMD (%)**
ArgoUML	3750 (32.4%)	447 (4.1%)	153 (1.3%)
Columba	643 (17.5%)	131 (3.6%)	20 (0.5%)
Eclipse	8087 (28.7%)	1099 (3.9%)	195 (0.8%)
JEdit	1508 (22.1%)	251 (3.7%)	40 (0.6%)
Scarab	814 (26.1%)	148 (4.7%)	24 (0.8%)

Table 5-8. Identified bug hunks (half hit) by PMD, BugMem, and their intersection.

5.3 Using Bug Fix Memories

Bug fix memories can be used to construct a bug finding tool and perform IDE integration. A project's bug fix memories can also be used as a code example repository to provide awareness to developers. We describe the possible applications of the BugMem approach in detail.

5.3.1 Bug Finding Tool

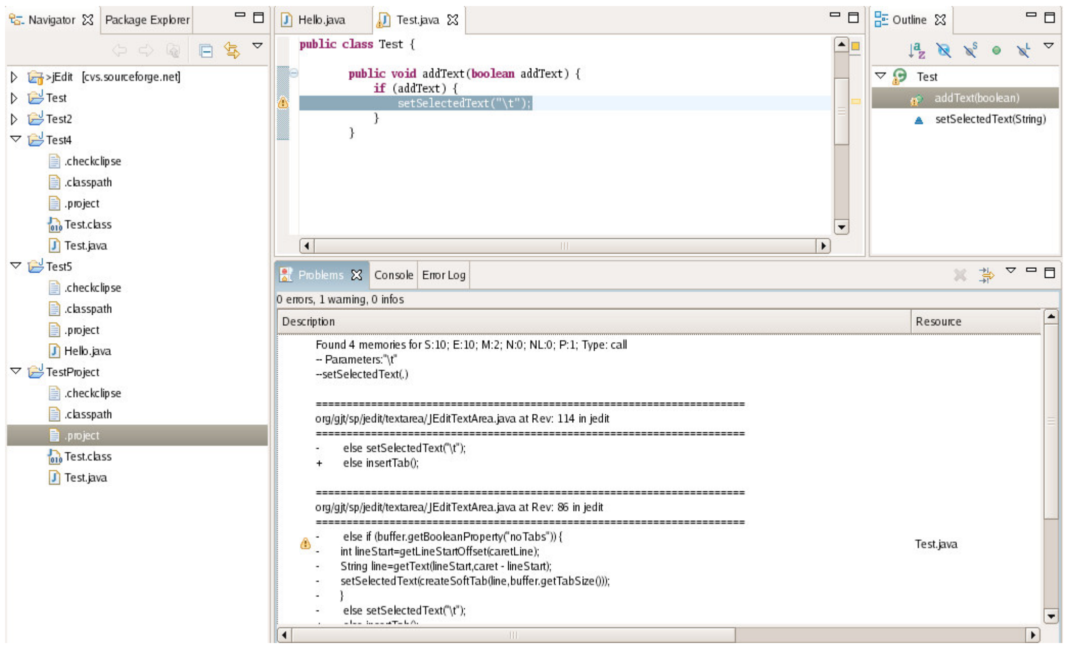
We implemented a proof-of-concept bug finding tool, using the memories of bug fixes approach. Like other bug finding tools such as ESC/Java, FindBug, and PMD, the BugMem tool is provided source code and generates warning messages, as shown in Figure 5-20. BugMem also provides real fix examples for identified bugs using fix data from the project's bug fix memories.

```
$ bugmem Test.java
Warning in addText at Test.java  (line 10)   Found 4 memories
Type: call "setSelectedText(.)"
===============================================================
=
org/gjt/sp/jedit/textarea/JEditTextArea.java at Rev: 114 in jedit
===============================================================
=
                    -     else setSelectedText("\t");
                    +     else insertTab();
 ……
```

Figure 5-20. Simple output of the BugMem command line tool.

5.3.2 Suggestion and Confirmation

We also implemented a proof-of-concept IDE (Eclipse) integration of BugMem, shown in Figure 5-21. During a source editing session using the IDE, BugMem can point out potential bug lines and provide real bug fix examples for those lines. Additionally, the tool can confirm the correctness of source code by assuming that fixed source code in the change history is correct. For example, consider the bug fix hunk in Figure 5-3. From the fix hunk, we can assume that '*setSelectedText()'* is wrong and '*insertTab()*' is the correct method to call. A developer who is not aware of this fact may use '*setSelectedText()*' in her code. The BugMem IDE integration points out the potentially buggy source code and provides fix examples from the project memories, as shown in Figure 5-21. In addition to that, BugMem learns that '*insertTab()*' is correct from memories such as the example in Figure 5-3. Whenever BugMem finds the components in the fix memories (***FM***), it confirms the code to be the correct source code.



```
public class Test {

    public void addText(boolean addText) {
        if (addText) {
            setSelectedText("\t");
        }
    }
```

```
0 errors, 1 warning, 0 infos
Description
Found 4 memories for S:10; E:10; M:2; N:0; NL:0; P:1; Type: call
-- Parameters:"\t"
--setSelectedText(.)

=========================================================================
org/gjt/sp/jedit/textarea/JEditTextArea.java at Rev: 114 in jedit
=========================================================================
-       else setSelectedText("\t");
+       else insertTab();

=========================================================================
org/gjt/sp/jedit/textarea/JEditTextArea.java at Rev: 86 in jedit
=========================================================================
-       else if (buffer.getBooleanProperty("noTabs")) {
-      int lineStart=getLineStartOffset(caretLine);
-      String line=getText(lineStart,caret - lineStart);
-      setSelectedText(createSoftTab(line,buffer.getTabSize()));
-      }
-       else setSelectedText("\t");
```

Figure 5-21. A BugMem IDE (Eclipse) integration screenshot. Based on the memories, BugMem warns that '*setSelectedText()*' is a potential bug, and shows bug fix examples from memories.

5.3.3 Bug and Fix Understanding

The memories of bug fixes are very useful for developers who are new to software projects. Core developers who know and remember all previous bugs and fixes may be able to avoid making the same mistakes again. For new developers, however, the memories of bug fixes are essential to guide their future development. When they do not know the right method or constant to use, automatically recovered memories of bug fixes can help correct mistakes and suggest correct examples.

5.4 Discussion

5.4.1 Threats to Validity

There are several major threats to the validity of the BugMem approach.

Systems examined might not be representative. We examined 5 systems, and it is still possible that we accidentally chose systems that have better (or worse) than average bug fix memory hit rates. Since we intentionally only chose systems that had some degree of linkage between change tracking systems and the text in the change log (so we could determine bug fix changes and hunks), we have a project selection bias. As our dataset increases the severity of this threat will diminish.

Systems are all open source. The systems examined all use an open source development methodology, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure, different personnel turnover patterns, and different development processes used in commercial development could lead to different memory hit rates.

All systems are written in Java. Extracting components from hunks and building memories requires a complete programming language parser. As a result, BugMem currently only supports the Java language. Other programming languages may have different bug patterns and memory hit rates.

Bug fix data is incomplete. Even though we selected projects that have change logs with good quality, we still are only able to extract a subset of the total number of bugs (typically only 40%-60% of those reported in the bug tracking system). The identified bug-fix data is not the oracle set. It may include false positives and false negatives. Incomplete bug fix data may increase (or decrease) false positives, and prevent the development of complete bug fix memories.

5.4.2 Limitations of BugMem

5.4.2.1 Limitation in the Initial Stage of a Project

The bug fix memories approach is applicable only when a project has been under development for awhile, and hence some bug fixes have been collected in the memories. In contrast, horizontal bug finding tools can work immediately from revision 1 of a newly started project. How many revisions must pass before BugMem achieves a reasonable hit rate? Figure 5-15 shows how hit rate varies over the life of a project. The results imply that waiting 200-900 revisions before using BugMem is a reasonable rule-of-thumb.

5.4.2.2 Limited Program Parsing and Analysis Employed

Due to performance considerations and in order to reduce implementation complexity, our approach only performs a one-pass scan of the source code files when parsing them and employs limited control flow or data dependence analysis to find bug fix patterns and discover buggy code. If we want to achieve better accuracy, more comprehensive program parsing and program analysis are preferred. For example, cross-file parsing can identify the types of variables and the return types of methods external to a class file; sophisticated program analysis [11, 38, 45, 58] and points-to analysis [6, 36, 56, 78] can identify more accurate dependence relationship between statements.

Our approach is based on comparing a source code file of two versions, so the bugs captured and fixes suggested are only file-by-file based. The cross-file relationships of bugs and fixes are not revealed. When considering cross-file relationships, such as inheritance, class usage, or method invocation to other classes, we may find more bug fix patterns.

5.4.2.3 Dependence on Change Log Quality and Revision Submission

The way we automatically identify bug fix revisions is based on keyword searching in change logs. If developers provide low-quality change logs or do not even provide change logs when submitting changes, it will impact the effectiveness of the BugMem approach, i.e. more false positive or false negatives will be introduced.

The way developers make and submit changes is another factor that affects the BugMem approach. If a developer submits bug fix changes and non-fix changes in the same revision, some bug fix changes will be wrongly regarded as non-fix changes or some non-fix changes will be wrongly treated as bug fix changes, which will impact the accuracy of the BugMem analysis.

5.4.3 Ways to Improve the BugMem Approach

5.4.3.1 Isolate Non-bug-prone Methods

We believe that some methods are unlikely to have bugs, e.g. *System.out.println()*, *length()* of a String object, getter/setter methods, etc. If these non-bug-prone methods are added to the memory database, they will cause false positives. A simple way to prevent these methods from going into the database is to manually create a list of such non-bug-prone methods and filter them out during the component extraction step. Another possible way is to use program metrics [32, 60, 71, 83] to identify non-bug-prone methods. For example, if a method only has a small number of lines of code [83], a small number of output or input variables [71, 83], low Cyclomatic complexity [60], or low program slicing complexity [71], we may deem this method as being non-bug-prone and exclude it from the memory database. In addition, we can use the bug cache algorithm [48] to identify bug-prone methods and eliminate those non-bug-prone methods from the bug memories.

5.4.3.2 Ranking the Matches

When BugMem is integrated into an IDE like Eclipse, user feedback on the results of memory hits can help BugMem produce more accurate results afterwards. A ranking value can be attached to each component in the memory. A positive feedback from a user will increase the ranking value for a component. A preset threshold can be used to prevent the components in the memories with low ranking values from being presented to the user for bug alerts.

We can also use some other possible factors to rank the matched components, such as reputation of the author of the code, rate of false positive hits of the component in the history, age in the database of the component, and the information value of the component.

5.4.3.3 Improve the Quality of Revision Submission

As discussed in Section 5.4.2.3, if a revision contains many kinds of changes including bug fix change and non-fix change, BugMem cannot differentiate bug fix changes from non-fix ones, and hence produces inaccurate results. For example, consider the following change log message attached to revision 33 of the JEdit project: “Fixing window docking, better HTML stylesheet, various other changes”.

We can see that changes in revision 33 fixed a bug in window docking, but also included “various other changes”. Since there is a keyword “fix” in the change log, BugMem regards the entire revision as bug fix revision, but this is not correct due to “various other changes” in the revision.

It is a good software engineering practice to do only one ‘thing’ or activity [41] in a revision. That is, if you have a bug to fix, make only the changes to fix the bug and then submit these changes; if you have a new feature to add, add the code for the new feature and submit the changes; but don’t make changes to fix a bug and add code for new features at the same time and submit two kinds of changes in one revision. Following this practice will help BugMem identify bug fix changes. A developer can further improve the BugMem approach by explicitly specifying in the change log message that a revision is a bug fix revision that can be recorded in the memory database to prevent similar bugs. That is, the user can write a keyword like “<Fix-to-memories>” in the change log message to inform BugMem to record the changes in this revision. Following such practices will likely improve the practical use of the BugMem approach.

6 Related Work

6.1 Software Change and Change Representation

There are many reasons why developers change source code. Swanson [82] and Pressman [75] classify software changes into four categories, correction, adaptation, enhancement, and prevention. Corrective changes are those performed to fix bugs in software. Adaptive changes are software modifications due to change in the external environment. Enhancement changes are needed when there are additional requirements for the functionality of the software. Preventive changes are performed when software reengineering is needed to prevent the deterioration of software. Barry et al. [7] elaborated on the classification of enhancement changes, adding six sub-categories including data handling, control flow, initialization, user interface, computation, module interface and initialization. From an empirical study, they found that most enhancement changes are in data handling and control flow logic. Chapin et al. [12] proposed an objective-based semi-hierarchical classification of maintenance activities. On the top level of the hierarchy are the four high-level categories of maintenance activities including the software, the documentation, the properties of the software, and customer-experienced functionality.

Mockus and Votta [63] merged enhancement changes with adaptive changes and classified changes into three major categories: adaptive, corrective and perfective. Besides these, they introduced one extra change category, inspection rework, which indicates changes performed following inspections. They also introduced an information retrieval approach to automatically classify software changes by analyzing the change log attached to the changes. Generally, they use the following strategy: if the change log contains keywords such as fix, problem, incorrect, correct, the change is a corrective change; if the change log contains the keywords, add, new, modify, or update, it indicates an adaptive change; keywords such as cleanup, unneeded, remove, and rework indicate perfective changes; and if the change log message contains any inspection keyword, the change is an inspection change. Our work used Mockus and Votta's algorithm to perform change classification. Since we only care about bug fix changes, we use keywords such as fix and bug to recognize bug fix changes.

To show the difference between a changed file and its original version, the GNU diff tool is usually used. GNU diff calculates the longest common subsequence (LCS) of two text files at the line level. The text lines that are not in the long common subsequence are the difference of the two file versions. GNU diff is favored in practical usage, since it is widely used, free, cheap in computation, and language neutral. Whereas, there are advanced approaches that compute and represent the syntax, structure, or semantic changes of source code files.

Computing document differences based on document syntax or structure has been explored by many research efforts. Yang [86] developed a syntactic comparison and merge tool based on parse trees for the C programming language. Different files are parsed to generate corresponding parse trees. Then, a tree matching algorithm runs to match nodes and locate differences. Finally, a pretty-printer traverses the trees and highlights the different code sections in the files. Cdiff [30] takes a similar approach for C++. Due to the limitation of the C++ Information Abstractor cdiff uses, it only handles comparison at the procedure level. There are also syntactic comparison approaches that use graph structures to represent the code. For example, Mens [62] uses labeled typed nested graphs and graph rewriting techniques to provide a formal foundation for software diff and merge.

The program behavior comparison problem has been explored somewhat. Horwitz and Reps [37] introduced the *TestIsomorphism* algorithm to compare two program slices based on the dependence graph representation of program slices. Yang et al. [88] developed a Sequence-Congruence algorithm that can find program components that have identical behaviors in one or more programs using static-single-assignment forms and program dependence graphs. Komondoor and Horwitz [52] presented an algorithm to detect clones in source code by using the program dependence graph and program slicing to find isomorphic program dependence subgraphs. Semantic Diff [43] detects the semantic difference of a function across two revisions by comparing their input-output behavior, i.e. the dependence relationship between the input variable (or constant) and output variable pairs in the function. Apiwattanapong et al. [2] introduced the CalcDiff algorithm that compares the behaviors of object-oriented programs. To compare program behaviors, their approach employs enhanced control flow graphs. We also explored the approach to detect program behavior changes across revisions. We introduce the PSE (program slice encoding) algorithm [72] that generates semantic fingerprints for C language functions by encoding program slices in each function to hash values. These hash values are used to identify function behavioral changes across revisions.

Though the semantic difference computed by program analysis techniques reveals deeper insights of program changes, these approaches are expensive in computation and do not scale well for large programs. In our research, we use GNU diff to represent changes across revisions, since the diff approach is straightforward and efficient in computing, and the line-based output of diff is easy to manipulate. To address the issue that the textual difference from the diff tool lacks syntactic or semantic information, we employ syntax parsing and analyses on the file versions examined, and provide syntax information in addition to textual difference.

6.2 Bug Taxonomy

The literature contains several bug classification taxonomies. IEEE presents a classification scheme of types of software abnormalities, including logic problem, Computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document quality problem, and enhancement [42]. Orthogonal Defect Classification (ODC) [13] is a scheme for capturing the semantics of software defects whose types are associated with different phases of the software life cycle, such as design, development, test and service. Emam and Wieczorek [23] evaluated ODC by showing that the classification scheme is in general repeatable.

Ostrand et al. [69] developed a fault categorization scheme by examining change report forms for an interactive special-purpose editor system. The change report forms contain several questions to reveal the causality of software errors. The questions on the change report include, why is this change being made, at what step of the software development was the problem first noticed, during which step was the problem created, how was the problem detected, why do you think the problem occurred, etc. From the case study, they categorized the faults into seven major categories including data definition, data handling, decision & processing, decision (alone), system, documentation and unknown. Their fault distribution results show that data definition and data handling categories contain the most defects.

Endres [24] explored the classification of program errors discovered during internal tests of the components of an operating system, and the possible causes of these errors. Endres defined three major groups of program errors: (1) errors in the understanding of the problem and in the choice of an algorithm to solve it; (2)

incomplete or incorrect implementation of a given algorithm; (3) spelling errors in message or comments, missing comments, or integration errors. In the analysis on the errors in an operation system, Endres found that almost half of the errors (46%) are in group one, i.e. errors in understanding the problems.

Perry et al. [73] surveyed software faults in a large real-time system through questionnaires. Design and code phase faults were classified into 22 types including language pitfalls, protocol, low-level logic, internal functionality, interface complexity, etc. The characteristics of these faults were analyzed for several factors, including fault categories, fault causality, and fault prevention. In their work, they found that the faults types, internal functionality, interface complexity, unexpected dependencies, low-level logic, and design/code complexity account for 60% of the faults; incomplete/omitted design, lack of knowledge, and ambiguous design are major causes of faults; and the major means of error prevention employed in the project include application walk-throughs, expert person/documentation, and guideline enforcement.

Leszak et al. [54] studied a networking product to explore the causality of defects. In their study, they investigated defect modification requests and classified the defects into three classes, implementation, interface, and external. Each class has a set of appropriate defects. For example, the following defect types belong to the implementation class: algorithm, functionality, performance or language pitfalls. They computed the distribution of defect types on the defects in the system studied, and found the algorithm defect type to be the most prevalent type.

How do bug fix patterns differ from these existing fault taxonomies? Bug fix patterns are very syntax-driven, while existing taxonomies tend to be either cause-driven (what caused this bug), and/or document-driven (what document is this cause located in). A major benefit of bug fix patterns is that they are automatically extractable, whereas existing taxonomies usually require human categorization of a bug into the taxonomy. Furthermore, since bug fix patterns are closely tied to the program text, the patterns are more concrete and less ambiguous than many existing fault categories that require human interpretation. This makes bug fix patterns far better suited for cross-project comparison of fault data.

6.3 Bug Finding Tools

Many bug finding tools such as Bandera [17], ESC/Java [26], PMD [16, 74], JLint [4, 5], and FindBugs [39] have been proposed and are in wide use [76]. Most of these tools use syntactic pattern matching, static analysis, model checking, or theorem proving.

Bandera [17] provides software checking capabilities for concurrent programs using finite-state verification techniques, such as model checking. Bandera automatically exacts safe, compact, finite-state models from Java source code, which can be used to verify the Java source code against requirements, such as a temporal logic formula.

ESC/Java [26] is a static checking system for Java programs based on verification-condition generation and automatic theorem-proving techniques. Assisted by user-added annotation of preconditions, postconditions, and loop invariants, ESC/Java verifies whether a program matches the design choices recorded in annotations. Without annotations, ESC/Java can also find potential runtime faults such as null pointer deference and array out-of-bounds.

JLint [4, 5] is a static Java program checker. JLint performs syntactic checks and dataflow analysis to find potential bugs in synchronization (such as race conditions, and deadlock), inheritance (such as mismatch of methods profiles and components shadowing), and data flow.

PMD [16, 74] performs syntactic checking on Java programs to identify potential bugs using pre-defined syntactic error patterns such as ‘empty if statement’, ‘misplaced null check’, or ‘no null check in the equal method’ [16]. These error patterns are recorded as rules, and users can extend PMD by adding their own error checking rules.

FindBugs [39] defines 50 bug patterns in Java programs. Each of the bug patterns has a corresponding detector that detects bugs of this pattern by performing syntactical matching or dataflow analysis in Java bytecode.

These bug finding tools are similar to BugMem in that they perform syntax matching or static analysis, find bugs, and then suggest correct code. They are good at detecting commonly known bugs, such as null dereferencing errors. However, they do not detect high-level project-specific bugs.

While prior bug finding tools use built-in and pre-defined bug patterns, BugMem learns project-specific bug patterns by analyzing an ongoing development history. Additionally, BugMem can suggest correct code in the project to repair detected buggy code.

6.4 Using Project History to Detect Bugs

The BugMem approach uses project histories to detect bugs and suggest fixes. Project histories are widely used to build project knowledge [18, 19], detect common bug patterns [57, 85], and find association rules among bugs [80].

Hipikat is a tool that recommends relevant software artifacts to developers based on project histories comprised of artifacts such as source code changes, mailing list messages, bug tracking entries, and written documentation [18, 19]. The Hipikat approach is similar to BugMem in that it builds up a repository of information from the project's history. However, we explicitly identify bad (bug) and good (fix) memories to detect potential bugs and suggest fixes. Hipikat tries to provide related references to developers rather than identify good or bad memories. Hipikat additionally requires keywords to search memories while BugMem analyzes source code and extracts components automatically.

Williams and Hollingsworth use project histories to improve existing bug finding tools [85]. When a function returns a value, using the value without checking it may be a bug. The problem in this approach is that there are too many false positives due to the generation of warnings about all source code that uses an unchecked return value. To remove these false positives, Williams and Hollingsworth use project histories to determine what kinds of function return values must be checked. For example, if the return value of the function '*foo()*' was always checked in the project history, but not checked in current source code, it is very suspicious.

Livshits and Zimmermann combined software repository mining and dynamic analysis to discover common method usage patterns that are likely to encounter violations in Java applications [57]. Their approach employs dynamic analysis and is more specific in finding violation patterns on method usage pairs. For example, *blockSignal()* and *unblockSignal()* should always be paired in the source code.

The approaches in HistoryAware [85] and DynaMine [57] are vertical bug finding techniques similar to ours, since they both analyze project-specific patterns. However, they only focus on a small set of bug patterns, such as the return value checking in HistoryAware [85] and the method usage pairs in DynaMine [57]. In contrast, BugMem uses all kinds of components to build memories and detect bugs, and the kinds of components keep growing along with the development process.

Song et al. find association rules among six bug types from project histories [80]. Using these association rules, they can predict future bugs. For example, suppose bug types *A* and *B* are often found together in the history. Then if we find only bug type *A* in the source code, we assume the code contains bug type *B* as well. BugMem uses components from bug hunks to detect bugs, and does not use any bug association rules. Using bug component association rules may increase hit rates; testing this idea remains future work.

Brun and Ernst extract properties from buggy code and feed it to machine learning algorithms to train a bug prediction model [10]. Their approach is similar in that they try to capture properties of buggy code and use it for future prediction. However, they use invariant information for their code properties, while we use syntactic information. Their approach does not provide any bug fix suggestions.

6.5 Identifying Buggy Areas

Identifying buggy code areas is quite useful for improving software quality, and many approaches have been proposed. Some approaches use software complexity metrics to identify buggy areas, assuming that complex software has more potential bugs [32, 33, 66, 68, 70, 71]. Other approaches leverage a project's bug history, change history, or code co-changes to identify buggy areas [8, 31, 64]. Prediction accuracy for these approaches range from 60-90%, but the areas predicted to be buggy are quite coarse, ranging from subsystems to modules, binaries, files, or functions. Even though BugMem has lower accuracy (hit rate), it precisely locates bugs at the line level and provides suggestions for fixes.

CP-Miner [55] is an approach that finds copy-paste code clone regions in source code and detects "forgot-to-change" bugs in them. The CP-Miner approach normalizes an entire statement and maps it to a hash code for duplication matching. Our BugMem approach also performs normalization for statements, but we extract normalized code segments in a statement based on the consideration that a statement can be very long, and only part of it has problems. CP-Miner works well for a statement with many identifiers, but not for those with few identifiers. For example, it maps the statement *out.println(abc)* and the statement *jButton.addActionListener(listener)* to the same hash code, but they are obviously different statements. CP-Miner only addresses the bugs caused by copy/paste and forgetting to rename an identifier. In contrast, BugMem is able to identify bugs in any changed source code region.

6.6 Using Code Examples to Assist in Development

Holmes and Murphy proposed an approach to extract structural components from example code and use them to assist coding when developers are working on similar code [34]. Mandelin et al. introduced a jungloid mining approach that automatically generates jungloid code fragments by mining library and example code to provide common API use examples [59]. The input to the jungloid mining is the input and the output types of APIs. The output of jungloid mining is examples of method call sequences extracted from sample client programs or synthesized from API method signatures. The jungloid mining approach focuses solely on method calls.

Source code search engines [51] are also widely used to find source code examples. BugMem is similar to source code example approaches, since we extract components from source code examples in the history (hunks). However, we identify both bad (from bug hunks) and good (from fix hunks) examples, using them to detect bugs. Prior code example approaches assume that all examples are good source code, but the existence of as-yet undiscovered bugs in all projects means that this is not true.

7 Future Work

Bug fix patterns were explored in this dissertation. We identified 27 bug fix patterns through a manual analysis of bug fixes in five open source Java projects. Due to the limited number of samples we have examined, there must be many bug fix patterns we have missed. Examining more bug fix changes and finding more bug fix patterns is a straightforward follow-on step. Currently, the bug fix patterns we defined are mostly syntax-driven ones. This characteristic of the bug fix patterns allows for automatic extraction, but with reduced ability to perform higher-level cause identification. For future work, we would like to explore higher-level bug fix patterns. Hopefully, we can find a balance between automatic extractability and semantic involvement of bug fix patterns.

We developed an extractor to automatically extract instances of bug fix patterns from bug fix changes. However, our approach only performs a one-pass scan of the source code files when parsing them and employs limited control flow and data dependence analysis to find bug fix patterns. The tradeoff of this design decision is the loss of analysis accuracy due to unknown variable types or missing data dependency between statements. Adding more comprehensive program parsing and program analysis in the extractor is planned for future work.

There is also much future work that can be performed on the BugMem approach. We notice that BugMem has a high false positive hit rate and there is still much room for reducing false positives. As discussed in Section 5.4.3.1, differentiating between bug-prone methods and non-bug-prone methods is a way to reduce false positives. The same approach can be applied to non-method-call components too. User feedback is another way to reduce false positives.

Our future work on BugMem also includes developing a full-fledged BugMem tool, so it can be used in a practical environment and can be used to further evaluate the BugMem approach. Feedback from a user study could improve the BugMem approach. Since BugMem complements the horizontal bug finding tools and vice versa, we will explore the possibility of integrating the bug patterns from the horizontal bug finding tools with the BugMem approach to improve its capability to find bugs.

The ideas in the BugMem approach can be used to produce a better source code search engine. Current source code search engines [14, 29, 51] are mostly keyword-based. But source code text has distinct characteristics from regular text. We believe that extracting these special characteristics, such as data types for variables, source code structure, and context for source code, and then using this characteristic information in source code search will improve the quality of search results. There are two ideas in the BugMem approach that can be used in source code search engines. First, the processing of the source code such as source code denoizing, data type normalization, in-statement component extraction, and context extraction will add more favorable information in addition to keywords to source code searching. Second, BugMem discriminates between good code and bad code. The ability to recognize good code and bad code is very important to a source code search engine, since it will reduce the possibility of retrieving and presenting buggy code to users. There are some other factors we can use to rank our confidence that a segment of code is good or bad, such as the reputation of the author of the code, bug rate of the code in the history, test reports for the code, code complexity, etc. We believe the ideas in BugMem can improve source code searching by helping search engines to search relevant and good code and assist people in learning from examples and previous development experience.

8 Conclusion

In this dissertation, we first explored the underlying patterns in the bug fixes mined from the change history of software projects. Through manual inspection of part of the bug fix change history of five open source projects, we defined 27 extractable bug fix patterns based on the syntax components and context of the code involved in the bug fix changes. We also developed an extractor tool that automatically detects bug fix pattern instances in a project's change history. The bug fix pattern extractor was used to characterize the bug fix patterns of seven Java open source projects, Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek. The results show that 45.7% to 63.6% of the bug fix changes are covered by the bug fix patterns we defined. In all the bug fix pattern instances, those related to *if* and method calls account for 44.6% to 60.3% of them. A correlation analysis on the extracted pattern instances on the seven projects shows that most of them have similar bug fix pattern distributions.

Analysis of If-conditional (IF-CC) sub-patterns found that these bug fix changes commonly increase the complexity of the conditional by adding operators, variables, or condition clauses.

We explored the application of using bug fix patterns to discover additional buggy code and fix code, which addresses the issue in using textual difference to identify bug and fix code. We introduced the context information attached to the discovered buggy code and fix code. The analysis on bug fix changes in seven open source projects shows that using bug fix patterns we can discover 5.0% to 17.4% more buggy code over that discovered by diff.

The second major goal in the dissertation was to leverage software evolution patterns to find duplicated bugs. We presented BugMem, a project-specific bug finding approach using memories of bug fixes. BugMem detects potential bugs and suggests corresponding fixes. We evaluated BugMem by computing bug fix memory hit rates. We found that 17.5% - 32.4% of bugs (half hit), and 7.5% - 13.0% of bug and fix pairs (full hit) appear repeatedly in the history. We also compared the bug sets identified by PMD and by BugMem, and found the two sets are nearly mutually exclusive. We conclude that prior bug finding tools and BugMem should be used together to maximize bug detection capability.

Source code repositories such as CVS and Subversion are typically used to store histories and make backups. In our view, a source code repository contains knowledge that can be used to discriminate between good and bad source code. So far, the knowledge available in source code repositories has not yet been fully leveraged. Our approach of computing memories of bug fixes provides a useful way to extract and deploy the knowledge latent in source code repositories. We harness this information to improve the quality of source code and provide detailed guidance to developers.

BIBLIOGRAPHY

[1] Apache, "Lucene Homepage," 2006, http://lucene.apache.org/.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," In Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04), Linz, Austria, 2004, pp. 2-13.

[3] ArgoUML, "ArgoUML Home Page," 2006, http://argouml.tigris.org/.

[4] C. Artho, "Finding Faults in Multi-threaded Programs." Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001

[5] C. Artho, "Jlint - Find Bugs in Java Programs," 2006, http://jlint.sourceforge.net/.

[6] D. C. Atkinson and W. G. Griswold, "Effective Whole-Program Analysis in the Presence of Pointers," In Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, 1998, pp. 46-55.

[7] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "Toward a Detailed Classification Scheme for Software Maintenance Activities," In Proceedings of the 5th Americas Conference on Information Systems, Milwaukee, WI, 1999, pp. 726-728.

[8] J. Bevan and E. James Whitehead, Jr., "Identification of Software Instabilities," In Proceedings of the 10th Working Conference on Reverse Engineering, Victoria, B.C., Canada, 2003, pp. 134-145.

[9] J. Bevan, E. James Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," In Proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, 2005, pp. 177-186.

[10] Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004, pp. 480-490.

[11] D. Callahan, "The Program Summary Graph and Flow-Sensitive Interprocedual Data Flow Analysis," In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1988, pp. 47-56.

[12] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance and Evolution*, vol. 12, no. 1, pp. 3-30, 2001.

[13] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal Defect Classification-A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943-956, 1992.

[14] Codase, "Codase - Source Code Search Engine," 2006, http://www.codase.com/.

[15] Columba, "Columba Home Page," 2006, http://www.columbamail.org/drupal/.

[16] T. Copeland, *PMD Applied*: Centennial Books, 2005.

[17] J. C. Corbett, M. B. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, Robby, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," In Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, 2000, pp. 439-448.

[18] D. Cubranic and G. C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003, pp. 408-418.

[19] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446-465, 2005.

[20] S. Dreilinger, "CVS Version Control for Web Site Projects," 1998, http://interactive.com/cvswebsites/.

[21] Eclipse, "Eclipse Home Page," 2006, http://www.eclipse.org/.

[22] Eclipse, "Eclipse Java Development Tools (JDT) Subproject Home Page," 2006, http://www.eclipse.org/jdt/.

[23] K. E. Emam and I. Wieczorek, "The Repeatability of Code Defect Classifications," In Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 322-332.

[24] A. Endres, "An Analysis of Errors and Their Causes in System Programs," In Proceedings of the International Conference on Reliable Software, Los Angeles, California, 1975, pp. 327-336.

[25] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," In Proceedings of 2003 Int'l Conference on Software Maintenance (ICSM'03), Amsterdam, the Netherlands, 2003, pp. 23-32.

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002), Berlin, Germany, 2002, pp. 234-245.

[27] GNU, "GNU Diffutils," 2006, http://www.gnu.org/software/diffutils/.

[28] M. Godfrey and Q. Tu, "Tracking Structural Evolution using Origin Analysis," In Proceedings of the International Workshop on Principles of Software Evolution, Orlando, Florida, 2002, pp. 117 - 119.

[29] gonzui, "gonzui: a source code search engine," 2006, http://gonzui.sourceforge.net/.

[30] J. E. Grass, "Cdiff: A Syntax Directed Diff for C++ Programs," In Proceedings of USENIX C++ Conference, Portland, OR, 1992, pp. 181-193.

[31] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.

[32] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, 2005.

[33] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 2005, pp. 263-272.

[34] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," In Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, 2005, pp. 117-125.

[35] S. Horwitz, "Identifying the Semantic and Textual Differences between Two Versions of a Program," In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, White Plains, New York, 1990, pp. 234 - 245.

[36] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence Analysis for Pointer Variables," In Proceedings of the ACM SIGPLAN 1989 Conference on

Programming Language Design and Implementation, Portland, Oregon, 1989, pp. 28-40.

[37] S. Horwitz and T. Reps, "Efficient Comparison of Program Slices," *Acta Informatica*, vol. 28, no. 9, pp. 713-732, 1991.

[38] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[39] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92-106, 2004.

[40] IBM, "Rational ClearCase Home Page," 2004, http://www-306.ibm.com/software/awdtools/clearcase/.

[41] IBM, "Software Configuration Management: A Clear Case for IBM Rational ClearCase and ClearQuest UCM," 2004, http://www.redbooks.ibm.com/redbooks/SG246399/wwhelp/wwhimpl/js/html/wwhelp.htm.

[42] IEEE, *IEEE Standard Classification for Software Anomalies*: IEEE Standard 1044-1993, 1993.

[43] D. Jackson and D. A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," In Proceedings of the International Conference on Software Maintenance, Victoria, BC, Canada, 1994, pp. 243-252.

[44] jEdit, "jEdit - Programmer's Text Editor," 2006, http://www.jedit.org/.

[45] J. Keables, K. Robertson, and A. v. Mayrhauser, "Data Flow Analysis and Its Application to Software Maintenance," In Proceedings of Conference on Software Maintenance, 1988, pp. 335-347.

[46] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Journal*, vol. 11, no. 1, pp. 19-37, 2003.

[47] T. M. Khoshgoftaar and E. B. Allen, "Predicting the Order of Fault-prone Modules in Legacy Software," In Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 344-353.

[48] S. Kim, "Adaptive Bug Prediction by Analyzing Project History." PhD Dissertation, Computer Science, University of California, Santa Cruz, Santa Cruz, CA, 2006.

[49] S. Kim, K. Pan, and E. James Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," In Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005), Pittsburgh, Pennsylvania, 2005, pp. 143-152.

[50] S. Kim, T. Zimmermann, K. Pan, and E. James Whitehead, Jr., "Automatic Identification of Bug-Introducing Changes," In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan, 2006.

[51] Koders, "Koders - Source Code Search Engine," 2006, http://www.koders.com/.

[52] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," In Proceedings of the Eighth International Static Analysis Symposium (SAS), Paris, France, 2001, pp. 40-56.

[53] R. Kumar, S. Rai, and J. L. Trahan, "Neural-network Techniques For Software-quality Evaluation," In Proceedings of 1998 Reliability and Maintainability Symposium, Anaheim, California, 1998, pp. 155-161.

[54] M. Leszak, D. E. Perry, and D. Stoll, "A Case Study in Root Cause Defect Analysis," In Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 428-437.

[55] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176-192, 2006.

[56] D. Liang and M. J. Harrold, "Efficient Points-to Analysis for Whole-program Analysis," In Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, 1999, pp. 199-215.

[57] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," In Proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, 2005, pp. 296-305.

[58] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," In Proceedings of the Conference on Software Maintenance, 1993, pp. 282-291.

[59] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, 2005, pp. 48-61.

[60] T. J. McCabe and A. H. Watson, "Software Complexity," *Crosstalk*, vol. 7, no. 12, pp. 5-9, 1994.

[61] MegaMek, "MegaMek Homepage," 2006, http://megamek.sourceforge.net.

[62] T. Mens, "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution," In Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance, Kerkrade, The Netherlands, 1999, pp. 127-143.

[63] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," In Proceedings of International Conference on Software Maintenance (ICSM 2000), San Jose, California, 2000, pp. 120-130.

[64] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169-180, 2005.

[65] A. Moore, "Cross-Validation," 2005, http://www.autonlab.org/tutorials/overfit.html.

[66] J. Munson and T. Khoshgoftaar, "The Detection of Fault-prone Programs," *IEEE Transaction on Software Engineering*, vol. 18, no. 5, pp. 423-433, 1992.

[67] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886-894, 1996.

[68] T. Ostrand, E. Weyuker, and R. Bell, "Where the Bugs Are," In Proceedings of the ACM/International Symposium on Software Testing and Analysis (ISSTA2004), 2004, pp. 86-96.

[69] T. J. Ostrand and E. J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289-300, 1984.

[70] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340-355, 2005.

[71] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Bug Classification Using Program Slicing Metrics," In Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, Philadelphia, PA, 2006.

[72] K. Pan, E. J. Whitehead, Jr., and G. Ge, "Textual and Behavioral Views of Function Changes," In Proceedings of the 3rd International Workshop on

Traceability in Emerging Forms of Software Engineering (TEFSE'05), Long Beach, CA, 2005, pp. 8-13.

[73] D. E. Perry and C. S. Stieg, "Software Faults in Evolving a Large, Real-Time System: a Case Study," In Proceedings of the Fourth European Software Engineering Conference, Garmisch, Germany, 1993, pp. 48-67.

[74] PMD, "PMD Home Page," 2006, http://pmd.sourceforge.net/.

[75] R. S. Pressman, *Software Engineering - A Practitioner's Approach*, 5th ed: McGraw-Hill Higher Education, 2001.

[76] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," In Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, Bretagne, France, 2004, pp. 245-256.

[77] Scarab, "Scarab Home Page," 2006, http://scarab.tigris.org/.

[78] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-To Analysis," In Proceedings of the 24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1997, pp. 1-14.

[79] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," In Proceedings of the International Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, 2005, pp. 24-28.

[80] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69-82, 2006.

[81] Subversion, "Subversion Project Home Page," 2005, http://subversion.tigris.org/.

[82] E. B. Swanson, "The Dimensions of Maintenance," In Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, 1976, pp. 492-497.

[83] S. Toolworks, "Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and FORTRAN," 2006, http://www.scitools.com/.

[84] Y. Wang, D. J. DeWitt, and J.-Y. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents," In Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, 2003, pp. 519-530.

[85] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466-480, 2005.

[86] W. Yang, "How to Merge Program Texts," *Journal of Systems and Software*, vol. 27, no. 2, 1994.

[87] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software Practice & Experience*, vol. 21, no. 7, pp. 739-755, 1991.

[88] W. Yang, S. Horwitz, and T. Reps, "Detecting Program Components with Equivalent Behaviors," Department of Computer Sciences, University of Wisconsin, Madison Technical Report 840, 1989.