2013

Improving software quality with programming patterns

Tung Thanh Nguyen
Iowa State University

Improving software quality with programming patterns

by

Tung Thanh Nguyen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Tien N. Nguyen, Major Professor

Suraj C. Kothari

Manimaran Govindarasu

Akhilesh Tyagi

Samik Basu

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to dedicate this work to my wife, Hong Minh Trinh, who has waited so long for it to be done. My six-year PhD study is a long journey full of ups and downs and she has always been beside me to share all the joys and sorrows. To be with me, she has made countless sacrifices, from enduring a long-time, long-distance relationship, to delaying her personal career to get marriage, living in a strange country very far from her family and friends, and being on her own through all the long nights while I was working late at the lab or traveling on multi-day trips for conferences and interviews. Most importantly, she has given me the greatest gift of all: our baby William, a cute and bright boy, whose birth and growth have taken from her a huge amount of health and effort. Hong and William are the main motivation and inspiration for me to overcome all the hardship of my career, from doing high quality research work to competing for very selective career opportunities in academia.

I would like to give my PhD as a special gift to my family, and especially, to my father. As a poor farmer, my father has spent all of his youth fighting for the independence of my country and the rest of his life fighting for the prosperity of my family. Due to the war and the poverty, he could not get a good formal education, thus, he has taken every opportunity and spent everything he has for me to get the best education I could have. My father knows nothing about computer, but twenty years ago, he has hired the first computer science teacher available in my village to teach me the first lessons on computer. He has spent a lot of money for me to attend one of the best high schools in the country with a specialized training curriculum in computer science. He has brought me my first personal computer, which worths an annual income amount of a typical worker at the

time, so I could have more time and tool support to study computer science courses and improving programming skills. Without his great vision and support, I might end up as a poor farmer rather than a computer scientist with a lot of opportunities ahead.

With my deepest appreciation, I would like to thank my parents-in-law, who have spent all of their time and effort to take care of my wife and my baby, so I could dedicate my final PhD year working on this dissertation and looking for jobs. I would like to thank my grand-parents, aunts and uncles, brothers and sisters, and everyone in my extended family who have supported us all the time with great love and sympathy.

I would also like to express my special thanks to those who have helped me with various aspects of my PhD study. First and foremost, I would like to thank my advisor, Dr. Tien N. Nguyen, for his guidance and support for my research and career development. His advices have inspired me for pursuing the most exciting research problems and his words of encouragement have kindled my desire for establishing a career in academia. Without his support, I might not have been succeeded in my PhD study as well as in applying for a faculty position.

I truly appreciate my friends and team-mates: Hoan Anh Nguyen, Anh Tuan Nguyen, Nam Hoai Pham, Hung Viet Nguyen, Jafar Al-Kofahi, and Ahmed Tamrawi, who have worked very closely beside me, contributed significantly to my work, and helped me with many other problems in my personal life. I would also like to thank my collaborators: Drs. Peter Santhanam, Evelyn Duesterwald, and Tim Klinger at IBM Watson Research Center, Dr. Miryung Kim at University of Texas at Austin, Dr. David Lo at Singapore Management University, and Dr. Tu Minh Phuong at Vietnam Posts and Telecommunications Institute of Technology, for their contributions to my research.

Finally, I would like to thank my committee members for their time evaluating my work, advising future research, and recommending me for job opportunities. I would like to thank the Vietnam Education Foundation and the US National Science Foundation for their financial support for my PhD study.

ABSTRACT

Software systems and services are increasingly important, involving and improving the work and lives of billions people. However, software development is still human-intensive and error-prone. Established studies report that software failures cost the global economy $312 billion annually and software vendors often spend 50–75% of the total development cost for finding and fixing bugs, i.e. subtle programming errors that cause software failures.

People rarely develop software from scratch, but frequently reuse existing software artifacts. In this dissertation, we focus on programming patterns, i.e. frequently occurring code resulted from reuse, and explore their potential for improving software quality. Specially, we develop techniques for recovering programming patterns and using them to find, fix, and prevent bugs more effectively.

This dissertation has two main contributions. One is Graph-based Object Usage Model (GROUM), a graph-based representation of source code. A GROUM abstracts a fragment of code as a graph representing its object usages. In a GROUM, nodes correspond to the function calls and control structures while edges capture control and data relationships between them. Based on GROUM, we developed a graph mining technique that could recover programming patterns of API usage and use them for detecting bugs. GROUM is also used to find similar bugs and recommend similar bug fixes.

The other main contribution of this dissertation is SLAMC, a Statistical Semantic LAnguage Model for Source Code. SLAMC represents code as sequences of code elements of different roles, e.g. data types, variables, or functions and annotate those elements with sememes, a text-based annotation of their semantic information. SLAMC models

the regularities over the sememe sequences code-based factors like local code context, global concerns, and pair-wise associations, thus, implicitly captures programming idioms and patterns as sequences with high probabilities. Based on SLAMC, we developed a technique for recommending most likely next code sequences, which could improve programming productivity and might reduce the odds of programming errors.

Empirical evaluation shows that our approaches can detect meaningful programming patterns and anomalies that might cause bugs or maintenance issues, thus could improve software quality. In addition, our models have been successfully used for several other problems, from library adaptation, code migration, to bug fix generation. They also have several other potential applications, which we will explore in the future work.

CHAPTER 1. INTRODUCTION

"Software is eating the world", wrote Marc Andreessen, a famous software engineer and investor. From billion-user search engines to personal mobile games, software systems and services are becoming essential components of our society. In the physical world, software controls vehicles, manufacturing machines, transportation, and utility networks. In the cyberspace, software directs digital communications, mediates business transactions, and optimizes information flows. Software is an integrated part of our everyday life, helping us at work, entertaining us at home, managing our financial and medical records, and connecting us to other people. Even ordinary devices like phones or TVs are becoming "smart" because of having software running inside. Software systems are also critical to national security, as they monitor terrorist activities, defend infrastructures again malicious intruders, and control militarily facilities, from satellites, missiles, to unmanned drones. With such important economic, militarily, and social impacts, software is involving and improving the work and lives of billions people.

1.1 Software Quality Problem

Software is mainly written and maintained by human. A software system might be developed by a team of thousands engineers, is programmed in millions lines of code, and is used by billions users. Due to this high level of complexity, programming errors are unavoidable, causing unexpected behaviors and failures. Often called as "**bugs**", these programming errors and software failures occur frequently with several negative affects.

Chrome is an interesting example. It is currently the most popular Web browser with hundreds of millions of users world-wide. It is developed by Google, one of the best software companies in the world. However, after five years on the market (2008–2013), its bug tracking system, where users and developers reporting bugs and other problems with the product, has recorded nearly 250,000 issues[1]. Among those issues, more than 50,000 are still open, i.e. have not been fixed yet. Microsoft Windows, the most popular desktop operating system, is another example for software bugs. Although its bug statistics are not disclosed, we are all familiar with its error messages and the infamous *"blue-screen-of-death"*, which often causes us to lose all unsaved work.

Not only annoying, software bugs are also costly. A recent report entitled *"The Big Cost of Software Bugs"* by Bloomberg[2], lists ten high-profile software bugs which often costs hundreds of millions to billions dollars. The failed voyage of Mars Climate Orbiter, a spacecraft built by NASA's Jet Propulsion Laboratory is one among them. Due to a bug its control software, this spacecraft approached Mars in a wrong angle and was destroyed, causing a loss in total of more than $655 million. According to the report from NASA[3], the cause of this failure is a silly mistake when *"one team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation."*. Y2K is another silly bug caused when software engineers store year information with just two, rather than four, digits. However, as estimated by the research firm IDC, *"$296.7 billion was spent worldwide from 1995 to 2001 to mitigate the damage, with outages costing $21 billion"*.

At a broader scope, in 2002, the National Institute of Standards and Technology (NIST), a U.S. government agency, reports that software bugs cost the US economy around $60 billion each year [18]. In the same year, another study from IBM Research estimates that software vendors also spend huge amount of money, often 50–75% of the

[1]https://code.google.com/p/chromium/issues/list?can=1 - Accessed at 12:00 on 12/02/2013

[2]http://www.bloomberg.com/slideshow/2012-08-03/the-big-cost-of-software-bugs.html - Accessed at 12:05 on 12/02/2013

[3]http://mars.jpl.nasa.gov/msp98/news/mco990930.html - Accessed at 12:10 on 12/02/2013

total software development cost, for finding and fixing bugs [21]. The situation does not improve over the last ten years, as a recent research from University of Cambridge reports that programmers need to spending 50% of their working time for debugging. The research also estimates the cost of software bugs to the global economy to be of $312 billion annually [6].

Not just causing the loss of money, software bugs sometimes cause the loss of human lives and other fatalities. The Therac-25 incident is a representative example [39]. Therac-25 is a computer-controlled medical device used in radiation therapy. Due to an error in its control software, it has overdosed a number of patients with doses up to 100 times of the intended ones. At least three of those patients have died due to such overdosed radiation. The aforementioned report from Bloomberg discusses another incident in which a U.S. Patriot missile defense system failed to detect the attack of an incoming Scud missile, causing the death of 28 American soldiers.

1.2 Reuse Practice and Reuse-related Bugs

Due to such huge and negative consequences of software bugs, in my PhD study, I dedicate my research effort on developing methods and techniques that help software engineers find and fix bugs more effectively and more desirably, write code that is less error-prone in the first place. This would improve the quality and reliability of their software products, increase their programming productivity and reduce the development cost, making software more useful and accessible to people.

There are different approaches to achieve that goal. My research approach is based on the observation that in software development, developers often do the same of similar programming tasks over and over again. While repeating tasks, they might end up repeating errors and mistakes. To do the same or similar tasks, developers would reuse existing software artifacts rather than rewrite from the scratch. Therefore, I focuses my

research to understand how reuse could cause bugs and how to use the knowledge of reused artifacts to improve such situations.

The conventional wisdom and established studies suggest that reuse is an encouraged and widely-used practice in software development. Developers have different ways to reuse. They might directly copy-and-paste a piece of code, or sometimes, duplicate a whole codebase. Developers could reuse API (Application Programming Interface) elements, e.g. functions and data structures, from existing libraries and frameworks. In many cases, the reused artifacts are of higher levels of abstraction like design patterns or algorithms. In principle, reuse provides quick and effectively tested solutions for common problems or recurring programming tasks, thus, reducing the development time and cost. However, in practice, reuse could lead to problems when people reuse the wrong things or reuse the wrong ways, as shown in the following examples.

Example 1. Figure 1.1 shows an example of copy-and-paste code. It contains two methods setColspan and setRowspan of class Auxheader in ZK, a Java framework for building enterprise web and mobile application[4]. As seen, these two methods are highly similar and actually provide similar functionality: adjusting column span or row span of an Auxheader object. Therefore, the developer just wrote one method, then copied and made slight modifications to create the other. Due to this convenience, similar code fragments resulted from copy-and-paste practice like these ones, often called *code clones* in the research community, are pretty popular. Established studies estimate that 20-30% of source code of typical software systems are code clones [33, 40].

However, the convenience backfires when people reuse buggy code, which exactly happens in this case. The original function has a bug that it adjusts the span but does not update the user interface, making no changes visible to users. Thus, by reusing code, the developer duplicates this bug and then needs to fix in both locations. Figure 1.2 shows the fixed code (in boxes) applied to those two methods. We call these *"recurring*

[4]http://www.zkoss.org - Accessed at 12:16 on 12/02/2013

```
public void setColspan(int colspan) throws WrongValueException {
 if (colspan <= 0) throw new WrongValueException("Positive only");
 if (_colspan != colspan) {
  _colspan = colspan;
  smartUpdate("colspan", Integer.toString(_colspan));
}
```

```
public void setRowspan(int rowspan) throws WrongValueException {
 if (rowspan <= 0) throw new WrongValueException("Positive only");
 if (_rowspan != rowspan) {
  _rowspan = rowspan;
  smartUpdate("rowspan", Integer.toString(_rowspan));
}
```

Figure 1.1 Code clones in class Auxheader.java of ZK

bug fixes" and found that they might account for up to 40% of all bug fixing changes in a software system [54].

In this case, it is fortunate that the bug duplicated due to copy-and-pasting buggy code is spotted and fixed at the same time with the original bug, leaving no consequence. However, in practice, the developer who fixes the bug might not be aware about the copied code (e.g. he makes the copies a long time ago and totally forgets, or worse, he is not the one who makes the copies and his code is copied to another system unknown to him). In this case, the bug is not fixed completely and still has potential to cause bad affects. Our empirical study has found at least 228 reported software vulnerabilities caused by bugs recurring on code-and-paste and duplicate code [59].

Example 2. Figure 1.3 shows an example of bugs that recur because developers make the same mistake when reusing APIs. EVP_VerifyFinal is an API function of OpenSSL[5], an open source toolkit implementing SSL and TLS protocols. This function verifies a signature against corresponding public key(s) and returns *three* values: 1 if the signature is correct; 0 if it is incorrect; and -1 if the verification process fails. However, the return value of -1 is overlooked by developers of NTP[6], the Network Time Protocol project. As

[5]http://www.openssl.org - Accessed at 12:17 on 12/02/2013
[6]http://www.ntp.org - Accessed at 12:17 on 12/02/2013

```
public void setColspan(int colspan) throws WrongValueException {
 if (colspan <= 0) throw new WrongValueException("Positive only");
 if (_colspan != colspan) {
  _colspan = colspan;
  final Execution exec=Executions.getCurrent(); if (exec!=null && exec.isExplorer()) invalidate();
  smartUpdate("colspan", Integer.toString(_colspan));
}
```

```
public void setRowspan(int rowspan) throws WrongValueException {
 if (rowspan <= 0) throw new WrongValueException("Positive only");
 if (_rowspan != rowspan) {
  _rowspan = rowspan;
  final Execution exec=Executions.getCurrent(); if (exec!=null && exec.isExplorer()) invalidate();
  smartUpdate("rowspan", Integer.toString(_rowspan));
}
```

Figure 1.2 Recurring bug fixes at revision v5089 of ZK

a. Security bug in NTP 4.2.5

```
static int crypto_verify() {
...
  EVP_VerifyInit (&ctx, peer->digest);
  EVP_VerifyUpdate (&ctx, (u_char *)&ep->tstamp, vallen + 12);
  if (!EVP_VerifyFinal(&ctx, (u_char *)&ep->pkt[i], siglen, pkey))
    return (XEVNT_SIG);
...
```

b. Security bug in Gale 0.99

```
...
  EVP_VerifyInit(&context, EVP_md5());
  EVP_VerifyUpdate (&context, data.p, data.l);
  for (i = 0; is_valid && i < key_count; ++i) {
    if (!EVP_VerifyFinal(&context,...)) {
      crypto_i_error();
      is_valid = 0;
      goto cleanup;
  }
  cleanup: EVP_PKEY_free(key);
...
```

Figure 1.3 Recurring security vulnerabilities

seen in Figure 1.3a, the statement "if (!EVP_VerifyFinal..." only considers the signature to be unverified when the return value of EVP_VerifyFinal is 0 (XEVNT_SIG means Signature Not Verified). This is a vulnerable target for hackers to exploit. They could just create a mal-formed signature, causing EVP_VerifyFinal to return -1, and thus, bypassing this signature verification process of NTP.

It is interesting that developers of Gale[7], an instant messaging software system, also make the same mistake. As seen in Figure 1.3b, they also use the same flawed statement "if (!EVP_VerifyFinal...", and thus, creating the same vulnerability. And it is not the only one. In our empirical study reported in [59], we have found six other instances of this reuse error. In total, we found at least 50 reported security vulnerabilities involving to programming bugs that recur due to the same errors in reusing different API functions, some of them also belong to OpenSSL, such as DSA_verify and ECDSA_verify.

Example 3. Figure 1.4 shows an example of a different kind of bugs related to API reuse. An software library often has several functions (in object-oriented programming, such functions could be implemented via classes and methods). To reuse this library to perform a task, developers need to use such functions following certain rules. For example, in Figure 1.4a, to change attribute(s) of a node in an XML document using API in Fluid, one first needs to get access the node using its name using method getNodeWithName. If the node does not exist, he needs to create it using method createNode before making any change with method setAttr.

This is a correct and preferred way to change nodes' attributes. Therefore, such code appear frequently and thus, is called an *API usage pattern* in Fluid. However, people do not always follow the rules. Figure 1.4b shows a case when the code does not check for the existence of a node before making change(s). This leads to a Null Pointer Exception error when the accessed node does not exist.

The examples and published studies suggest that reused code is prevalent and reuse-

[7]http://www.gale.org - Accessed at 12:18 on 12/02/2013

a. Usage pattern to change a node's attributes

```
IRNode locNode = doc.getNodeWithName(node, "location");
if (locNode == null) // check for existence
  locNode = doc.createNode("location"); // and create node before modify
doc.setAttr(locNode, "x",...);
```

b. Usage error: change a node's attributes without checking its existence

```
IRNode locNode = doc.getNodeWithName(node, "location");
doc.setAttr(locNode, "x",...); // Null Pointer Exception when node "location" not exist
```

Figure 1.4 API usage pattern and error in Fluid

related bugs are also popular [54, 59, 32, 41, 33]. Those bugs occurred when people made the same mistakes or did not follow the common rules when reusing source code and APIs. Therefore, one way to help people improve code quality is to find the instances of both good code and bad code. Known good code could be used as a guide for people to write other good new code, or as a reference to detect existing bad code. Similarly, known bad code could be used to detect the same or similar written bad code, or as examples to avoid when writing new code. In this dissertation, we focus on detecting programming patterns, which could be considered as a kind of good code. We also investigate in detecting bad code, in the form of recurring bugs and vulnerabilities.

1.3 Programming Patterns

Programming patterns are the code frequently written in codebases of software systems [56, 73, 78, 26]. They could be simple programming idioms, like a general-purposed for loop for iterating over elements of an array or a try catch construct for handling an IOException. They could also be more complex and project-specific, especially usage patterns of API (Application Programming Interface) libraries and frameworks. The API usage pattern in Fluid illustrated Figure 1.4a is an example of project-specific programming patterns. Figure 1.5 shows another example of a more common programming pattern. This pattern performs a common task in Columba, a mail client written in Java:

```
StringBuffer strbuf = new StringBuffer();
BufferedReader in = new BufferedReader(new FileReader(file));
String str;
while ((str = in.readLine()) != null)
  strbuf.append(str + "\n");
if (strbuf.length() > 0)
  saveMessage(strbuf.toString(), ...);
in.close();
```

Figure 1.5 An example of programming patterns

reading a text file and storing the content as an email message. As seen in the figure, many Java APIs (e.g. classes StringBuffer and BufferedReader, methods StringBuffer.append, StringBuffer.toString, and BufferedReader.readLine) are used in this pattern.

Due to their popularity in source code, programming patterns could be considered as *"the wisdom of the crowds"* [20], representing the correct ways to use APIs or the most efficient/convenient/preferred ways to program a common task. Unusual code that slightly deviate from patterns might potentially be programming mistakes. For example, in Figure 1.5, if the check "!= null" (in the while loop) is missing, the code will have an infinite loop error: at the end of file, the method readLine will always return null and the loop will not stop. This suggests that we could use programming patterns to detect bugs in existing code or guide programmers to prevent bugs when writing new code by following those patterns. Programming patterns are also useful for other software development tasks, automated generation of test cases [71], adaptation of API usages when API evolves [53], or automated patching of source code [35], etc.

However, programming patterns are often not readily available for programmers, especially project-specific patterns. For example, the documentation of the API libraries and frameworks might be outdated or even unavailable. The original designers and developers of the systems might retire or move to other projects, to management, or to other companies. Sometimes, a user of the APIs might invent new ways of reuse, which are unknown to original developers of the APIs, or to other users.

However, since programming patterns would appear frequently in written code and related software artifacts (e.g. execution traces, API documentation), we could analyze those artifacts to (automatically) infer the embedding patterns, a task called *pattern mining*. Due to the importance of programming patterns, pattern mining is an active and fruitful research area. Researchers have proposed many models and techniques [73, 78, 69, 41, 71] to represent source code and detect patterns.

This dissertation makes two novel contributions to this rich literature. One is *Graph-based Object Usage Model* (GROUM), a graph-based representation of source code, and the accompanying techniques to recover API usage patterns from source code and use them for detecting bugs. GROUM is also used to find recurring bugs and recommend corresponding bug fixes. The other main contribution is SLAMC, a *Statistical Semantic LAnguage Model for Source Code*. SLAMC represents code as sequences of code elements and captures programming idioms and patterns as sequences with high probabilities of appearing. We have used SLAMC as the core of a code suggestion engine, which can recommend most likely code sequences for editing code, thus, improving programming productivity and reducing the odds of programming errors. Each contribution will be introduced in more details in the following sections.

1.3.1 GROUM: Graph-based object usage model

In the beginning part my PhD study, my research focuses on programming patterns with graph-based approaches. The result is a graph-based representation of object usage called *Graph-based Object Usage Model* (GROUM). A GROUM abstracts a given portion of code by its object usages and represents those usages as a graph. In this graph, the nodes correspond to the method invocations and control structures while the edges captures control and data relationships between them. Figure 1.6 shows the GROUM representing the programming pattern in Figure 1.5. As seen in the figure, two method calls BufferedReader.readLine and StringBuffer.append are represented by two nodes with the

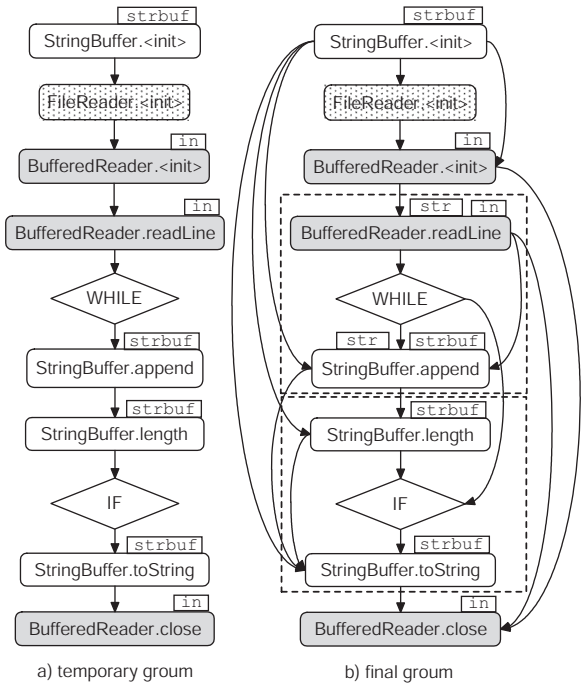

Figure 1.6 Graph-based object usage model

corresponding labels. The edge between them represents their control and data relationship: BufferedReader.readLine is called before StringBuffer.append and they share data via variable str. Similarly, there is an edge from BufferedReader.readLine to BufferedReader.close since they are method calls on the same object in and BufferedReader.readLine is called before BufferedReader.close. The nodes labeled WHILE and IF represent two control structures: the while and if statements in this pattern.

Based on GROUM, we have developed several techniques to support software developers to improve software quality and productivity. Among them, GrouMiner is a tool that that could automatically infer programming patterns from a given codebase and check for rare violations of those pattern that could potentially cause bugs. Our evaluation suggests that GrouMiner is effective. For example, it can analyze a system of a half million lines of code in around one hour, and several other smaller systems in couples of minutes. GrouMiner can detect many high quality programming patterns, both common and project-specific, which could be easily reused. Using those patterns, GrouMiner

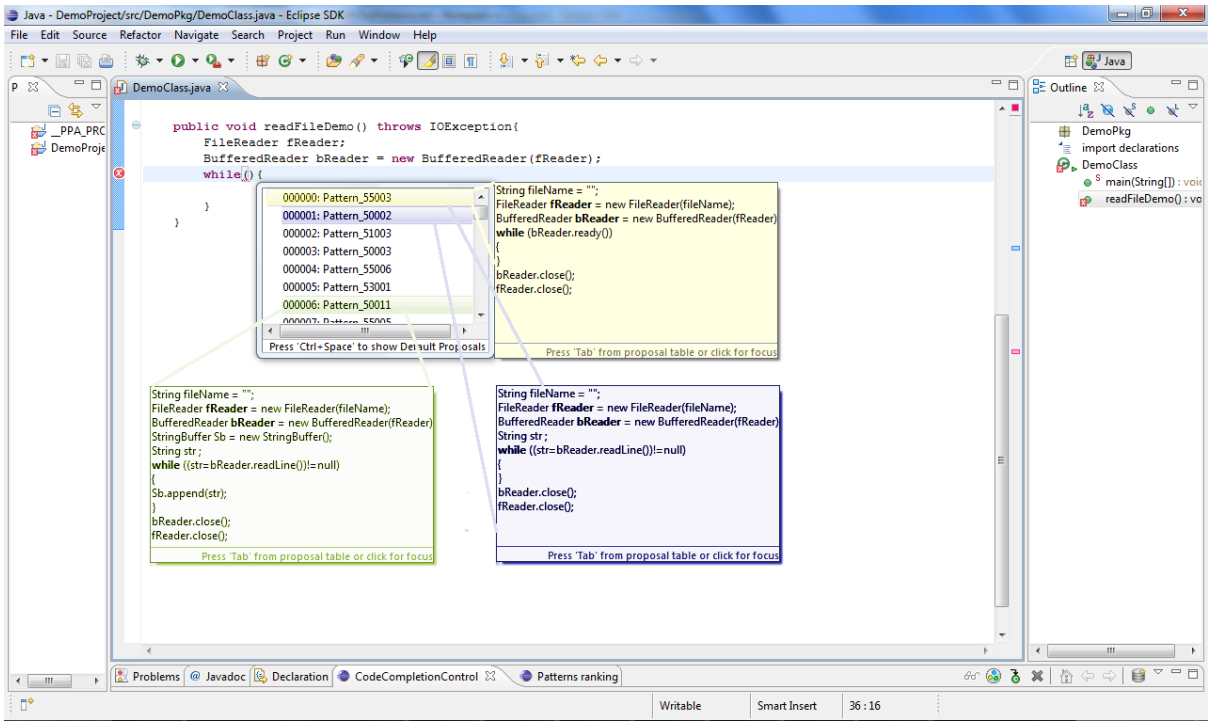

Figure 1.7 Grapacc: A pattern-based code completion tool [50]

has detected several bugs and problematic usages in those systems which have not been found previously by their programmers. The usage error presented in Example 2 and Figure 1.4 is one of the detected errors. Section 2 will present GrouMiner in full details.

Programming patterns could help programmers write code more efficiently and less error-prone. My collaborators and I have developed Grapacc [50], a pattern-based code completion tool, using GROUM as the internal representation of code and patterns. Armed with an extensible knowledge base of patterns, Grapacc could analyze the current editing code, determine the missing parts, and recommend the most suitable patterns. When a pattern is chosen, Grapacc will automatically fill it in. Since the whole pattern is filled in at the time, programmers are less likely to miss some steps and cause usage

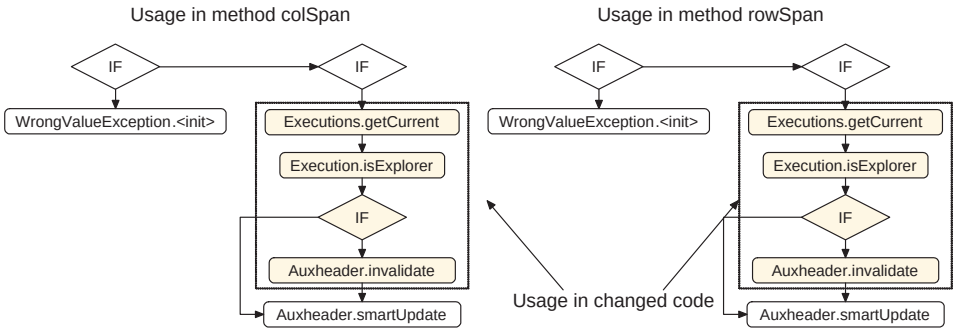

Figure 1.8 Usage models of code with recurring bugs and fixes

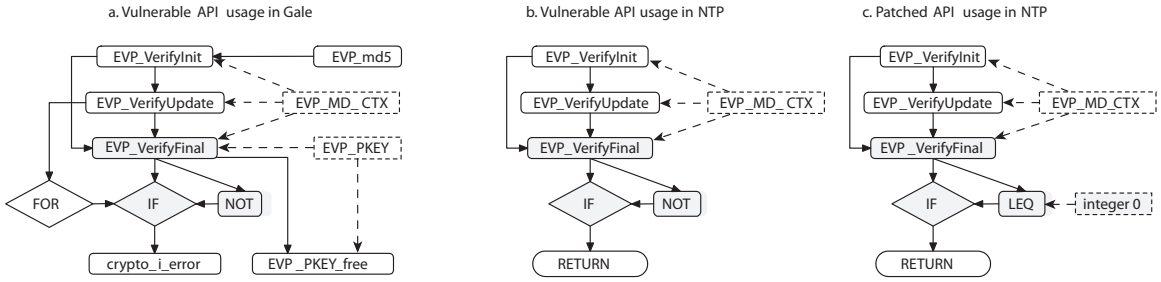

Figure 1.9 Usage-based signatures of recurring vulnerabilities and patches

errors. They also write code faster since most of the code has been filled by Grapacc. Figure 1.7 demonstrates the running of Grapacc in a usage scenario. Technical details of Grapacc are presented in [50].

GROUM is also applicable for detecting recurring bugs and fixes. Figure 1.8 shows the GROUM representation of the code with recurring bugs and fixes presented in Example 1, Figure 1.1, and Figure 1.2. As seen, the two methods have identical GROUMs, both before and after the bug fixes. Our empirical study on five systems estimates that there might be up to 40% of recurring bug fixes occurring on code units with identical or similar GROUMs. Based on this study, we have developed FixWizard, a tool that could scan a given software system for those similar code units and monitor their changes. Then, when a unit has a bug and gets a fix, FixWizard will alert the potential bugs recurring in the units similar to it and recommend similar fixes for them. Full details about FixWizard will be presented in Section 3.

My collaborators and I also adapted GROUM to detect recurring security vulnera-

bilities. Our tool, SecureSync, has a knowledge base of known/reported security bugs in which each bug is represented with a GROUM-based signature. SecureSync uses those signatures to scan other software systems for any API usages that are similar to those signatures and alert them as potential recurring bugs. Figure 1.9b illustrates the signature of the security bug in NTP presented in Example 2 and Figure 1.3 and the recurring bug in Gale. As seen, the signature of the bug in NTP matches exactly to the API usage in Gale (Figure 1.9a). Thus, using the bug signature in NTP, SecureSync is able to detect the recurring bug in Gale. Full details of SecureSync could be referred in [59].

1.3.2 SLAMC: Statistical semantic language model for source code

Since GROUM abstracts source code by object usages, GROUM-based techniques works very well for API usage patterns, which often involve several objects and method calls. However, there are programming idioms involving other code elements, like a general-purposed loop for (int i = 0; i < n; i++) or a check for nullity if (x != null)). Unlike API usage patterns, those idioms are often shorter and more localized, i.e. do not expand to wide ranges in source code and might not involve API usages.

Thus, in the second part of the thesis, we developed another approach to capture and utilize those kinds of patterns. The core of this approach is the novel model called *Statistical Semantic Languge Model for Code* (SLAMC). Unlike GROUM, this model captures all meaningful code elements (e.g. function calls, data types, variables, operators, etc), not just focusing on object usages. Unlike statistical models for natural languages (e.g. n-gram models or topic models), SLAMC is specially designed for source code and uses program semantic information (e.g. data types, scope and dependency) and several code-based factors like the local code context, the global technical concerns, and the pair-wise associations of code elements in the modeling process.

Source code is represented in SLAMC as sequences of code tokens annotated with their semantic information, e.g. data types and roles such as variables, fields, or methods

in the program. Such semantic annotation of code tokens are called *sememes*. For example, the statement "int l = s.length" is represented as a sequence of sememes "TYPE[Integer] VAR[Integer] OP[assign] VAR[String] CALL[String,length]". In this sequence, VAR[String] annotates the semantic information of token s as a String variable and CALL[String,length] annotates length as an invocation of method String.length.

As a statistical language model, the core of SLAMC consists conditional probabilities $P(c|p)$ specifying how likely a code token with sememe c will occur next to a code sequence with corresponding sememe sequence p. These conditional probabilities are modeled using on several factors including local code contexts, topics, and pairwise associations of code elements, and are estimated from existing code in the training process. Then, those conditional probabilities could be used to estimate the occurring probability of any given code sequence. Programming patterns are implicitly captured by SLAMC as the sequences that have high occurring probabilities.

Using SLAMC, we have developed a code recommendation and completion engine. Once trained on a codebase, SLAMC could estimate how likely a code token will occur after a given code sequence. For example, for the code sequence "int l = s.", the code tokens such as length and indexOf are more likely to appear next than ones like lastIndexOf or substring. By searching token-by-token and checking the relevancy of the search paths to the suggestion context, our engine predicts and suggests the most likely code sequences, which then are chosen by the user and filled in on request. The empirical evaluation shows that SLAMC outperforms lexical model for source code with an absolute improvement in accuracy from 10–25%.

1.4 Related Publications and Dissertation Outline

1.4.1 Related publications

This dissertation consists the main results of three papers, which are joint work of me, my advisor, and other members in my research group.

1. Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M Al-Kofahi, Tien N Nguyen. *Graph-based mining of multiple object usage patterns.* In Proceedings of the 2009 ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 383–392. ACM, 2009.

2. Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, Tien N. Nguyen. *Recurring bug fixes in object-oriented programs.* In Proceedings of the 2010 ACM/IEEE International Conference on Software Engineering, pages 315–324. ACM, 2010.

3. Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, Tien N. Nguyen. *A statistical semantic language model for source code.* In Proceedings of the 2013 ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 532–542. ACM, 2013.

I also mention in this dissertation the results from two other papers, which use GROUM as the internal representation to perform the detection of recurring software vulnerabilities and code completion.

1. Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Tien N. Nguyen.*Detection of recurring software vulnerabilities.* In Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering, pages 447–456. ACM, 2010.

2. Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, Tien N Nguyen. *Graph-based pattern-oriented, context-sensitive source code completion.* In Proceedings of the 2012 ACM/IEEE International Conference on Software Engineering, pages 69–79. IEEE, 2012.

1.4.2 Dissertation outline

The remaining of this dissertation is organized as the following. In Chapter 2, we present GROUM and its application in pattern mining and bug detection. In Chapter 3, we report a study in recurring bug fixes and present another application of GROUM for detecting and recommending those fixes. In Chapter 4, we present SLAMC and its application in code recommendation. In Chapter 5, we discuss the related work of our studies. Finally, Chapter 6 discusses other potential applications of GROUM and SLAMC and concludes my dissertation.

CHAPTER 2. GRAPH-BASED PATTERN MINING

As introduced in Chapter 1, in this chapter, we present Graph-based Object Usage Model (GROUM), a graph-based representation of source code via object usages. We then discuss how we could extract such object usage models from source code, how to detect programming patterns from the extracted models, and how to use those patterns to detect related errors.

2.1 Concept and Formulation

In this section, we will present Graph-based Object Usage Model (GROUM) in details. Since GROUM is originally designed for the purpose of mining API usage patterns, we first start with a few examples of API usage patterns as motivating examples for the design of GROUM. Then, we define the concepts of GROUM and the algorithm to extract GROUMs from source code.

2.1.1 Examples of API usage patterns and errors

2.1.1.1 Example 1. Common API usage pattern

Columba[1] is an email client written in Java. As an email client, it saves email messages as text files, and thus, frequently needs to read a text file and store its content as an email message back in memory. Figure 2.1 shows a code snippet extracted from Columba for that task. As seen, the code uses several Java APIs: classes StringBuffer, BufferedReader,

[1]http://freecode.com/projects/columba - Accessed at 12:19 on 12/02/2013

```
StringBuffer strbuf = new StringBuffer();                                    1
BufferedReader in = new BufferedReader(new FileReader(file));                2
String str;                                                                  3
while ((str = in.readLine()) != null)                                        4
  strbuf.append(str + "\n");                                                 5
if (strbuf.length() > 0)                                                     6
  saveMessage(strbuf.toString(), ...);                                       7
in.close();                                                                  8
```

Figure 2.1 Reading a text file using Java API

and FileReader along with their methods such as StringBuffer.append, StringBuffer.toString, and BufferedReader.readLine.

The flow of usage is as the following. First, strbuf, an StringBuffer object, is created (line 1). Then a FileReader is created for the given text file, and used as the input to create the BufferedReader object in (line 2). Then, in is used in a while loop to read each line of the file to a String variable str via its method readLine (line 4). If no more line is available in the file, readLine returns a null object and the reading loop stops. Otherwise, the read line (stored in variable str) is added to strbuf using its method append, with an additional new line character (line 5). After reading, if the content of strbuf is not empty (i.e. by checking whether its length is larger than zero in line 6), it is ready to be output via method toString (line 7). Finally, the BufferedReader object is closed (line 8).

As seen, the code uses five objects: strbuf, in, str, file, and an unnamed FileReader. It uses an object either by calling the object's methods (e.g. in.readLine or strbuf.append) or by specifying them as the input or output of a method call (e.g. str is used as the output of in.readLine and as the input of strbuf.append). There are some certain rules among such usages. For example, in needs to be created before reading. And in.readLine is called before strbuf.append. We call these as *temporal usage orders*.

This code snippet presents a convenient way to use Java API for a common programming task: reading the content of a text file and storing it as a string. Since this task is popular in Columba, the code appears frequently Columba's source code. Therefore,

```
IRNode locNode = doc.getNodeWithName(node,"location");                    1
if (locNode == null)                                                      2
  locNode = doc.createNode("location");                                   3
doc.setAttr(locNode, "x", loc.width+"");                                  4
```

Figure 2.2 Updating node attribute(s) in Fluid

we consider it as a *usage pattern* of Java API. Knowing this pattern would help novice Java programmers to learn how to use Java API to perform that task. Since this is a common programming task, it is even better if the pattern is incorporated in the code completion functionality of the code editors, thus the programmers can code this task and similar ones faster and less error-prone.

2.1.1.2 Example 2. Project-specific API usage pattern

Fluid[2] is a framework for program analysis. With strong support of software evolution, Fluid allows program analysis to be performed incrementally. Therefore, it frequently needs to update objects in its internal representation. Figure 2.2 shows a code snippet in Fluid to change an attribute of an IRNode object named locNode in a SCUmlDocument object named doc. First, locNode is accessed via its label "location" in the SCUmlDocument as by calling method doc.getNodeWithName (line 1). Then, if such a node does not exist (line 2), it will be created by calling method doc.createNode in line 3 before getting its attribute changed by calling method doc.setAttr in line 4.

Similar to Example 1, this code snippet also uses several objects (e.g. doc, loc, node, locNode). The objects could be used by method calls (e.g. doc.getNodeWithName(...) or doc.setAttr(...), field accesses (e.g. loc.width), or as input/output of other method calls (e.g. node is used as input of doc.getNodeWithName(...) and locNode is used as its output). There are also some certain rules. For example, we need to check whether locNode is not null, and create a new node with the given label if needed, before calling doc.setAttr.

Since updating objects is a frequent task in Fluid, code similar to this snippet also

[2]http://www.fluid.cs.cmu.edu - Accessed at 12:20 on 12/02/2013

```
IRNode locNode = doc.getNodeWithName(node, "location");
doc.setAttr(locNode, "x",...); // Null Pointer Exception when node "location" not exist
```

Figure 2.3 Usage error in Fluid

appears prevalent in Fluid. The specific node label ("location" in this case) or attribute ("x" in this case) might be different, but the method calls, field accesses, and their orders are the same. Therefore, we also consider this code snippet as an API usage pattern in Fluid. It is different from the pattern in Example 1 in the extent that it is a *project-specific* patterns, i.e. involving the internal APIs of the project, rather than common APIs like the pattern in Example 1.

Detecting project-specific programming patterns is useful and even more necessary than detecting common programming patterns. First, project-specific patterns are often known to only some team members in the project. In addition, due to the busy schedule and high speed of development, they are often lack of documentation and are changed frequently. Therefore, other members, especially newly joined developers, have to learn those patterns by looking through written code. This is a very inefficient, confusing, and error-prone process. A developer might overlook the code examples and do not properly use newly introduced classes, leading to errors. Moreover, specific rules of the usages, e.g. temporal orders method calls, cannot be checked at compile time. As a consequence, errors could not be caught until testing and even go unnoticed for a long time.

2.1.1.3 Example 3. API usage error

Figure 2.3 shows an example when developers in Fluid do not use the APIs correctly as described above. The code does not check for the existence of a node before making change to it attribute. This leads to a Null Pointer Exception error when the accessed node does not exist. This error has not been detected for a long time, until being exposed in our experiment. If the developer checked the code against the corresponding usage pattern, the error could have been detected earlier.

2.1.1.4 Discussion

In a software system, developers often reuse internal and/or external APIs to perform many programming tasks. Since object-oriented programming is currently the mainstream paradigm in software development, API usages often describe via objects and their interactions such as method calls (including the calls of constructors for creating objects), field accesses, and reference manipulations (e.g. passing object references as input, or assigning object references as output for method calls and/or field accesses). Some API usages also involve programming constructs like `while` loops or `if` statements. Since field accesses could be replaced by the calls to accessor (i.e. getter and setter) methods, we consider method calls as the main way for object interactions and call method calls of an object as its actions.

While developing a software system, developers often need to write code to perform the same or similar tasks (e.g. reading files or updating object attributes) again and again. Using APIs to program those tasks, developers end up creating many code snippets that are the same or similar to each other in the codebase. Those code snippets are called API usage patterns and are useful for learning about the APIs as well as for checking errors in existing code related to API usages.

As seen in the examples, an API usage pattern often involve several objects and object actions (i.e. method calls and field accesses). There might be some certain rules among them, for example, the temporal orders between method calls or the input/output relations between objects and method calls. The temporal order is not always exhibited in the textual order in source code. For example, the creation of the `FileReader` object occurs before that of `in` while the corresponding constructor call appears after in the source code. It is not the order in execution traces either, where `strbuf.append` could be executed before or after `in.readLine`. Therefore, we consider an action a to be *used before* another action b if a is *always generated before* b in the corresponding executable code.

These observations suggest that we could describe object usages with object actions

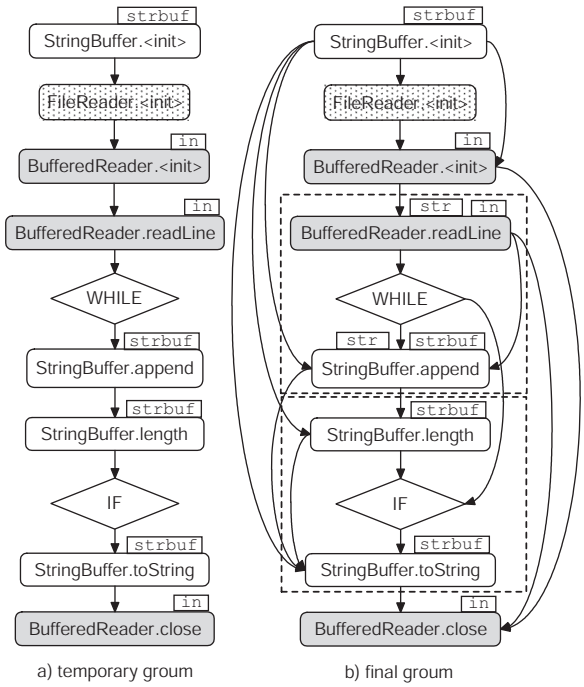

Figure 2.4 Graph-based object usage model

and their relationships. In the next section, we will present GROUM, a graph-based representation for object usages. In a GROUM, object actions are represented as nodes and their relationships (e.g. temporal orders) are represented as edges. Involving programming constructs like while loops or if statements are also represented as nodes. Then, we could extract object usages in a given codebase as a collection of GROUMs and detect usage patterns as their frequent sub-graphs.

2.1.2 Defining GROUM

This section describes Graph-based Object Usage Model (GROUM). A GROUM is a labeled, directed acyclic graph representing the usage of one or several objects in a given code snippet. Figure 2.4b shows the GROUM that represents the object usage illustrated in Example 1 (Figure 2.1). Let us explain the components of GROUM.

2.1.2.1 Action node

In a GROUM, an *action node* represents an object action (e.g. a method/constructor call or a field access) and has a label of C.m, in which C is the class/type name of the object and m is the name of the method/field. For example, the action node labeled BufferedReader.readLine in Figure 2.4b represents the method call in.readLine in line 4 of Figure 2.1. In the context where the class name is clear, we will use just the method name to identify the action node. As a convention, we use <init> as constructors' name.

We use methods' names for nodes' labels, instead of their signatures, because in practice, determining methods signatures is more expensive and complex, especially, with usages involving to type casting. More importantly, methods' names describe the usage of the objects, i.e. the corresponding actions, more generally. For example, class BufferedReader has two constructors, with and without the parameter for buffer size. However, the invocations of those two constructors could be considered to have the same meaning, because they are all used to initialize BufferedReader objects. In a usage scenario, a method could be called several times. That means, a GROUM could contain several action nodes with the same labels, although they represent different object actions.

Since an object action might involve data (e.g. input/output) relationships with other objects (e.g. the result of method call in.readLine is stored to str), we annotate an *action node* with all variables having such relationships. For example, action node BufferedReader.readLine is annotated with str and in (the object of that action). Action node StringBuffer.append is also annotated with str since the corresponding method call strbuf.append(...) uses str as its input. We use the following rules to determine variables involving an action node:

1. In a method call o.m(...) or a field accesso.f, o is an involving variable.

2. In an assignment C x = new C(...), x = o.m(...), or x = o.f, the assigned variable x is an involved variable.

3. In a field assignment o.f = E, all variables involving in the evaluation of E are considered as involved variables.

4. In a method call o.m(E1, E2,...), all variables involving in the evaluation of the arguments E1, E2,... are considered as involved variables.

2.1.2.2 Edges

In a GROUM, the edges represent the (temporal) usage orders. An edge from an action node a to an action node b indicates that a is used (e.g. called) before b. For example, in Figure 2.4b, two nodes labeled StringBuffer.<init> and StringBuffer.append represent the object instantiation and the invocation of method append of the StringBuffer object strbuf, respectively. The edge from <init> to append shows the usage order, i.e. ¡init¿ is called before append.

Because we determine usage order based on the order of code generation, if a is used before b, there is an edge from a to b and no edge from b to a. Thus, the edges in a GROUM are directed and a GROUM is a directed, acrylic graph (DAG). In addition, we only connect two nodes when they have *data relationship*. In our implementation, data relationship is determined by sharing variables. For example, in Figure 2.4b, there is an edge from Buffered.readLine to StringBuffer.append because the former is called before the latter, and they share data via variable str. Of course, action nodes for the same object are always considered as having data relationship.

2.1.2.3 Control node

We use *control nodes* to represent how objects are used within the control flow structures such as *conditions*, *branches*, or *loop* statements in a GROUM. To conform to the use of edges for representing temporal orders, such control nodes are placed at the branching points (i.e. where the program selects an execution flow), rather than at the

starting points of the corresponding statements. Thus, the edges between control nodes and action nodes also represent the usage orders.

For example, in Figure 2.4b, the control node labeled WHILE represents the while statement in the code in Figure 2.1, and the edge from the node BufferedReader.readLine to WHILE indicates that the call in.readLine(...) is used (i.e. generated) before the branching point of that while loop. The edge from node WHILE to node IF indicates the while loop is used before the if statement.

To represent the scope of a control structure (e.g. the calls of readLine and append are within the while loop), the corresponding control node has an attribute recording all action and control nodes within that control structure. In Figure 2.4b, such scope information is illustrated as the dashed rectangles. Then, the involved variables of a control node are all involved variables of nodes in its scope. It should be noted that a GROUM has no backward edge for loop statements since it is a DAG. However, without backward edges, scope information is still sufficient to show that the actions in a loop could be invoked repeatedly.

2.1.2.4 Formal definition

Combining aforementioned design decisions, we formally define Graph-based Object Usage Model as the following:

Definition 1 *A Graph-based Object Usage Model (GROUM), representing usage of one or several objects, is a DAG such that:*

1. Each node is an action node or a control node. An action node represents a call of a constructor or a method, or an access to a field of one object. Label of an action node is C.m with C is its class name and m is the method (or field) name. A control node represents the branching point of a control structure. Label of a control node is the name of its corresponding control structure.

2. Each edge represents the temporal usage order and data relationship between two nodes. An edge from node a to node b indicates that a is used before b, i.e. a is generated before b in the executable code, and a and b share data. Edges have no label.

GROUM has several advantages compared to existing representations for object usages. For example, GROUM is better than a set of method calls [41] because it could model the relationship between function/method calls (e.g. temporal orders), while a set can't. GROUM is better than a sequence of method calls [1], because a sequence specifies a total order over its elements, while in object usage, pair of method calls might have no order. GROUM is better than a collection of ordered pairs of method calls [73], since as a graph, it can represent the relationship of more than two nodes. In addition, in a collection of ordered pairs, the same method name will indicate the same method call. However, in a GROUM, two nodes might have the same labels, but still have different relations (represented via their edges) to other nodes.

Compared to Program Dependence Graph (PDG) and Control Flow Graph (CFG), two graph-based abstract models of source code, GROUM is specialized toward object usages and patterns. For example, GROUM does not have nodes for representation of literals, primitive variables, and arithmetic operators. Therefore, GROUM is more compact, thus, speeding the pattern mining process.

2.1.3 Extracting GROUM from source code

To extract the GROUM representing object usages/interactions from a portion of code of interest, one could extract usages for individual usage of each object and connect them based on usage orders and data relationships. However, to increase the efficiency, our extraction algorithm does this directly. It extracts GROUM from a portion of code of interest in the following steps:

1. Parse the code into an AST,

2. Extract all possible action and control nodes with their partial usage orders from the AST into a temporary GROUM, and

3. Identify data relationship and total usage orders between the nodes to build the final GROUM for the usage of all objects in the code portion.

We use Eclipse JDT to perform step 1. Two remaining steps are discussed in details in the following sections.

2.1.3.1 Extracting temporary GROUM

In this step, a temporary GROUM is extracted from the AST for each method. The extraction is processed bottom-up, building-up the GROUM of each structure from the GROUMs of its sub-structures. For a simple structure such as a single method invocation or a field access, a GROUM with only one action node is created. For more complex structures such as expressions or statements, the GROUM is merged using two operations: sequential merge (denoted by $\Rightarrow$) and parallel merge (denoted by $\vee$). The GROUM of a programming structure having neither action nor control node is empty.

The merge operations are defined as follows. Let X and Y be two GROUMs. $X \vee Y$ is a GROUM that contains all nodes and edges of X and Y and there is no edge between any nodes of X and Y. $X \Rightarrow Y$ is also a GROUM containing all nodes and edges of X and Y. However, there will be an edge from each sink node (i.e. node having no outgoing edge) of X to each source node (i.e. node having no incoming edge) of Y. Those edges represent the temporal usage order, i.e. all nodes of X are used before all nodes of Y. It could be checked that those two operations are associated; and parallel merge $\vee$ is symmetric but sequential merge $\Rightarrow$ is not.

Sequential merge is used where the code has an explicit generation order such as between statements within a block. Parallel merge is used where there is no explicit generation order such as between the branches of an `if-else` or a `switch` statement. With

Table 2.1 Composition rules of usage models

Code structure	Code template	Usage model
method invocation	o.m()	C.m
field access	o.f	C.f
parameters	o.m(X,Y,Z,...)	(X∨Y∨Z∨...)⇒ C.m
cascading call	X.m()	X⇒C.m
expression	X∘Y	X∨Y
if statement	if (X) Y; else Z;	X⇒IF⇒(Y∨Z)
switch statement	switch (X) case Y, case Z, ... ;	X⇒SWITCH⇒(Y∨Z∨...)
while statement	while (X) Y;	X⇒WHILE⇒Y
do while statement	do X while (Y);	X⇒DO⇒Y
for statement	for (X;Y;Z) W;	X⇒Y⇒FOR⇒W⇒Z
block	{X;Y;Z;...}	X⇒Y⇒Z ⇒...
try statement	try X catch Y	X∨Y

the use of parallel merge, a resulting GROUM is not affected by the writing order of some structures. For example, two syntactically different expressions $X + Y$ and $Y + X$ will have an identical GROUM, i.e. are considered as equivalent in usages.

Table 2.1 shows the composition rules of GROUM for different programming structures. Symbols such as X, Y, Z, and W denote the structures (in column Code template) and their corresponding GROUMs (in colume Usage model). Other symbols o, m, f, and C denote the object, method, field, and class names, respectively.

We explain those rules as the following. Two rules for field accesses and method calls with no parameters and are obvious. For a method call with parameters, the parameters need to be evaluated before calling the method. However, the order of evaluation between parameters are not explicitly, therefore, the GROUMs of parameters are merged parallelly, and then their combined GROUM is merge sequentially to the GROUM of the method call. Similarly, for a cascading call $X.m()$, we need to evaluate X before calling m, therefore, the GROUM resulted from the analysis of of X is merged sequentially to that for the call of m. For an infix expression, we do not have the explicit evaluation order of its components, thus, we merge the corresponding GROUMs parallelly.

GROUM composition rules for control structures like while, if, or for are defined

based on the order of evaluation and code generation of those structures. For example, for an if statement, we need to evaluate its conditional expression before executing each of two branches. However, two branches do not have explicit orders (they will never be evaluated in the same execution). Therefore, their GROUMs are merged parallelly, and the resulted GROUM is merge sequentially from that of the conditional expression. Because statements in a block are executed in order, the GROUM of the block is merged sequentially from GROUM of those statements. Since in a try catch construct, the catch part could be triggered at any time while the try part is executed, we merge their GROUMs parallelly, rather than sequentially.

2.1.3.2 Building final GROUM

To build the final usage model we first determine data relationships between all the nodes in the extracted temporary usage model. For each node (including both action and control nodes), a list of involved variables is collected using the rule in Section 2.1.3 and stored as its attributes. Then, any two nodes that share at least a common variable in their lists are considered to have a data relationship. Finally, if two nodes a and b have a data relationship and their is a path from a to b in the temporary GROUM, we will make an edge from a to b.

Our data analysis is only intra-procedural and explicit because we focus on the point of view of individual methods. (This individual method approach was shown to be scalable and to get comprehensive results [73].) To make a GROUM capture better the semantics of object usages, one could use inter-procedural analysis techniques to determine more complete data dependencies. Since those techniques are expensive, in our current implementation, we use a heuristic. That is, to increase the chance of connecting usages of objects having implicit data dependencies, each action node of an object will be connected to the *nearest* (downward) action node of any other object. For example, two nodes StringBuffer.<init> and BufferedReader.<init> in Figure 2.4b are connected using this

heuristic. This idea is based on the belief that the (implicitly) related objects tend to be used in near locations in code. Thus, these edges connect different parts of a method's usage model where each part represents the usage of a different object.

This step also helps discriminating the usages of different objects of the same class with the same method call. In this case, their action nodes have the same labels, but the involved variables might be different, thus, have different edges (usage orders and data dependencies). For example, assume that a scenario has two opened files: the first is for reading and the second for writing. If reading and writing involve a shared variable, the series of calls for two File objects would be connected as in a single usage. Otherwise, they would be identified as two separated usages of File objects.

2.2 Mining Usage Patterns

In this section, we will discuss how we could recover API usage patterns from source code. Intuitively, a usage is considered as a pattern if it frequently *"occur"* in a codebase (might contain source code of one or several projects). Using GROUM, usages in source code and patterns will be represented as graphs. We define the important concepts for pattern mining as following.

2.2.1 Formulation

First, we consider two object usages as equivalent if they involve the same object actions and relationships, i.e. their GROUM representation are identical. Since a GROUM is a labeled graph, use the concept of *label-isomorphic* to determine whether two GROUMs are identical.

Definition 2 *Two GROUMs* $G = (V, E, L)$ *and* $G' = (V', E', L')$ *are* ***label-isomorphic*** *if there exists an one-to-one mapping* f *between their nodes that preserves the edges and labels. That is:*

1. For each node $v \in V$, there exists one and only one $v' \in V'$ such that $v' = f(v)$ and $L'(v') = L(v)$.

2. For any pair of nodes (u, v), if $(u, v) \in E$ then $(f(u), f(v)) \in E'$. If $(u, v) \notin E$ then $(f(u), f(v)) \notin E'$.

In a method, developers might perform several tasks, thus, a usage pattern often is just a part of the code. Thus, the corresponding GROUM of the pattern might involve just some action and control nodes, which form a sub-graph, of the GROUM of that method. In other words, a usage could be considered to *"occur"* in a method if its GROUM representation matches a part of the method's GROUM.

Definition 3 *If a GROUM P is label-isomorphic to an induced subgraph Q of another GROUM G, P is considered to occur in G and Q is called an* ***occurrence*** *of P.*

Figure 2.6 shows an example. A usage pattern of size 3 is used in four methods, i.e. the GROUM representing the pattern "occurs" in, i.e. is label-isomorphic to an induce sub-graph of four corresponding GROUMs of the methods. In the last two methods, it occurs twice. However, in GROUM 4, we could only consider the pattern is used once since the two corresponding sub-graphs are overlapped. Thus, we define the number of occurrences of a GROUM as the following.

Definition 4 *The* ***number of occurrences*** *of a GROUM P in a GROUM G, denoted by count(P, G), is the maximal number of non-overlapping occurrences of P in G.*

We could extract object usages in a codebase into a collection of GROUMs. In our current implementation, we use only intra-procedural analysis to extract object usages (see Section 2.1.2), thus, we extract a GROUM for each method in a codebase (methods without object usages will have empty GROUMs, and thus, are discarded). Then, we can compute the number of occurrences of a GROUM P in the entire dataset as the total number of occurrences of P in all GROUMs in the dataset and consider ones with sufficient occurrences as usage patterns.

Definition 5 *A usage, represented as a GROUM P, is considered a pattern in a codebase, represented by a usage dataset, i.e. a collection of GROUMs $D = \{G_1, G_2, ..., G_n\}$, if the* ***total number of occurrences*** *P in D, denoted by $count(P, D) = \sum_{G \in D} count(P, G)$, exceeds a chosen threshold σ.*

We also call the total number of occurrences of a GROUM in a dataset as its frequency. Based on this formulation, finding usage patterns in a codebase becomes a frequent subgraph mining problem. There have been many algorithms developed for mining frequent subgraphs on a graph dataset (i.e. multi-settings) or on a single graph. However, they are not applicable for this mining problem because (1) the existing mining algorithms for multi-settings count only one occurrence in each graph (i.e. the frequency of a candidate pattern is the number of graphs it occurs, which is different from our problem); and 2) mining algorithms on a *single* graph setting are developed for edge-oriented subgraphs, i.e. a subgraph is defined as a set of edges that form a weakly connected component. They are only efficient on *sparse* graphs while our patterns are the induced subgraphs of *dense* graphs [63]. In the next section, we will present GrouMiner, a novel mining algorithm we specifically designed for our usage pattern mining problem.

2.2.2 Algorithm design strategies

2.2.2.1 Challenges

The brute-force for finding patterns is to generate all possible usages (each as a GROUM), compute their frequencies in the usage dataset, and output ones with sufficient frequencies. However, that strategy cannot work well in practice due to following reasons. First, it is impossible to generate all possible usages. A possible usage could be a GROUM with any action, control nodes, and control/data relationships. A software system might have thousands of classes and tens of thousands of methods. Thus, there are a huge pool of choices for even an action node, let alone the choices for a combination of nodes and

their relationships to form a GROUM.

Focusing only on the usages existed in the codebase is also challenging. In practice, we have encountered datasets with ten thousands graphs and some graphs with several hundreds nodes. An existing usage could be any sub-graph of those graphs and a graph has an exponentially high number of sub-graphs (a graph of n nodes has 2^n sub-graphs).

Even when a pattern candidate P is given, counting the number of occurrences of P in a GROUM G is hard. Checking whether P is isomorphic to a sub-graph of G is alone a difficult problem: it is sub-graph isomorphism, an NP-Complete, problem. Finding the maximal number of non-overlapping occurrences of P is also challenging, since it is an instance of the maximal independent set, another NP-Complete problem.

Based on these observation, we have designed GrouMiner with the following key design strategies: i) incremental generation of candidates, ii) signature-based, approximated graph isomorphism, and iii) approximated occurrence counting.

2.2.2.2 Strategy 1. Incremental generation of pattern candidates

The first design strategy aims to reduce the number of pattern candidates. Rather than generating candidates as all possible usages, we focus only on ones that have high potential to be patterns. This strategy is based on the observation that large patterns contain smaller ones. Thus, we use detected patterns of size k (i.e. having k nodes) to generate the candidates of patterns of size $k+1$. When detect a pattern of size k, we keep all its occurrences. Then, any occurrence of each detected pattern of size k will be extended to generate a new graph of size $k+1$ by adding a new node from the enclosing GROUM, thus, the generated one will always be a sub-graph of that GROUM. This reduces the need of checking sub-graph isomorphism. The generated graphs of size $k+1$ are grouped into isomorphic groups, each of which represents a candidate pattern. The frequency of each candidate is evaluated and if it is larger than a threshold, the candidate is considered as a pattern and is used to recursively discover larger ones.

2.2.2.3 Strategy 2. Signature-based, approximated graph isomorphism

The first design strategy reduces the number of pattern candidates and removes the need for checking sub-graph isomorphism. However, we now need to group the generated candidates into isomorphic groups. Currently, exact-matched graph isomorphism is highly expensive for dense graphs. To the best of our knowledge, a state-of-the-art algorithm for checking graph isomorphism is *canonical labeling* [63], which works well with sparse graphs, but not with dense graphs. Our previous experiment also confirmed this: it took 3,151 seconds, i.e. nearly one hour to produce the unique canonical label for a graph with 388 nodes and 410 edges [52].

Thus, the second design strategy aims to reduce the computation time of graph isomorphism. Rather than comparing graphs for exact isomorphism, we employ an approximate signature-based approach called Exas [52]. That is, we produce for each graph a signature capturing its structural information, and if two graphs have the same signature, we consider them to be label-isomorphic. In our implementation, the signature of a graph is the hashcode of its corresponding Exas characteristic vector. This vector counts the number of occurrences of short label sequences of nodes existing in that graph.

Exas was shown to be highly accurate, efficient, and scalable [52, 58]. For example, it took about 1 second to produce the vector for the aforementioned graph. It is about 100% accurate for graphs with sizes less than 10, and 94% accurate for sizes in 10–30. In our evaluation of GrouMiner, most patterns are of size less than 10, thus, Exas can work very well for them. In addition, the pattern candidates are generated incremental by size, thus, we also compute their Exas vector and signature incrementally. That is, if P' is a sub-graph of size $k+1$ extended from P, a sub-graph of size k, Exas vector of P' is computed from that of P by just counting occurrences of label sequences involving the added node, which are much faster than recounting all occurrences in P'.

```
function MinePattern(D)                                          1
  L ← {all patterns of size 1}                                   2
  for each pattern P ∈ L do Explore(P,L,D)                       3
return L                                                         4
                                                                 5
function Explore(P, L, D)                                        6
  for each pattern U of size 1 ∈ L do                            7
    C ← P ⊕ U                                                    8
    for each candidate Q ∈ Group(C)                              9
      if count(Q, D) ≥ σ then                                    10
        L ← L ∪ {Q}                                              11
        Explore(Q, L, D)                                         12
return L                                                         13
                                                                 14
function Group(C)                                                15
  for each graph X ∈ C do                                        16
    h = Hash(Vector(X))                                          17
    Gr[h] ← Gr[h] ∪ {X}                                          18
return Gr                                                        19
```

Figure 2.5 Pattern mining algorithm

2.2.2.4 Strategy 3. Approximated occurrence counting

The third design strategy address the difficulty in finding the maximal set of non-overlapping sub-graphs in calculation of frequencies of the pattern candidates. In fact, this is equivalent to the problem of maximum independent set on graphs, since the overlapping relation could be represented as a graph in which the sub-graphs could be considered as "nodes", and their overlapping relation could be considered as "edges". Therefore, instead of finding the maximal independent, i.e. non-overlapping, set of sub-graphs exactly, we find this approximately. That is, when a sub-graph is chosen to the independent set, its overlapping sub-graphs will be removed from the remaining set, i.e. will not be chosen.

2.2.3 Detailed algorithm steps

The pseudo-code of GrouMiner's mining algorithm is in Figure 2.5. D denotes the usage dataset, i.e. the collection of GROUMs extracted from code base. L denotes the

list of patterns. P denotes an individual pattern, stored in our algorithm as a set of occurrences (i.e. a sub-graph of a GROUM) in D. Q denotes a pattern candidates, also stored as a set of occurrences. X denotes an occurrence and C is a set of occurrences.

The algorithm first collects all patterns of size 1 (i.e. the smallest patterns) into L, the list of patterns (line 2). Then, each of such patterns is used as a starting point to recursively discover larger patterns by function Explore (line 3). The main steps of exploring a pattern P are: 1) generating from P the occurrences of candidate patterns (line 8), 2) grouping those occurrences into isomorphic groups (function Group) and considering each group to represent a candidate pattern (line 9); 3) evaluating the frequency of each candidate pattern to find the true patterns and recursively discovering larger patterns from them (lines 10-12).

2.2.3.1 Generating occurrences of candidate patterns

In the algorithm, each pattern P is represented the set of its occurrences in the whole usage dataset ($G_i(P)$ denotes the set of occurrences of P in G_i). Each occurrence $X \in G_i(P)$ is a subgraph and it might be extended into a larger subgraph by adding a new node Y and all edges connecting Y and the nodes of X (i.e. Strategy 1). Let us denote that graph $X + Y$. Since a large pattern must contain a smaller pattern, Y must be a frequent subgraph, i.e. an occurrence of a pattern U of size 1. This will help to avoid generating non-pattern subgraphs (i.e. cannot belong to any larger pattern). The operation $\oplus$ is used to denote the process of *extending* and *generating* all occurrences of candidate patterns from all occurrences of such two patterns P and U:

$$P \oplus U = \{X + Y | X \in G_i(P), Y \in G_i(U), i = 1..n\}$$

2.2.3.2 Finding candidate patterns

To find candidate patterns, function Group is applied on C, the set of all generated occurrences. It groups them into the sets of isomorphic subgraphs, using grouping criteria

based on Exas vectors (i.e. Strategy 2). That is, all subgraphs having the same vector are considered as isomorphic, i.e. are the occurrences of the same candidate pattern and thus, are collected into the same set (lines 17-18). Then, for each of such candidate Q, the corresponding subgraphs are grouped by the graph that they belong to, i.e. are grouped into $G_1(Q), G_2(Q), ...G_n(Q)$, to identify its occurrences in the whole usage dataset.

2.2.3.3 Computing frequencies

Function count(Q, G_i) is used to evaluate the frequency (i.e. number of occurrences) of Q in each graph G_i. In general, such evaluation is equivalent to the maximum independent set problem because it needs to identify the maximal set of non-overlapping subgraphs of $G_i(Q)$. However, for efficiency, we use a greedy technique to find a non-overlapping subset for $G_i(Q)$ with a size as large as possible (Strategy 3). GrouMiner sorts the occurrences in $G_i(Q)$ descendingly by their numbers of nodes that could be added to them. Then, it selects those occurrences by that order. As an occurrence is chosen in that order, its overlapping occurrences are removed. Thus, the resulting set contains only non-overlapping occurrences. Its size is assigned to $f_i(Q)$. After all $f_i(Q)$ values are computed, the frequency of Q in the whole dataset is calculated:

$$\text{count}(Q, D) = \text{count}(Q, G_1) + \text{count}(Q, G_2) + ... + \text{count}(Q, G_2)$$

If count$(Q, D) \geq \sigma$, Q is considered as a pattern and is used to recursively extend to discover larger patterns.

2.2.3.4 Disregarding occurrences of discovered patterns

Since the discovery process is recursive, occurrences of a discovered pattern could be generated more than once. (A sub-graph of size $k + 1$ might be generated at most $k + 1$ times from the sub-graphs of size k it contains.) To avoid this redundancy, when generating the occurrences of candidate patterns, function Explore checks if a sub-graph

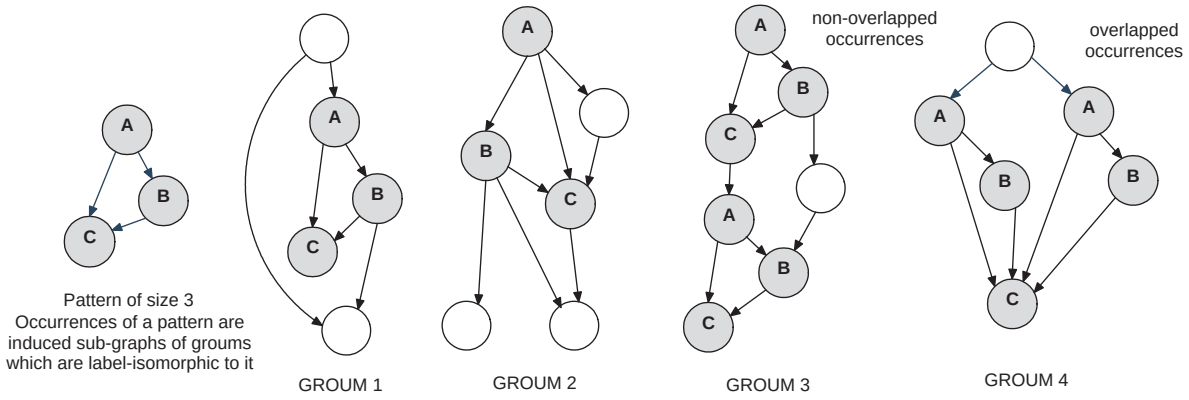

Figure 2.6 Pattern and occurrences

is an occurrence of a discovered pattern by comparing its Exas vector to those of stored patterns in L. If the answer is true, that sub-graph is disregarded in $P \oplus U$.

2.2.3.5 Running example

Let us explain the algorithm for the example in Figure 2.6. Assume that the threshold σ is chosen by 5. To find the pattern of size 3 as in the figure, we start with the pattern of size 1. They are the frequent nodes labeled A, B, and C. Now, the exploration starts for the pattern P as the node A. First, the algorithm generates occurrences of candidates from that of A. Each occurrence of A in a graph is extended with occurrences of B (and occurrences of C but in the next iteration). We now have the set of occurrences of AB. Function Group groups them into only one group, because they are all label-isomorphic. Therefore, we have only a candidate Q as the sub-graph AB. Calculating its frequencies, in the first two graphs, the frequencies of $G_1(Q)$ and $G_2(Q)$ are of 1, in the last two, the frequencies of $G_3(Q)$ and $G_4(Q)$ are of 2. Therefore, the total frequency of Q is of 6. It exceeds the threshold σ, therefore Q is considered a pattern, is added to the list L, and is used to recursively explore.

The next recursive exploration now from the pattern P as AB. Its occurrences are extended by the occurrences of C to have sub-graphs ABC. Note that, when a node of

C is added to a sub-graph AB, all the edges to node C, i.e. both AC and BC, are added to form an induced sub-graph containing A, B, and C. After generating such sub-graphs, function Group groups them into isomorphic group. In this case, we also have only one isomorphic group, therefore, there is only a candidate Q. Calculating its frequencies, we have that of $G_1(Q)$ and $G_2(Q)$ is of 1, that of $G_3(Q)$ is of 2. However, the frequency of $G_4(Q)$ is just 1 because when we choose a sub-graph ABC in G_4, the other is removed as it overlaps with the chosen one. Then, ABC has total frequency of 5, i.e. is considered another pattern, and is added to L.

However, using it for further exploration, we could not find any new pattern. Therefore, the recursion is back-tracked to the exploration of the pattern AB. Since we have no remaining frequent nodes to extend the occurrences of AB, the recursion is back-tracked to the exploration of the pattern A. Fortunately, the occurrences of A are able to be extended with occurrences of C to have sub-graphs AC. Similarly to the pattern AB, the algorithm also finds that AC is a pattern and use it to explore recursively. However, when occurrences AC are extended with the occurrences of B, the algorithm finds that the generated sub-graphs ACB are the occurrences of the discovered pattern ABC and discard them. Thus, no more pattern is discovered. By back-tracking, the algorithm returns from the exploration of A. It then starts the exploration of B and detects a new pattern BC. At last, the algorithm starts the exploration of C and finds no more pattern. At that time, the list L has the following patterns A, B, C, AB, AC, BC and ABC. By default, GrouMiner reports only ABC because it includes all other patterns.

2.2.4 Pattern-based bug detection

The mined usage patterns can be used to automatically find the potential API reuse-related bugs. In this work, we adapt the definition of usage anomalies from the prior work [73] for our graph-based representation and consider usage anomalies the potential API reuse-related bugs. Intuitively, an anomaly is a rare deviated usage of a pattern.

In term of graph-based representation, we consider an anomaly as a strict sub-graph of a pattern (i.e. it contains some but not all nodes and edges of that pattern) which has a low frequency in the whole usage dataset. This suggests that, the developers might have tried to use the pattern, but missed some of its steps. (Otherwise, if the usage has a high frequency, it might be an occurrence of another pattern).

Figure 2.7 shows an example where a BufferedReader is used without calling close. P is a usage pattern with a BufferedReader. P_1 is a sub-graph of P, containing only two action nodes <init> and readLine. A GROUM G contains an occurrence of P, thus contains also another occurrence G_1 of P_1 as a subgraph of that occurrence of P. Another GROUM H contains an occurrence H_1 of P_1 but no occurrence of P. Since P_1 is a sub-graph of P, H_1 is called an *inextensible* occurrence of P_1 (i.e. it could not extend to an occurrence of P), thus is considered to *violate* P. Because containing H_1, H is also considered to violate P. In contrast, G_1 is *extensible*, thus, G_1 and G do not violate P.

However, not all violations are considered as defects. For example, there might exist the occurrences of the usage <init>−close (without readLine) that also violate P, but they are acceptable. A violation is considered as an anomaly when it is *too rare*. The rareness of the violations could be measured by the ratio $v(P_1, P)/f(P_1)$, with $v(P_1, P)$ is the number of *inextensible* occurrences of P_1 corresponding to P in the whole dataset. If rareness is smaller than a threshold, corresponding occurrences are considered as anomalies. The lower a rareness value is, the higher the anomaly is ranked.

Definition 6 *A GROUM H is considered as a usage* ***anomaly*** *of a pattern P if H has an* inextensible *occurrence H_1 of a sub-graph P_1 of P and the ratio $count(P_1, P)/count(P_1, D) < \delta$, with $count(P_1, P)$ is the number of such inextensible occurrences in the whole usage dataset and δ is a chosen threshold.*

After mining patterns, GrouMiner performs anomaly detection. The main task of anomaly detection is to find the inextensible occurrences of all patterns P_1 corresponding to the detected patterns. In the first case, because storing the occurrence set $D(P_1)$,

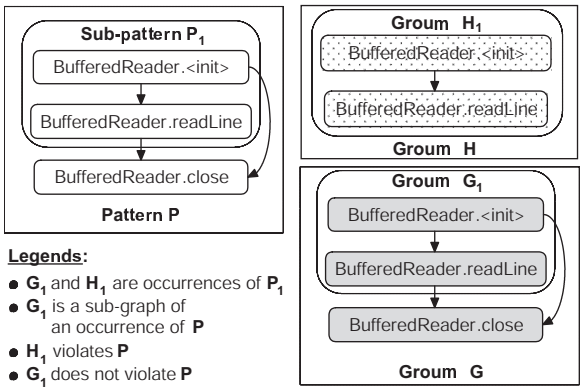

Figure 2.7 An example of violations of usage patterns

GrouMiner can check each occurrence of P_1: if it is inextensible to any occurrence of a detected pattern P generated from P_1, then it is a violation. Those violations are counted via count(P_1, P). After checking all occurrences of P_1, the rareness value count(P_1, P)/count(P_1, D) is computed. If it is smaller than the threshold δ, such a violation is reported as an anomaly. In the second case, GrouMiner must update the occurrence sets of detected patterns before finding the anomalies in the new version.

2.3 Empirical Evaluation

We have implemented GrouMiner for Java language and evaluated it on nine Java projects. In the experiments, we evaluated the performance of its mining process, the quality of the patterns it mined, and the accuracy of its pattern-based bug detection. All experiments were carried out in a computer with Windows XP, Intel Core 2 Duo 2Ghz, and 3GB RAM.

2.3.1 Subject systems

Table 2.2 lists nine subject systems and their usage datasets in our experiment. All subject systems are open-source, written in Java, and belong to different application domains. For example, Ant is a build tool, Columba is an email client, and jEdit is a

Table 2.2 Subject systems used in the evaluation of GrouMiner

System	Class	Method	GROUM	Max size
Ant 1.7.1	1,123	12,409	9,573	153
AspectJ 1.6.3	1,500	14,716	9,818	332
Axis 1.1	1,127	7,834	5,355	425
Columba 1.4	799	5,083	3,024	185
Fluid 12.05	229	3,506	2,477	115
jEdit 3.0	204	2,274	1,757	244
Jigsaw 2.0.5	701	6,528	5,073	152
Log4J 1.2.15	292	2,479	1,763	99
Struts 1.2.6	365	3,209	2,412	107

text editor. In the table, columns Class and Method describe the sizes of those systems in term of their number of classes and methods. As seen, the largest system, AspectJ, has nearly 15,000 methods. Columns GROUM and Max size represent the number of extracted usage models and the maximal size of them. The largest system, AspectJ, has nearly 10,000 usage models. The number of extract usage models is smaller than the number of methods since source code of many methods do not evolve object usages (some methods, for examples on of interface classes, are even empty). The largest usage model is extracted from Axis with 425 nodes in total.

2.3.2 Performance of pattern mining process

Table 2.3 shows the mining result of GrouMiner in nine subject systems, with the same frequency threshold σ of 6. Columns Pattern and Max size show the number of mined patterns and their maximal size in each system. The next four columns are numbers of patterns of different sizes. As seen, GrouMiner is able to mine several hundreds of patterns and some patterns can be as large as 17 nodes. It should be noted that GrouMiner reports only distinct patterns and does not report patterns that are contained within others. The numbers of mined patterns with sizes larger than 2 are about 44%–69% of the total numbers. This shows an advantage of GrouMiner over existing approaches,

Table 2.3 Running time and mined patterns

System	Pattern	Max size	Size 2	Size 3-5	Size 6-10	Size > 10	Time
AspectJ 1.6.3	1,055	15	429	413	180	33	69 min
Jigsaw 2.0.5	443	11	197	204	41	1	27 min
Ant 1.7.1	697	17	317	315	62	3	22 min
Axis 1.1	614	16	251	258	100	5	12 min
Fluid 12.05	236	14	92	94	46	4	9 min
jEdit 3.0	238	10	119	77	42	0	2 min
Struts 1.2.6	198	8	62	114	22	0	2 min
Columba 1.4	219	7	118	94	7	0	1 min
Log4J 1.2.15	141	10	79	60	15	0	1 min

which focus on patterns of pairs or a set of pairs of method calls.

The last column is the total running time. As seen, the running time depends more on the distribution nature of patterns and usage models of each system, rather than its dataset size. For example, Ant and AspectJ have similar number of extracted usage models (approximately 10,000). However, the number of patterns mined in AspectJ is nearly twice that of Ant. In addition, AspectJ has 33 mined patterns with size larger than 10, while Ant has just 3. Thus, the running time of AspectJ is about 3 times that of Ant. Systems like Log4J and Columba have very short running time, since they do not have many patterns, especially patterns of large sizes (which is more time-consuming to mine). Nevertheless, the pattern mining time is very reasonable (a few minutes for small systems, to tens of minutes for larger ones). The largest system, AspectJ, has around 500 KLOCs and is processed in around an hour.

2.3.3 Quality of mined patterns

As seen in Table 2.3, GrouMiner mined from nine different open-source projects for the total of nearly 4,000 patterns. It is impractical to examine all of them. Therefore, we studies only a sample set of those patterns and selected to present some interesting patterns among them.

a. Interleaving pattern: change a document model with version control

```
SCUmlDocument doc = model.getDocument();
ConfigController c = model.getConfigController();
Version initial;
VersionTracker tracker;
doc.parent(node);
do {
  tracker = c.getVersionTracker();
  initial = tracker.getVersion();
  Version.setVersion(initial);
  IRNode locNode = doc.getNodeWithName(node, "location");
  if (locNode == null)
    locNode = doc.createNode("location");
  doc.setAttr(locNode, "x", theLoc.width+"");
} while (!tracker.moveFromVersionToCurrent(initial));
```

b. Individual pattern: access and modify a node in a document model

```
SCUmlDocument doc = model.getDocument();
doc.parent(node);
IRNode locNode = doc.getNodeWithName(node, "location");
if (locNode == null)
  locNode = doc.createNode("location");
doc.setAttr(locNode, "x", theLoc.width+"");
```

c. Individual pattern: version control of a document model

```
ConfigController c = model.getConfigController();
Version initial;
VersionTracker tracker;
do {
  tracker = c.getVersionTracker();
  initial = tracker.getVersion();
  Version.setVersion(initial);
} while (!tracker.moveFromVersionToCurrent(initial));
```

Figure 2.8 Usage patterns mined from Fluid

Example 1. Figure 2.8 shows example patterns that GrouMiner mined from Fluid project. The code in Figure 2.8a contains a usage pattern in Fluid that set up the Fluid version controller to track the changes to an UML element in a graphical editor. The particular type of changes to be tracked in that code is that of the element's location on screen. The individual pattern to change the location of an UML element is listed Figure 2.8b, This pattern involves the retrieval of an UML element object, the setting of the parent node, the checking for the existence of the "location" node, and the setting of the new value for the "location" attribute. The individual pattern which sets up the tracker and monitors the changes is listed in Figure 2.8c.

GrouMiner is able to detect those two patterns even though they interleave with each other in the code. Each pattern involves multiple objects interacting with one another. For example, the pattern in Figure 2.8b involves 4 objects and 5 method invocations. The pattern in Figure 2.8c also involves 4 objects, 5 method invocations, and a while loop. Interestingly, the entire procedure of tracking changes to the location of UML elements was also detected as a pattern. The reason is that this procedure frequently occurs due to the needs of tracking changes to different types of UML elements in Fluid's editors. Since GrouMiner discovers the patterns from the smallest to the largest sizes, it is able to detect all three patterns (two smaller patterns connect via data sharing and usage order edges).

Example 2. Figure 2.9 shows another example mined from Ant. The piece of code in Figure 2.9a contains a pattern to test a mail server with a client-server paradigm. Similar to Fluid's example, GrouMiner is able to detect three patterns. The first pattern is the steps to initiate a server thread, which involves two objects: a ServerThread and a Thread (Figure 2.9b). The second pattern is the procedure to launch the client thread and to test the returned result. There are also two interplaying objects: a ClientThread and a Thread (Figure 2.9c). Unlike in the Fluid's example, there is no intra-procedural data dependency between objects in two patterns. However, the temporal orders between

a. Interleaving pattern: start both server and client threads

```
ServerThread testMailServer = new ServerThread();
Thread server = new Thread(testMailServer);
server.start();
ClientThread testMailClient = new ClientThread();
testMailClient.from("...TaskTest...ant.apache.org...");
testMailClient.setSubject("Test subject");
testMailClient.setMessage( "...line 1... ");
Thread client = new Thread(testMailClient);
client.start();
server.join(60 * 1000);
client.join(30 * 1000);
String result = testMailServer.getResult();
if (testMailClient.isFailed())
  fail(testMailClient.getFailMessage());
```

b. Individual pattern: start a server thread

```
ServerThread testMailServer = new ServerThread();
Thread server = new Thread(testMailServer);
server.start();
server.join(60 * 1000);
String result = testMailServer.getResult();
```

c. Individual pattern: start a client thread

```
ClientThread testMailClient = new ClientThread();
testMailClient.from("...TaskTest...ant.apache.org...");
testMailClient.setSubject("Test subject");
testMailClient.setMessage( "...line 1... ");
Thread client = new Thread(testMailClient);
client.start();
client.join(30 * 1000);
if (testMailClient.isFailed())
  fail(testMailClient.getFailMessage());
```

Figure 2.9 Usage patterns mined from Ant

```
StringBuffer sb = new StringBuffer();
sb.append("{");
for (Iterator iter = supportedTargets.iterator(); iter.hasNext();) {
  String evalue = (String) iter.next();
  sb.append(evalue);
  if (iter.hasNext()) sb.append(",");
}
sb.append("}");
return sb.toString();
```

Figure 2.10 A common usage pattern of Java API mined from AspectJ

Table 2.4 Accuracy of pattern-based bug detection

System	Reported	Checked	Bug	Code smell	False positive
Fluid 12.05	64	64	5	8	40
AspectJ 1.6.3	244	15	1	2	12
Jigsaw 2.0.5	115	15	1	1	13
Ant 1.7.1	145	15	1	0	14
Columba 1.4	40	15	1	0	14
jEdit 3.0	47	15	1	0	14
Axis 1.1	145	15	0	2	13
Struts 1.2.6	33	15	0	0	15

method calls in an individual pattern and between calls in two patterns are important and captured as edges (e.g. a server thread is *started* before a client thread). These temporal properties are exhibited frequently as well. Moreover, this example shows that GrouMiner is able to handle two objects server and client of the same type Thread.

Example 3. Figure 2.10 shows another pattern mined from AspectJ to illustrate a routine to convert a Set to a String using StringBuffer and Iterator objects. GrouMiner is able to detect this pattern with four interplaying objects and the control structures for, if among method calls. For object iter, JADET [73], a well-known object usage miner, would produce a pattern $P = \{$hasNext() <hasNext(), hasNext() <next()$\}$ ($<$ means "occurs before"), thus providing less information.

```
public void setLocation(SCThornModel model, IRNode node, Point thePt) {
  SCUmlDocument doc = model.getDocument();
  doc.parent(node);
  ...
  IRNode locNode = doc.getNodeWithName(node, "location");
  // missing a check and a call : if (locNode == null) locNode = doc.createNode("location");
  doc.setAttr(locNode, "x", thePt.x+"");
  ...
}
```

Figure 2.11 Null Pointer Exception due to missing of a check for existence

2.3.4 Pattern-based bug detection

Table 2.4 shows result of pattern-based bug detection of GrouMiner on the subject systems, with the threshold δ of 0.1. The total number of anomalies detected on each system is reported in column Reported. Due to time constraint, we examined only the top 15 anomalies for each system. For Fluid, we examined all 64 reported anomalies because for we have the domain knowledge of that project.

2.3.4.1 Evaluation result on Fluid

From 64 anomalies reported on Fluid, we have found 5 programming errors (defects) that have not been yet discovered. Figure 2.11 shows an error, which is a violation of the pattern in Figure 2.8b. That is, before calling doc.setAttr to change locNode, the code in Figure 2.11 does not check whether that node exists, and if it does not, create such a node, as in the pattern in Figure 2.8b. In our manual verification, this violation lead to a Null Pointer Exception, thus, the program crashed when it reached that method and no IRNode with the name of location existed yet.

Figure 2.12 lists another defect occurs in method changeProperty of class SCThornDiagramElementVersion. This method also violates the pattern of tracking the changes to the properties of a UML graphical element in Figure 2.8b. It was supposed to check the existence of an IRNode with the name Property by calling getNodeWithName before it

```
public void changeProperty(SCThornModel model, ...) {
  SCUmlDocument doc = model.getDocument();
  doc.parent(node);
  ...
  // missing a call: IRNode propertyNode = doc.getNodeWithName(node, "Property");
  // and a check: if (propertyNode == null)
  propertyNode = doc.createNode("Property");
  doc.setAttr(propertyNode, "name", name);
  doc.setAttr(propertyNode, "value", value);
  doc.addChild(node, propertyNode);
  ...
}
```

Figure 2.12 Creating a duplicate node due to missing of a check for existence

called createNode. In this case, the defect did not cause a program to crash. However, it is harder to detect because document doc would have more than one Property nodes, thus, creating a semantic error.

We also found three instances of the third defect in Fluid. They violate the following pattern: if (IRNode.valueExists(IRAttr)) IRNode.getSlotValue(IRAttr). The pattern means that one must check the existence of an attribute (by calling valueExists) before getting its value (by calling getSlotValue). Those three locations did not have the if statement with that checking expression and caused program errors.

In total, we had manually examined all 64 violations in Fluid and classified them into 1) bugs (i.e. true defects), 2) code smells (any program property that indicates something may go wrong), and 3) hints (i.e. code that could be improved for readability and understandability). We used the same classification as in JADET [73]. Among 64 anomalies, there were 5 defects, 8 code smells, 11 hints, and 40 false positives. Among the top 10 anomalies in Fluid, 3 of them are defects, two are code smells, one is a hint, and 4 of them are false positives. We confirmed the reported bugs by running/testing the program. In this case study, the false positive rate is 40/64 = 62.5%. In [73], the reported false positive rate of JADET on AspectJ was 87.8%.

2.3.4.2 Evaluation result on other systems

In addition to Fluid, for eight other systems, we examined the top 15 anomalies and manually classified them. These case studies show that our graph-based ranking approach is successful. GrouMiner can reveal 5 new defects in even mature software like Columba and jEdit. Carefully examining those defects, we found that they are in the form of missing necessary steps in using the objects and missing condition and control structures. For example, in the method PointcutRewriter.simplifyAnd of AspectJ, the call of Iterator.next was not preceded by an Iterator.hasNext. Similarly, in the method MapEntry.parseRestNCSA of Jigsaw 2.0.5, the call to StringTokenizer.nextToken was not preceded by a call to StringTokenizer.hasNext.

On the other hand, in the method AbstractMessageFolder.recreateMessageFolderInfo of Columba, a call to ICloseableIterator.close is missing in the usage involving an ICloseableIterator object. The method Registers.toString of jEdit also misses a call to BufferedReader.close when it uses a BufferedrReader object. The discovered patterns with all required steps have enabled the detection of those errors.

2.3.5 Discussion

As seen from the evaluation, GrouMiner is able to detect patterns of high quality with reasonable running time. The mined patterns represent the code that performs common programming tasks in the corresponding software systems (e.g. adjusting nodes' attributes with versioning control tracking in Fluid). In addition, the mined patterns are both common (i.e. using common external APIs like Java API in Example 3) or project-specific (i.e. using internal APIs such as the patterns in Example 1). Interestingly, GrouMiner is able to mine the interleaving patterns (of large size), which present how developers combine patterns for individual tasks into a patterns for a bigger tasks. Due to such characteristics, the patterns mined by GrouMiner will be very useful for the developers to learn, not only for using individual APIs and but also for combining APIs

of different packages for the recurring tasks in the project.

The evaluation also shows that, it is also possible to use patterns to detect usage errors. Those errors often occur because the developers miss some steps in the usages, for example they do not check for the existence of nodes or attributes before accessing and modifying them, or do not close resource objects after using them. However, pattern-based bug detection still produces high numbers of false positives. There are several possible reasons for those false positives. First, usage patterns are the frequent, and often preferred ways to reuse APIs, but they are not the only correct ways. For example, one often uses while loops to reading files with Scanner, but it is possible to replace the while loops by for loops or do while loops. Thus, reading files using Scanner objects and for loops would be rare (and deviates from the common usage pattern using while loops), however are still correct. In addition, sometimes due to the specific situations, missing a step in the pattern does not cause an error. For example, if the developer is certain that an Iterator has available elements to read (i.e. the collection is not empty), he does not need to call hasNext before accessing those elements by next. More importantly, GROUM is just a model of program semantics and executions using static analyses with limited capability. For example, in our current implementation, we just use intra-procedural analysis for temporal usage orders and use simple data flow analysis via variable sharing for data dependencies. That means, some programming properties captured by GROUM (e.g. method a is called before method b) might not be the exact specification of the program, and thus, violating them might not lead to errors.

2.3.6 Pattern-based code completion

The evaluation suggests that the patterned detected from source code present the code snippets that are frequently reused to program common and recurring tasks in a software project. It is also shown that, developers often make errors when some steps in the patterns are missing. Those errors would be avoided if they are aware about the

patterns and follow those patterns exactly. Motivated by this idea, we have developed Grapacc, a code completion tool that can suggest usage patterns for the code developers are editing and fill in selected patterns when requested.

Code completion is a useful feature of code editors, and is often the built-in feature of modern IDE like Eclipse or Visual Studio. While editing code, a developer could request code completion, and the code editor will suggest several options for next code tokens. Built-in code completion in Eclipse or Visual Studio currently just is able to recommend a method call or a variable at the time. Thus, developers unfamiliar with the APIs still have to choose individual method calls for the API usage. This still leaves chances for API usage errors, when developers miss or call a wrong method.

In contrast, Grapacc is able to recommend the whole usage pattern (with several method calls, objects, and involving control flow statements). Armed with an extensible knowledge base of patterns, it analyzes the current editing code, determines the missing parts, and recommends the most suitable patterns. Figure 1.7 demonstrates the running of Grapacc in a usage scenario, when Grapacc recommends several usage patterns, with previews of the code if a pattern is selected. When a pattern is chosen, Grapacc will automatically fill that pattern in, completing the missing parts of the usage. Since the whole pattern is filled in at the same time, programmers are less likely to miss some steps and make usage errors. They also write code faster since most of the code has been filled by Grapacc.

Grapacc internally uses GROUM to represent patterns and editing code. Grapacc uses a compound formula to compute the relevancy between the editing code and a pattern, including factors related to the frequency of the pattern, their structural similarity (e.g. the orders of method calls, the involving control statements), the textual similarity (e.g. class and method names), and the editing context (e.g. the location of the editing cursor). Recommended patterns are ranked based on their total relevancies to the editing code, allowing the user to select them easier. Finally, Grapacc is context-sensitive,

i.e. when the user changes the code or editing position, the recommendations are also changed accordingly. Full details on Grapacc could be referred in [50].

CHAPTER 3. RECURRING BUG FIXES

As seen from the previous chapter, using usage patterns to detect bugs results in high number of false positives because the assumption that "usages deviated from good ones are bad" is not always correct in practice. That is because patterns are not the only good way to reuse. In this chapter, we investigate on the complement philosophy, i.e. "usages similar to bad ones are bad". Our approach focuses on understanding and then detecting recurring bugs. Analyzing bug fixing changes on five subject systems, we found that up to 40% of those changes are highly similar and could be considered to be the fixes of recurring bugs. More importantly, those fixes occur on code units having highly similar GROUM. We call those units "code peers", because they have similar roles and functionality in the system. Based on this empirical study, we have developed FixWizard, a method that could scan a given software system for code peers and monitor their changes. Then, when a code peer has a bug and gets fixed, FixWizard will alert the potential bugs recurring in the similar code units and recommend similar fixes. This chapter presents our empirical study of recurring bug fixes and FixWizard in full details.

3.1 Empirical Study of Recurring Bug Fixes

Previous research had reported the existence of recurring bug fixes [36]. A bug fixing change is considered recurring if it is repeated identically or with relevant, slight modifications on several code fragments at one and/or multiple revisions. This existence inspires us with many research questions: Why, where, and how often do such changes

Table 3.1 Subject systems used in the empirical study

System	Domain	Revision range	Fixes
ArgoUML	Graphic modeling	2 - 1,130	2,318
Columba	Mail client	4 - 370	829
Eclipse	Development tool	400 - 10,300	1,126
FlashRecruit	Job listing	100 - 600	1,007
ZK	Web framework	2,400 - 6,200	490

recur? How could they be characterized and recognized? And, importantly, how could we use them to help the developers in fixing future bugs?

Aiming answer those questions regarding recurring bug fixes, we conducted an empirical study with manual examination of existing bug fixes. The study has two parts. First, a group of experienced programmers was asked to examine all fixing changes of the subject systems and manually identify the similar ones. Then, we analyzed their reports to characterize such similar fixing changes and their locations in order to verify our hypothesis: similar fixes tend to occur on similar code units, i.e. ones providing similar functions and/or participating in similar interactions, in term of object usages.

3.1.1 Subject systems and bug fixing changes

Table 3.1 shows subject systems used in our study, two of them were also examined by Kim *et al.* in previous research on bug fixes [36]. We first identified the bug fixing revisions of our subject systems. For each fixing revision in a system, we consider all code changes to a method as an atomic bug fix. Then, seven Ph.D. students in Software Engineering at Iowa State University with the average of five years of experience in Java examined those fixes and sorted them into groups of fixes that are recurring. Conflicting assessments were resolved by the majority vote among them. In fact, there were only two disputed groups of two recurring fixes.

```
public void setColspan(int colspan) throws WrongValueException{
 if (colspan <= 0) throw new WrongValueException(...);
 if (_colspan != colspan) {
  _colspan = colspan;
 final Execution exec=Executions.getCurrent(); if (exec!=null && exec.isExplorer()) invalidate();
  smartUpdate("colspan", Integer.toString(_colspan));...
```

```
public void setRowspan(int rowspan) throws WrongValueException{
 if (rowspan <= 0) throw new WrongValueException(...);
 if (_rowspan != rowspan) {
  _rowspan = rowspan;
 final Execution exec=Executions.getCurrent(); if (exec!=null && exec.isExplorer()) invalidate();
  smartUpdate("rowspan", Integer.toString(_rowspan));...
```

Figure 3.1 Fixing changes at revision v5089 in ZK

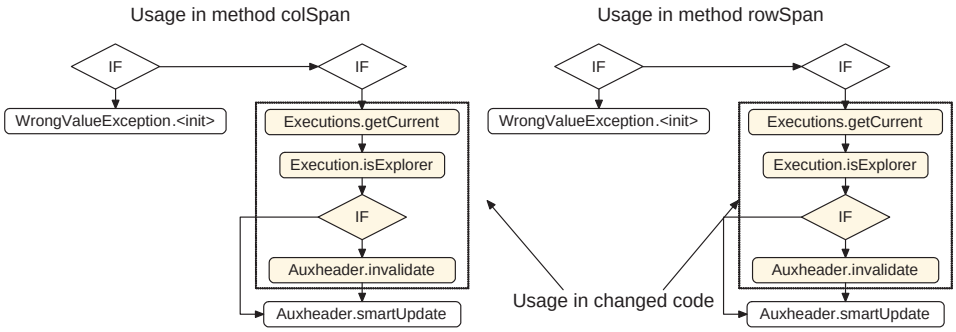

Figure 3.2 Graph-based object usage models for code in Figure 3.1

3.1.2 Analysis of recurring bug fixes

After obtaining the identified recurring bug fixes from the human subjects, we analyzed those fixes to understand the nature of the bugs fixed, of the code units (e.g. methods and classes) where the fixes were applied, and most importantly, the reasons why those fixes recur in several places. In this section, we will present three representative examples of the reported recurring bug fixes. Then, we will discuss the overall statistics and characteristics of those fixes.

3.1.2.1 Representative examples

Example 1. Figure 3.1 shows an example of recurring fixes taken from ZK[1], a Java framework for enterprise web and mobile applications. The figure lists two methods setColspan and setRowspan of class Auxheader. As seen, these two methods have highly similar code and actually provide highly similar functionality: adjusting column span or row span of an Auxheader object. However, their original code has the same bug: the methods adjust the span but do not update the user interface, making no change visible to the users. Thus, the same fixes, shown the boxes, have been applied to those methods. The fixed code locates the current Execution object. If it is not null and is an "Explorer" execution (checked by exec.isExplorer, then invalidate is called to redraw the user interface, making the newly adjusted span visible to users.

In this example, the same bug recur on two methods with highly similar code and functionality. It is possible that they were written via copy-and-pasting, i.e. the developer wrote one method and copy-and-pasted to create the other. Nevertheless, as they have similar functionality, the developer has made the same mistake when implementing them, and thus fix them with the same fixes. Thus, this example is an empirical evidence for the hypothesis that *"similar code has similar bugs and fixes"*.

We hypothesize that the similarity of the code, bugs, and fixes would be more recognizable under an abstract model of source code. Therefore, we use GROUM (developed and presented in Chapter 2 as an abstract model of source code) to further analyze those two methods. Figure 3.2 shows their GROUMs, with the changed parts also shown in the boxes. It should be reminded that, in GROUM representation, the nodes such as Executions.getCurrent or Auxheader.smartUpdate represent the invocations of the corresponding methods. An edge such as the one from Executions.getCurrent to Execution.isExplorer indicates their usage order, i.e. the former is called before the latter. As we could see, both methods have identical GROUMs, both before and after being fixed. That is, they

[1]http://www.zkoss.org - Accessed at 12:16 on 12/02/2013

```
public class UMLOperationsListModel extends UMLModelElementCachedListModel {
public void add(int index){
 Object target = getTarget();
 if (target instanceof MClassifier) {
  MClassifier classifier = (MClassifier)target;
  Collection oldFeatures = classifier.getFeatures();
  MOperation newOp = MMUtil.SINGLETON.buildOperation(classifier);
  classifier.setFeatures(addElement(oldFeatures,index,newOp, _operations.isEmpty()?null:
      _operations.get(index)));
```

```
public class UMLAttributesListModel extends UMLModelElementCachedListModel {
public void add( int index){
 Object target = getTarget();
 if (target instanceof MClassifier) {
  MClassifier classifier = (MClassifier)target;
  Collection oldFeatures = classifier.getFeatures();
  MAttribute newAt = MMUtil.SINGLETON.buildAttribute(classifier);
  classifier.setFeatures(addElement(oldFeatures,index,newAt, _attributes.isEmpty()?null:
      _attributes.get(index)));
```

Figure 3.3 Fixing changes at revision v0460 in ArgoUML

are implemented with the same object usage. Then, they were fixed (with modifications to their object usages) in the same way.

Example 2. Figure 3.3 shows another example of recurring bug fixes. In class UMLOperationsListModel, _operations is List object. The method add originally has a call to _operations.get(index). According to Java documentation, if a List object is empty, calling get(index) will cause an IndexOutOfBoundsException. To fix this bug, the developer adds the code to check whether _operations is empty, and if it is, a null object is used in place

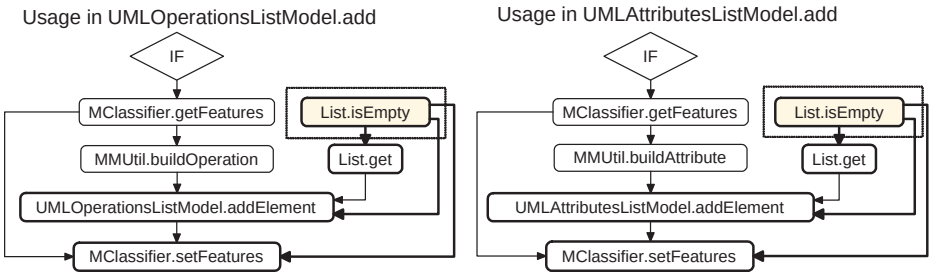

Figure 3.4 Graph-based object usage models for Figure 3.3

of a call to _operations.get. The same bug also occurs in the method add of class UMLAttributesListModel, which also has a call to _attributes.get(index) on the List object _attributes. Thus, the same fix has been applied on that method.

Analyzing the classes and methods contain these fixes, we found several interesting observations. First, two classes UMLOperationsListModel and UMLAttributesListModel inherit from the same class UMLModelElementCachedListModel. The majority of their methods, including two methods add shown in the figure, are very similar and in fact, override the same methods in their parent class UMLModelElementCachedListModel. Thus, those two classes could be considered as clones in class level, and have the similar roles, both in function and interaction with other classes.

The GROUMs representing the interactions of the two methods add with other classes/methods are shown in Figure 3.4. As seen, they have identical structures, and if we consider two methods buildOperation and buildAttribute of class MMUtil, as well as the two methods addElements of classes UMLOperationsListModel and UMLAttributesListModel having the same role, the two usages could be considered representing the same routines. In these routines, two List objects _operations and _attributes are used in the same way (as a caching mechanism) and their usages are changed in the same manner, i.e isEmpty should be checked before calling get on a List object.

Example 3. Figure 3.5 shows another case. The fixes (in the boxes) are very similar although the enclosing methods are not much similar to each other as in the previous examples. However, analyzing the usages of the enclosing classes TableController and TreeController, we found that the two classes are used only once, and used together, in the context of the class ThreePaneMailFrameController (Figure 3.6). Figure 3.6 shows the code and the GROUMs representing such usage scenarios. It could be seen that two classes TableController and TreeController are used in the similar ways in ThreePaneMailFrameController (and also in the whole system). This explains why their constructors are changed similarly, resulting in recurring fixes. That is, they need to interact to their re-

```
public class TableController implements TreeSelectionListener{
public TableController(MailFrameController mailFrameController){
 this.mailFrameController = mailFrameController;
 headerTableItem = (TableItem)MailConfig.getMainFrameOptionsConfig().getTableItem();
 headerTableModel = new HeaderTableModel(headerTableItem);
 view = new TableView(headerTableModel);
 tableSelectionManager = new TableSelectionManager();
 mailFrameController.getSelectionManager().addSelectionHandler(new TableSelectionHandler(view));
 tableChangedListenerList = new Vector();
 actionListener = new HeaderTableActionListener(this); ...
```

```
public class TreeController implements TreeSelectionListener{
public TreeController(MailFrameController mailFrameController, TreeModel model){
 this.model = model;
 this.mailFrameController = mailFrameController;
 view = new TreeView(model);
 actionListener = new FolderTreeActionListener(this);
 treeSelectionManager = new TreeSelectionManager();
 mailFrameController.getSelectionManager().addSelectionHandler(new TreeSelectionHandler(view));
 view.addTreeWillExpandListener(this); ...
```

Figure 3.5 Fixing changes at revision v0225 in Columba

spective MailFrameController object in the same manner (i.e. adding to its SelectionManager a relevant SelectionHandler object for their corresponding views).

Another interesting point is that, the interaction of TableController to TableView, TableSelectionManager, and TableSelectionHandler is *identical* to that of TreeController to TreeView, TreeSelectionManager, and TreeSelectionHandler. Examining such classes, we found that they follow the *Model-View-Controller* (MVC) design pattern. Therefore, they in pairs have the identical roles in the design of this system.

3.1.2.2 Code peers

All the methods having recurring bugs and fixes in previous examples share common nature that they have *similar object interactions* (in term of object usages, as represented by GROUMs). The similar interactions could be in their implementation code (Figures 3.1 and 3.3), i.e. where they use other classes and methods, or in their client code (Figure 3.5), i.e. where they are used by other classes and methods.

```
public ThreePaneMailFrameController(ViewItem viewItem) {
...
 trCtrl = new TreeController(this, FolderTreeModel.getInstance());
 tbCtrl = new TableController(this);
 TableSelectionHandler tbHdl = new TableSelectionHandler(tbCtrl);
 getSelectionManager().addSelectionHandler(tbHdl);
 TreeSelectionHandler trHdl = new TreeSelectionHandler(trCtrl.getView());
 getSelectionManager().addSelectionHandler(trHdl);
 tbCtrl.getView().addMouseListener(new TableMouseListener());
 trCtrl.getView().addMouseListener(new TreeMouseListener());
...
```

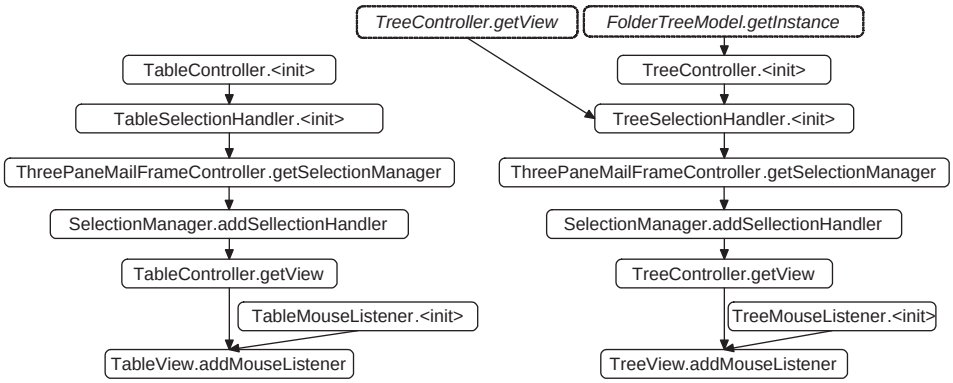

Figure 3.6 Usage of classes TableController and TreeController

The code units have the similar object interactions because they provide similar functionality and/or they have similar roles in the design of the system. For example, in Example 1, two methods setColspan and setRowspan have similar functionality of adjusting the span of the column or row of an Auxheader object. In Example 2, two methods adds have the similar functionality to add an element to either an UMLOperationsListModel object or an UMLAttributesListModel. Their classes could be considered to have the same role in the system design, as they inherit from the same class and have many methods with similar functionality. (They involve two similar and related concepts of UML *operations* and *attributes*). Thus, their two methods add could also be considered have the same role. In Example 3, two classes TableController and TreeController and their constructors also have the same role. Because they have similar functionality and/or similar roles in the system, we call them **code peers** (will be formally defined in Section 3.2).

Table 3.2 Recurring bug fixes

Project	Total fixes	Recurring	In code peers	% in Total	% in Recurring
Columba	829	377	332	40%	88%
ZK	490	188	171	35%	91%
FlashRecruit	1,007	244	224	22%	92%
Eclipse	1,126	215	185	16%	86%
ArgoUML	2,318	390	347	15%	89%

3.1.2.3 Recurring bug fixes on code peers

Table 3.2 shows the summary result of our empirical study. Column "Total fixes" lists the total number of examined bug fixes in each subject system. Column "Recurring" lists the total number of recurring bug fixes identified by our human subjects, while column "In code peers" lists ones that have been checked to occur on code peers. Columns "% in Total" and "% in Recurring" lists the percentage of such recurring bug fixes in the total examined bug fixes and the total identified recurring bug fixes, respectively. The table shows that almost all (86%–92%) recurring bug fixes occur on code peers. The recurring bug fixes on code peers also account for 15–40% of all bug fixes.

3.1.3 Discussion

Our empirical study provides two important observations. First, in a software system, there exist many code units (e.g. classes and methods) that have similar functionality or role, expressed by similar object interactions. Second, recurring bug fixes account for a reasonable amount of total bug fixes and most of them occur on code peers. Those observations could be explained based on the principles and practice of object-oriented programming in software development.

That is, in object-oriented programming, a system is expressed, i.e. designed and implemented, via objects and their interactions, which are realized in the classes/methods which provide the abstraction to the objects and their behaviors. The interaction of an object toward other objects is expressed in the implementation code of its class/methods,

in which it uses the other objects (i.e. internal usage). The interaction of other objects toward itself is expressed in its client code within other classes/methods in which it is used by other objects (i.e. external usage). In either case, the interactions of the objects could be realized via object usages, i.e. method invocations/field accesses, their usage orders, and the involved control structures.

In a large-scale system, there tends to exist several objects having similar functions or interactions with other objects. When they are implemented in source code, such similar functions/interactions are realized by classes and methods having similar object usages, which we called code peers. Bug fixing is to change the functions and interactions of objects. Similar functions and interactions usually need to be changed in the similar ways. This is the reason why similar fixing changes often occur on code peers.

As conventional in object-oriented programming, objects with similar functions will often be abstracted into parent classes. The specific behaviors are implemented in children classes. In other cases, the methods/classes might not be implemented in the similar ways, but they implement the same interface, i.e., promise the similar functions. The other objects could interact in the same way with the objects in such classes via their promised methods. The classes/methods having similar functions and/or being related via inheritance/interface will often be named similarly by the developers to help themselves in better understanding the roles of such classes/methods. In other cases, to implement the methods/classes having similar functions, developers tend to copy-and-paste the implementation code, thus creating similar code fragments. This is the reason why code peers (classes/methods having similar functions/interactions) tend to have similar code or names, or inherit from the same class, or implement the same interface(s).

3.1.4 Implication

Our empirical study confirms the common wisdom that *"similar code has similar bugs"*. More specially, the study implies that bugs recur often on code peers (up to

40% of total bug fixes). Thus, detecting code peers and monitoring them for recurring bugs and fixes would be useful for the early detection and resolution of bugs. Our study also finds several characteristics of those code peers. First, they are classes and methods that have similar functionality and role in the system, thus involving similar object interactions (e.g. method invocations, usage orders, etc). They often have similar structure and/or names and are related on the inheritance hierarchy. Peer classes tend to have several peer methods.

We have built FixWizard based on those implications. In general, its main task is to identify and monitor code peers. Then, when one peer gets fixed, FixWizard recommends similar fixes to other peers. Our approach characterizes code peers and the recurring changes made to them via object usages. That is, code peers are code units (e.g. methods/classes) having similar object usages, internally (i.e. in their implementation code) and/or externally (in the code using them). Recurring fixes at code peers are also the changes involving in similar object usages. Thus, FixWizard detects the changes in object usage models of code peers to derive the recommended bug fixes. Details of FixWizard will be presented in the next sections.

3.2 Concept and Formulation

In this section, we will define "code peers" and the related concepts such as usage and feature similarity. We introduce peer-isomorphism as a broader concept than label-isomorphism to model the similarity of object usage models of "code peers". Since checking peer-isomorphism and computing peer-based similarity would be computationally expensive operations, when developing the technique for detecting code peers, we will introduce an heuristic algorithm using the similarity of graph-based structural features in Section 3.3.

3.2.1 Code peers and usage similarity

3.2.1.1 Code peers

Definition 7 (Internal/External Usage) *Internal usage of a method A.m, denoted $UI(A.m)$, is the set of all usage models (GROUM) in the implementation code of A.m. External usage of A.m, denoted $UE(A.m)$, is the set of all usage models in the implementation code in the system, that could have an invocation of A.m.*

This definition also takes into account *dynamic binding* in object-oriented programming. That is, an invocation of $A_0.m$ or $I.m$ might actually be an invocation of $A.m$ if A is a subclass of class A_0 or A implements interface I.

Definition 8 (Peer) *Two methods are peers if and only if (iff) the usage similarity, measured by a function Sim, of their respective internal or external usages exceeds a pre-defined threshold. Two classes are peers iff the number of their peer methods exceeds a chosen threshold.*

Peer relation between methods/classes is denoted by $\equiv$. It is reflexive (a method/class is a peer of itself), symmetric (i.e. if x is a peer of y, then y is also a peer of x), and not transitive. Definition 8 could be written as: $A.m \equiv B.n$ iff $Sim(UI(A.m), UI(B.n)) \geq \sigma_1$ or $Sim(UE(A.m), UE(B.n)) \geq \sigma_2$, in which σ_1 and σ_2 are chosen thresholds.

Let us now present the formulation of Sim, the usage similarity measure between any two sets of graph-based usages.

3.2.1.2 Usage similarity measurement

Definition 9 (Peer-isomorphic Usage) *Two GROUMs are* peer-isomorphic*, if there exists a bijective (one-to-one) mapping for their nodes such that the mapped nodes represent the invocations of the same or peer methods, and their usage orders are the same.*

An illustrated example for peer-isomorphic usages is in Figure 3.4. Assume that buildOperation and buildAttribute of MMUtil, as well as addElement UMLOperationsListModel and UMLAttributesListModel are peer methods. Then, all the nodes between two GROUMs in Figure 3.4 could be mapped while the usage orders are still preserved. Thus, the two usages are peer-isomorphic. Note that, because peer relation is reflexive, peer-isomorphism for two GROUMs subsumes label-isomorphism.

However, the usages might not always be peer-isomorphic. They could be similar as in Figure 3.6. Therefore, we define the similarity of two object usages as follows.

Definition 10 (Usage Similarity) *Given two GROUMs G and H. Assume that G_o and H_o are their largest peer-isomorphic sub-graphs, respectively (the size of a GROUM is measured by its number of nodes). Then, the usage similarity of G and H is defined as $sim(G,H) = \frac{|G_o|+|H_o|}{|G|+|H|}$.*

Let us revisit Figure 3.5. Assume that all corresponding methods of TableXXX classes and TreeXXX classes are peer methods. Then, two graphs could be mapped such that two peer-isomorphic subgraphs have their sizes up to seven nodes. (Two methods FolderTreeModel.getInstance and TreeController.getView could not be mapped). Thus, the similarity of two usages is $(7+7)/(7+9) = 0.88$.

Using the usage similarity *sim* for any pair of GROUMs, we could define function *Sim* used in Definition 8 measuring the usage similarity of two methods as in the following.

Definition 11 (Similarity of Two Usage Sets) *The usage similarity of two sets of GROUMs U and V, $Sim(U,V)$, is the ratio between the total usage similarity of the maximum weighted matching between the members of U and V and their average size.*

This definition could formally written as

$$Sim(U,V) = \frac{max_M \sum_{(G,H)\in M} sim(G,H)}{(|U|+|V|)/2}$$

for all $M = \{(G, H) | G \in U, H \in V\}$ such that

$$\forall (G, H), (G', H') \in M : G = G' \Leftrightarrow H = H'$$

For example, assume that two methods have the external usage sets $\{G_1, G_2\}$ and $\{H_1, H_2\}$, respectively. The usage similarity of each pair is $sim(G_1, H_1) = 0.84$, $sim(G_1, H_2) = 0.36$, $sim(G_2, H_1) = 0.54$, and $sim(G_2, H_2) = 0.78$. Then, the maximum matching of two sets is (G_1, H_1) and (G_2, H_2), with the total similarity is $0.84 + 0.78 = 1.62$. Since their average size is 2, the usage similarity is $1.62/2 = 0.81$.

3.2.2 Recurring bug fixes in code peers

If code peers are modified, their corresponding object usages (represented by GROUMs) tend to be changed. The change of a GROUM might include the added, deleted, re-labeled, or edge-changed nodes. For example, in Figure 3.4, the node List.isEmpty is added. Then, three nodes (List.get, UMLOperationsListModel.addElement, and MClassifier.setFeature) are edge-changed, because they have the added edges due to the addition of List.isEmpty. In this case, we could say that the change affects all four nodes, and such impact could be represented by the sub-graph containing them.

Of course, the change of a GROUM could affect several nodes, and they might belong to different usages, i.e. the change might affect different sub-graphs. Two disconnected sub-graphs are considered belonging to different usages, since if they had dependency, they would have been connected. Therefore, the impact of a change is modeled as a set of connected sub-graphs.

Definition 12 (Impact Usage) *Impact usage of a change to a code peer is the set of connected sub-graphs of the changed nodes in the usage model of that code peer.*

Since code peers have similar object usages, if their object usages are changed in the similar ways, i.e. having similar impacts on the corresponding GROUMs, we could consider such changes as recurring.

Definition 13 (Recurring Changes) *Two changes are recurring, if their impact usages are sufficiently similar.*

For example, in Figure 3.1 and Figure 3.3, the changes are recurring since their respective impact usages are identical (as in Figure 3.2) or peer-isomorphic (as in Figure 3.4).

3.3 Recommending Recurring Bug Fixes

In this section, we will discuss three algorithms to (1) identify code peers, (2) recognize recurring fixes made to them, and (3) derive the recommended recurring fixing changes for a code peer from one of its peers. Algorithms (1) and (3) are used in the recommendation task while algorithm (2) is needed to record/recognize the recurring fixes in time for current recommendation. It also helps in verifying and improving the accuracy of code peer identification, which in turn improves the recommendation accuracy.

3.3.1 Detecting code peers

If we use pair-wise comparison between all methods to identify all code peers as in its definition in Section 3.2, the computational cost could be expensive because:

1. In large systems, the number of methods could be tens of thousands, which makes pair-wise comparison expensive.

2. Finding maximal peer-isomorphic subgraphs to calculate the similarity between two usage sets is hard. Finding maximal isomorphic subgraphs is already an NP-hard problem.

3. There is a possibility that the computation of peer-isomorphic subgraphs would result in an infinite loop due to the recursive nature of the definition of code peers. In the example of Figure 3.5, to calculate the similarity of the external usage of TableController.<init> and TreeController.<init> in ThreePaneMailFrameController

(see Figure 3.6), we need to check the peer relation of TableSelectionHandler.<init > and TreeSelectionHandler.<init>. However, both internal and external usages of two XXXSelectionHandler's constructors use two XXXController's constructors, respectively. Thus, the peer checking for two XXXSelectionHandler's constructors requires the peer checking for two XXXController ones. This recursive checking could cause infinite computation.

Due to those reasons, we design an heuristic algorithm for code peer identification using the following ideas:

1. Instead of pair-wise comparison for all methods, we use a heuristic to identify the candidates for code peers. We check the peer relation for only the methods/classes that 1) are similar in their code structure or names, *or* 2) share the same ancestor method/class or implement the same interface(s) (i.e. promising the same set of functions), *or* 3) belong to the classes that have other code peers or recurring fixes.

2. Graph-based usage similarity is computed approximately. Instead of finding maximal peer-isomorphic subgraphs of two usages to calculate their similarity, FixWizard extracts from them characteristic features (see Section 3.3.1.1). If such features are similar, the corresponding usages are considered to be similar.

3. To avoid the possibility of recursive calculation of the peer relation, FixWizard iteratively calculates the usage similarity of candidates using already-identified peers. When any candidates are identified as peers, they will be used to update the usage similarity of the rest of candidates.

3.3.1.1 Usage Feature Similarity

In GrouMiner presented in Chapter 2, we have used structural features to compare the similarity of GROUMs which are labeled, directed, and acyclic graphs. We extend that technique to support peer-related similarity.

Definition 14 *A feature, extracted from a path of a GROUM, is the sequence of labels represented by the nodes along that path.*

For example, in Figure 3.4, the extracted features could be [List.isEmpty] (size 1), [List.isEmpty]-[List.get] (size 2), [List.isEmpty]-[List.get]-[UMLOperationsListModel.addElement] (size 3), etc. Such features could describe an object usage approximately, such as the method invocations, their usage orders, and the interactions between objects (by sequences of method calls) in the usage.

Definition 15 (Similar Feature) *Two features* $x = x_1 - x_2 - ... - x_n$ *and* $y = y_1 - y_2 - ... - y_n$ *are considered similar, denoted by* $x \approx y$*, if* $x_i \equiv y_i$ *for all* i*.*

For example, if addElement methods of UMLOperationsListModel and UMLAttributesListModel are peers, then two features [List.isEmpty]-[List.get]-[UMLOperationsListModel.addElement] and [List.isEmpty]-[List.get]-[UMLAttributesListModel.addElement] are considered similar. The similarity of two feature sets is defined using the following definition:

Definition 16 (Similarity of Two Feature Sets) *The similarity of two feature sets* X *and* Y*, denoted by* $fsim(X, Y)$*, is the ratio between the size of their maximum matching based on similar feature relation and their average size.*

This could be written formally as

$$fsim(X, Y) = \frac{max_F|F|}{(|X| + |Y|)/2}$$

for all $F = \{(x, y)|x \in X, y \in Y, x \approx y\}$ such that $\forall(x, y), (x', y') \in F : x = x' \Leftrightarrow y = y'$.

The similarity of two GROUMs is measured by the similarity of its two feature sets, i.e. function $fsim$ is used, instead of sim in the calculation of function Sim in Definition 11.

```
function IdentifyCodePeer(P)                                                          1
  M.add(SimilarStructure(P)) //find clones as candidates                              2
  C.add(SimilarClass(P)) //find similar classes and                                   3
  C.add(SimilarFixedClass(P)) //classes have recurring fixes                          4
  M.add(SimilarNamedMethod(C)) //match methods as candidates                          5
                                                                                      6
  for each pair (A.m,B.n) ∈ M //process candidates list                               7
    if Sim(UI(A.m), UI(B.n)) ≥ σ1 or Sim(UE(A.m), UE(B.n)) ≥ σ2 //similar enough      8
      M.remove((A.m,B.n)), Pm.add((A.m,B.n)) // peer                                  9
      C.add((A, B)) //check enclosing classes                                         10
      M.add(SimilarNamedMethod((A, B))) // for new candidates                        11
                                                                                      12
  Pc.add(PeerClass(C)) //find peer classes                                            13
                                                                                      14
return Pm, Pc                                                                         15
```

Figure 3.7 Algorithm for detecting code peers

3.3.1.2 Code Peer Detection

Figure 3.7 shows the algorithm for detecting code peers. As any time, we have the lists of identified code peers and candidates: Pc and C for classes and Pm and M for methods. Each element of such list is a pair of classes or methods. The algorithm works by iteratively updating the elements of those lists. (It runs incrementally for each revision, i.e. whenever new code is added).

Step 1. Finding candidates. First, structural clones are detected by our incremental clone detection algorithm [55] (line 2, function SimilarStructure). The pairs of cloned methods are added to the candidate list M. Candidates are also scanned from classes that have similar interface, or inheritance, or names (line 3, function SimilarClass), or used to have recurring fixes reported from the previous revisions (line 4, function SimilarFixedClass). In function SimilarClass, for each class, the extracted features include its name, its parent name, the interface(s) it implements, the names of its methods/fields. Then, the classes are compared pair-wise to find the ones having similar features. We compare classes/methods' names as follows. First, we separate a name into words. For example, UMLOperationsListModel will be separated into UML, Operations, List, and Model. The

similarity of two names, as two sequences of words, are calculated based on their largest common subsequence. For example, UML-Operations-List-Model and UML-Attributes-List-Model will be matched respectively. Their largest common subsequence has size of 3. Thus, the overall similarity is (3+3)/(4+4)=0.75.

For each pair of candidate classes in C, their methods are matched based on the similarity of their names (function SimilarNamedMethod), and are added to the list M.

Step 2. Evaluating candidates. Candidates (pairs of methods) in M are stored as a descending sorted list based on their current usage similarity (either of internal usage or external usage, whichever higher). Such usage similarity is calculated via features (i.e. using $fsim$ in Definition 16), and the features are compared using identified peers in Pm only. That is, features having the names of the methods that are not determined as peers yet will not be considered as similar to any other feature (Definition 15).

Each pair of candidates having usage feature similarity (internal or external) larger than chosen thresholds (line 8) will be moved from M to Pm (line 9). Their corresponding classes are then considered as candidate classes (line 10). Other methods are matched (function SimilarNamedMethod) to get new candidates for adding into M (line 11). This step runs until all candidates are evaluated and no new peers are added. After all peer methods are identified, the candidate classes are evaluated to find peer classes (line 13). Finally, identified peer classes and methods are reported (line 15).

3.3.2 Recognizing and recommending recurring fixes

3.3.2.1 Recognizing recurring fixes to code peers

Figure 3.8 shows the algorithm to recognize recurring fixes, which first extracts the impact usages of all changes and then compare them to determine recurring changes.

Step 1. Extracting Impact Usages. In FixWizard, code is represented as ASTs and GROUMs are derived from such trees. Therefore, for each change, to find the changed nodes in the GROUM, instead of comparing the corresponding GROUMs of

```
function RecognizeRecurringChange(Changes)                                    1
  for each Δ ∈ Changes                                                        2
    IU(Δ) = ImpactUsage(Δ) //extract impact usage                             3
  for each pair of changes Δ, Δ' //pair−wise                                  4
    if Sim(IU(Δ), IU(Δ') ≥ σ3) //if having enough similar impact              5
      Report((Δ, Δ')) //report as recurring changes                           6
```

Figure 3.8 Algorithm for recognizing recurring bug fixes

two versions, FixWizard first finds the changes in two ASTs of two versions. It uses Treed [55] to detect all AST node-level tree edit operations, i.e. *insert*, *delete*, *update* (relabel), and *move* an AST node. Because FixWizard keeps the mappings between each AST node and the corresponding GROUM node (if any), it is able to determine the changed nodes in the GROUM from the script returned by Treed. From those nodes, it traverses the GROUM to find edge-changed nodes and connects all the changed nodes into connected GROUMs to have the impact usage of the change.

Let us illustrate this via the example in Figure 3.3. First, from Treed, FixWizard knows that the AST node of type Method Invocation _operations.isEmpty() is added. Thus, the corresponding node List.isEmpty in the GROUM is determined as added. This addition also adds new edges from List.isEmpty to three other nodes (List.get, UMLOperations-ListModel.addElement, and MClassifier.setFeature) due to changes in usage orders (Figure 3.4). From List.isEmpty, FixWizard traverses through such edges and detect those three edge-changed nodes. Then, the GROUM of all four nodes are extracted as the impact usage of the change.

Step 2. Clustering. After impact usages of every change are extracted, FixWizard compares pair-wise those impact usages for each pair of changes, in order to find the ones having similar sets (lines 4-6). Since the number of changes at each revision is small comparing to the number of methods, a pair-wise comparison is still affordable. Similar to the peer detection algorithm, this comparison uses the usage feature similarity, i.e. function $fsim$, in the calculation of the total similarity Sim between the impact usages.

```
function RecommendRecurringChange(X, ΔX)                                    1
 for each Y ∈ PeerOf(X) //for each peer of X                                2
   X* = Affect(X, ΔX) //detect affected subtrees of X                       3
   M = Map(X*, Y)  //map them and other code elements to Y                  4
   for each mapped pair (x, y) ∈ M //for the mapped elements                5
     O = DeriveOperation(x,y) //derive the relevant operation               6
     Recommend(O) //to recommend                                            7
```

Figure 3.9 Algorithm for recommending recurring bug fixes

3.3.2.2 Recommending recurring fixes to code peers

Since recurring changes are modeled by the impact usages, the recommendation should also be represented as the change operations to the GROUMs (i.e. insert, delete, re-label nodes in GROUMs, etc). However, to make the fixing changes in a more readable and instructive manner to developers, FixWizard recommends code changes via change operations at the *syntactic level*.

Figure 3.9 shows the recommendation algorithm. When a code unit X is changed, the algorithm will derive the recurring changes for all code peers Y of X (X might have several peers). First, FixWizard determines the impact usage of the change to X (as in Section 3.3.2.1). Via the mappings between the GROUM's and AST nodes, it identifies the changed sub-trees in AST involved in the change in object usage of X (line 3). Then, those sub-trees are mapped into the corresponding sub-trees in Y (line 4). Each pair of sub-trees is mapped based on their structural similarity and usage similarity of the sub-GROUMs extracted from those two sub-trees. Then, we use the mapped sub-trees to map other code elements, such as fields, variables, types, method invocations. Finally, for each mapped node x of X, if it is affected by a tree edit operation, FixWizard will suggest the corresponding operation to its mapped element y of Y (lines 5-7).

Let us revisit Figure 3.3. Assume that the top method changes, and we need to recommend for the bottom. First, FixWizard determines the change of the expression containing _operations.isEmpty(). It knows that the corresponding GROUM node List.isEmpty

is added. It could determine the statement containing MClassifier.setFeature and the if statement as the involved sub-trees. It then maps those subtrees to the corresponding ones of the bottom method. From the mapped expressions, FixWizard is able to map two variables _operations and _attributes. Thus, the addition of _operations.isEmpty() is used to derive the addition of _attributes.isEmpty(). Other added nodes of the expressions are the same.

Similarly, for the example in Figure 3.5, FixWizard is able derive the correct addition of the statement. However, since it could not map two types TableSelectionController and TreeSelectionController, the recommendation for the bottom method contain the incorrect class name TableSelectionController. Therefore, in the current implementation, FixWizard stops at the recommendation of the locations of code peers at method and statement levels, and the first change operation. From there, developers could be able to complete the changes by consulting the identified recurring bug fix.

3.4 Empirical Evaluation

We have implemented those three algorithms in a tool named FixWizard. The tool has two key functions. The first one is detecting the recurring fixes over time and space. FixWizard could operate in two modes. It could analyze the history of bug fixes in a period and discover the recurring fixes. It could also recognize the current bug fixing change as a recurring one with the others in the past, and then provide recommendations. This is the second key function of FixWizard. When a developer makes a change to fix a bug, FixWizard could recommend other locations (*class, method, statement*) in the code that might also require similar changes. Based on previous recurring fixes, FixWizard could recommend the change operations such as the deletion/insertion/modification of methods or even statements and expressions.

We also performed an empirical study to evaluate FixWizard regarding those two

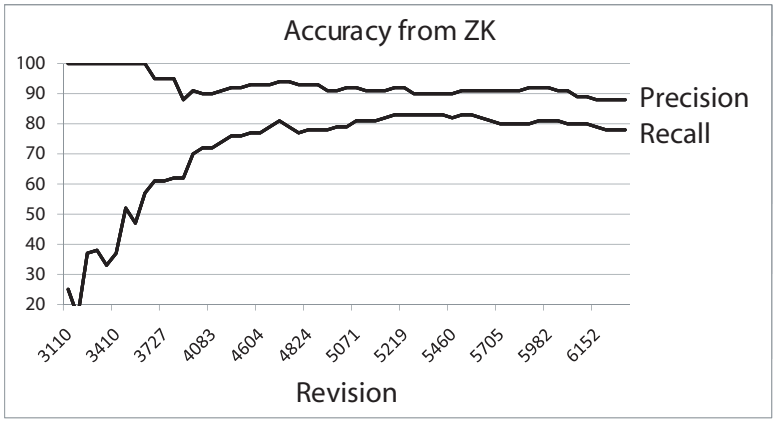

Figure 3.10 Recognition accuracy on ZK

functions. In the evaluation, we set up FixWizard to work with the feature size between 1–4 and all similarity thresholds of 0.75. The experiments were carried out in a computer with Windows XP, Intel Core 2 Duo 2Ghz, 3GB RAM.

3.4.1 Recurring fixes recognition

To evaluate the detection of recurring fixes in FixWizard, we execute it on the subject systems whose recurring fixes were manually verified as described in Section 3.1. Two metrics for performance evaluation are precision and recall. The **precision** value is defined as the number of correctly detected recurring fixing changes over the total number of detected ones. The **recall** value is defined as the number of correctly detected recurring fixing changes over the total number of recurring fixing ones.

For each system, we selected the range of fixing revisions r_1 to r_n as exactly as the one in the experiment in Section 3.1. We incrementally executed FixWizard for each fixing revision r_k in the range from r_1 to r_n. At each r_k, based on the recognized recurring fixes and detected code peers in the past from r_1 to r_{k-1}, FixWizard examines the current fix and detect if those fixing changes are recurring. Note that all fixing changes to the same method at a fixing revision are considered as an atomic fixing change. We compared the detected result with the human-verified data. The *accumulated* precision and recall

Table 3.3 Detection accuracy of recurring bug fixes and running time

System	Class	Method	Detected	Precision	Recall	Time
ArgoUML	1,063	2,318	390	65%	70%	310 sec
Columba	1,161	829	377	87%	73%	95 sec
ZK	295	490	188	91%	80%	110 sec
FlashRecruit	665	1,007	244	84%	75%	120 sec
Eclipse	672	1,126	215	78%	70%	176 sec

values are recorded at each fixing revisions. An accumulated value at r_k refers to the value measured for all fixing revisions from r_1 to r_k. The accumulated values reflect better the accuracy of FixWizard over time than instant precision/recall at a single revision.

Figure 3.10 shows the accumulated precision and recall values, respectively, for the subject system ZK in the fixing revisions. For example, among 124 fixing revisions in ZK project, we could see that after the initial phase of about 6–8 revisions, the accumulated precision and recall reach the ranges between 87–92% and 78–83%, respectively. In the initial phase, because the number of accumulated recurring fixes is still small, precision and recall are affected much by a couple of miss or incorrectly detected recurring fixes. The similar graph results are also achieved for other systems in which after the initial phases the average precision and recall values are 81% and 74%, respectively.

The values are quite consistent and stable after the initial phase for all projects. Moreover, comparing with the percentage of the recurring fixes that occur in code peers (see the empirical study in Section 3.1), we could see that FixWizard is able to detect the majority of such recurring fixes (74% vs 88% on average). This shows that our approximated algorithm for code peer detection is quite accurate. In addition, Table 3.3 shows that FixWizard are also very efficient in term of running time (column Time). In ArgoUML, a large subject system, it processed more than 2,300 fixed methods in about 5 minutes. In brief, our model and algorithms for recurring fixes as the changes to code peers are quite accurate and efficient in object-oriented programs.

Table 3.4 Recommendation accuracy for recurring bug fixes

System	Checked	Recommended	Correct	Precision	Recall
ArgoUML	283	515	217	42%	77%
Columba	199	293	139	47%	70%
Eclipse	152	206	127	62%	84%
FlashRecuit	65	77	39	51%	60%
ZK	69	103	44	43%	64%

3.4.2 Recurring fixes recommendation

This section describes our evaluation for the accuracy of the recommendation algorithm. We used the same set of subject systems as in the previous experiment. We also performed a similar process as in the detection experiment in which FixWizard was incrementally executed for each fixing revision the chosen range from r_1 to r_n. At each fixing revision r_k, we executed FixWizard for each fixing change and recorded the recommendations for its code peers. Then, we compared with the actual fixes. The number of produced recommendations and the number of correct ones are counted. A recommendation is considered as correct if it correctly suggests the fixing location and the first change operation. We recorded the correctness of recommended locations at both method and statement levels (e.g. add/delete methods, add/delete/modify statements). Only the first operation is considered because if the correct location is suggested, it would already save much effort for developers.

Because the human-verified oracle (Section 3.1) did not contain the detailed change operations, we had to check the recommendation results manually. Therefore, we chose a smaller range of fixing revisions to check. Table 3.4 shows the recommendation results. Column Checked is the number of recurring fixes we manually checked, which is less than the total number of recurring fixes in the subject systems. Columns Recommended and Correct are the numbers of changes that FixWizard recommended and changes that we considered to be correct. Two columns Precision and Recall are the precision and recall values, respectively. It could be seen that, on average for all subject systems, FixWizard

has 71% recall and 49% precision. This result shows that using code peers to suggest the locations for the recurring fixes are acceptably accurate. The mapping task between the nodes in code peers and the detected editing operations need to improve.

3.4.3 Discussion

The empirical evaluation result shows that the philosophy of "using bad usages to detect other bad usages" works pretty well. The false positive rate has gone down to around 50%, i.e. a half of the cases reported as having bugs actually have bugs. However, in our current approach, the detection of recurring bugs is based on a broad, method-level context of the bugs, that is, if a method is sufficiently similar to (i.e. is a code peer of) another method with detected bugs, it is likely to have the same bugs. This suggests that, if we add more features specific to the detected bugs, the precision would be improved more.

We have followed that idea in a related work focusing on detecting recurring software vulnerabilities [59]. In this work, we adapted GROUM for C/C++ languages by adding more node types for representing data structures (e.g. `struct`), literals, and operators. For a known bug, we create a GROUM-based signature represents the misused APIs and related code elements (e.g. operators, variables, etc). Then this signature is matches against GROUMs of code fragments in other systems of interest. If an API usage in those GROUMs matches closely to the bug signature, it will be reported as a potential recurring bug. To reduce the odds of false alarms, we also compare the suspicious usage against the GROUM-based signature of the fixed code and only report it when it is more similar to the buggy code than the fixed code.

In our empirical evaluation, we found that the precision can be as high as 90%. This suggests that detecting recurring bugs can be highly effective using graph-based signatures and graph matching. Improving the accuracy of control and data analysis in creating bug signatures and representing code could improve the precision even higher.

CHAPTER 4. STATISTICAL MODELING OF CODE

As shown in previous chapters, since GROUM abstract source code via object usages, GROUM-based techniques works very well for API usage patterns and bugs related to API usages, which often involve object usages (e.g. method calls). However, in source code, there are programming idioms involving other kinds of code elements such as a general purpose loop `for (int i = 0; i < n; i++)` or a printing statement `System.out.println(...)`. While an API usage pattern could contain code elements scattering in a code fragment, those programming idioms are often more local, i.e. they contain sequentially occurring code elements and many of them might not involve object usages.

In this chapter, we developed another model to capture and utilize those programming patterns. Our work is inspired by the similarity between natural languages and programming languages. That is, people use natural languages to express ideas and communicate. In written text, language idioms and regularities, such as *"I love you"* or *"I have a baby"*, occur frequently. Studies in natural language processing show that language idioms and regularities could be captured well by statistical language models, and be used for several purposes, e.g. spelling checking, suggesting words, searching documents, etc [44]. People also use programming languages for to express ideas and communicate. In written code, programming patterns and idioms appear frequently. Thus, could those programming patterns be captured by statistical language models?

A recent study by Hindle *et al.* suggests a positive answer to that question [26]. Using n-gram, a simple language model, to analyze written code, they found that code also has "naturalness". That is, similar to written text, code also has a high level of

repetitiveness and predictability (even higher than in text, as programming languages are more regular than natural languages). Thus, programming patterns are captured well by the statistical language models and can be used for code completion.

However, in this state-of-the-art work, the authors consider code as text, thus use only lexical information in the modeling process. This limits the predictive power and the abstraction of programming patterns captured by their lexical model. In this chapter, we present a new model called *Statistical Semantic Languge Model for Code* (SLAMC). SLAMC is specially designed for source code and uses program semantic information (e.g. data types, scope and dependency) and additional code-based factors like technical concerns and pair-wise associations of code elements in the modeling process. The empirical evaluation shows that SLAMC outperforms the lexical model with an absolute improvement in accuracy from 10–25%. Both models and the evaluation results will be discussed in full details in the remaining of the chapter.

4.1 Background

In this section, we introduce the general concepts of statistical language modeling and n-gram model, a simple yet popular statistical language model for natural languages. Then, we discuss how n-gram model has been adapted into a lexical model for source code. We also discuss the limitation of this lexical model and propose several ideas to address such limitation.

4.1.1 Statistical language modeling

In statistical language modeling, a language is considered as an (infinite) collection of sequences, each is made of elements of its vocabulary. Those sequences represent linguistic units of the language such as words, phrases, sentences, paragraphs, and documents [44]. A statistical language model assigns occurring probabilities for those se-

quences. Language idioms and regularities are sequences with high probabilities.

When the context (i.e. language, vocabulary, and model) is clear, we use the notation $P(s)$ to denote the occurring probability of sequence s assigned by the language model. If $s = s_1 s_2 ... s_m$, by the Bayes rule, we have:

$$P(s) = P(s_1)P(s_2|s_1)...P(s_m|s_1 s_2 ... s_{m-1})$$

That means, the occurring probability of a sequence is computed based on occurring probabilities of its elements, given the previously occurring ones. A conditional probability $P(c|p)$ specifies how likely an element c would occur right after a sequence p. The core task of a language model is to estimate such conditional probabilities. Once trained on a corpus (containing a huge amount of textual data), a language model could be used for prediction. Assume that the editing text is a sequence p, we could use a language model to compute $P(c|p)$ for any possible word c. Then, $P(c|p)$ is the probability that c will appear right after p.

An n-gram model is a language model in which $P(c|p)$ is approximated by $P(c|l)$, where l is the last $n-1$ words of p. In other words, it assumes that the occurring probability of a word depends only on its *local context*, i.e. a window of $n-1$ previously occurring words. The conditional probability $P(c|l)$ is estimated from training data as:

$$P(c|l) \simeq \frac{\text{count}(lc) + \epsilon}{\text{count}(l) + V\epsilon}$$

In this formula, count(lc) and count(l) are numbers of occurrences (i.e. frequencies) of sequences lc (of n words) and l (of $n-1$ words) in training data, respectively. V is the size of the vocabulary and ϵ is a small, positive constant used for smoothing purpose. That is, if lc is unseen in training data (i.e. count$(lc) = 0$), $P(c|l)$ will be assigned a small probability, rather than zero. As seen, an n-gram model captures language regularities as sequences of n words. A sequence of n words is called an n-gram, hence the name of this model. When n is fixed at 1, 2, or 3, the model is called unigram, bigram, or trigram, respectively.

4.1.2 Lexical model for source code

To apply a statistical language model to source code, one first needs to define the basic units of code. In their paper, Hindle *et al.* choose tokens as basic units of code and represent code as sequences of such tokens [26].

Definition 17 (Code token and sequence) *A code token is a unit isolated from the textual representation of source code based on the programming language specification. A code sequence is a sequence of consecutive code tokens.*

Definition 18 (Lexeme) *The lexeme of a token is a sequence of characters representing its lexical value.*

Code tokens and lexemes could be recognized by lexical analysis. For example, given a statement 'len = str.length();' as a as a sequence of characters, a Java lexical analyzer (lexer) will breaking it into eight tokens with corresponding lexemes and categories. For example, the first token is an Identifier with a lexeme of 'len'. The next one is a Punctuation with a lexeme of '=', and so on. It should be noted that lexical analysis provides no semantic information. For example, str is not recognized as a String variable, and length is not recognized as a method in class String.

In [26], Hindle *et al.* adapt n-gram model to analyze written code. We call their model the *lexical model* for source code, since it represents code as sequences of lexemes and captures regularities of code as lexical n-grams.

Definition 19 (Lexical n-gram) *The sequence of lexemes of n consecutive code tokens is a lexical n-gram.*

In the previous code example, [len] [=] [str] and [str] [.] [length] are two lexical 3-grams (we wrap lexemes with brackets, like [len] or [str], to separate them from the others in the sequence). Similar to the n-gram model for natural languages, the lexical model for

Table 4.1 Adding semantic information

Lexeme	Purpose	Adding semantic information
x.next	If x is a LinkedList, access field next	VAR[LinkedList] FIELD[LinkedList, next]
x.next	If x is a Scanner, call method next	VAR[Scanner] CALL[Scanner, next]
str.length()	Get string's length	VAR[String] CALL[String, length]
s.length()	Get string's length	VAR[String] CALL[String, length]

source code also predicts a next code token based on its local context. $P(c_n|c_1c_2...c_{n-1})$ specifies how likely a code token c_n will occur given $n-1$ previously occurring tokens of $c_1c_2...c_{n-1}$. This conditional probability is also estimated from a training code corpus:

$$P(c_n|c_1c_2...c_{n-1}) \simeq \frac{\text{count}(\text{lex}(c_1c_2...c_{n-1}c_n)) + \epsilon}{\text{count}(\text{lex}(c_1c_2...c_{n-1})) + V\epsilon}$$

In this formula, function lex produces the lexical n-gram of a code sequence.

4.1.3 Discussions

The lexical model has been shown to capture well code regularities and programming patterns at the lexical level to support code suggestion and completion [26]. However, because of using only lexical information in its modeling process, this model might be have both false positives and negatives in capturing programming patterns. Table 4.1 shows two examples of such inaccuracies. As seen, the lexical model will consider two sequences s.next and s.next as occurrences of the same pattern (because they have identical lexemes). However, the same lexemes can refer to different code elements. If token s in the first sequence refers to a LinkedList variable, and a Scanner variable in the second sequence, two sequences actually have different purposes. In contrast, developers might use the same pattern, but name variables differently. For example, two sequences str.length() and s.length() have the same purpose of *"getting a string's length"*. However, due to the differences of lexemes (e.g. str and s), the lexical model will fails to recognize them as occurrences of the same pattern.

4.1.3.1 Adding semantic information

If we recognize semantic information of code tokens, such as programming roles (e.g. variables, fields, method calls) and data types, we could address those inaccuracies. For example, as seen in Table 4.1, in the latter example, str and s could be recognized as two String variables and .length() is a call to method length of class String. Then, if we annotate the corresponding sequences with such information, their differences of lexemes will be resolved and both now look the same. Similarly, in the former example, when data types and programming roles of the tokens are annotated, the two sequences will look different, thus, will not be wrongly recognized as the same pattern. This suggests that, adding semantic information to code tokens, a language model would capture better the patterns at higher abstraction levels, thus, could provide better code suggestion.

4.1.3.2 Adding other code-related factors

Similar to n-gram models for natural languages, the lexical model for code uses only *local context* in modeling and prediction. However, other factors in source code could have influences and add more predictive power. One strong factor is *topic*. A software system often has several system-wise, global technical concerns, performing different functionality, such as *file I/O*, *database*, *networking*, *graphics*, etc. A technical concern (also called *topic*) often involves several APIs. For example, *file I/O* is usually implemented with Scanner, BufferedReader, or PrintWriter, while *database* involves Statement, ResultSet, or Connection more often. Therefore, topic information could add predictive power to local context. Table 4.2 shows an example in which the local context is an incomplete loop while (. Since both Scanner and ResultSet are often used with while loops, it is difficult to predict which is more likely. However, if the topic is *file I/O*, the next code token will be more likely a Scanner variable. In contrast, if the topic is *database*, a ResultSet variable would be likely the next token.

Another factor is the *pair-wise association* of code elements. In source code, due

Table 4.2 Adding topic information

Local context	Topic	Prediction of next token
while (	File I/O	a Scanner variable is more likely than a ResultSet
while (	Database	a ResultSet variable is more likely than a Scanner

to the syntax specification of the programming language or the usage specification of the APIs in libraries, some program elements often go together, such as lock/unlock, fopen/fclose, or try/catch. Thus, the occurrence of one token would likely suggest that of the other. However, associated tokens often locate far apart (e.g. there would be many code tokens in-between the pair fopen and fclose). Thus, their association could not be captured within their local context using n-grams.

Scope and *dependency* information is also important in modeling. For example, due to the modularity in system design, a source file often involves few technical concerns. Thus, different files will have different topics, i.e. a file should have its own topic(s). An associated pair like fopen and fclose often has control and data dependencies. That is, fopen is called before fclose, and they operate in the same FILE variable. Dependencies also occur in local context. For example, in sequence buf.append(str), str and buf has a data dependency, as str is a parameter for a method call of buf. This suggests that, using scope and dependency information would improve the predictive power of other factors.

4.2 Semantic Language Model

Following the discussed ideas to address the limitation of the lexical model, we have developed SLAMC, a statistical Semantic LAnguage Model designed for source Code. SLAMC annotates code tokens with *sememes*, the encoding of their semantic information, and captures language regularities and programming patterns as sememe sequences. It also combines local context, topic, pair-wise association, and scope/dependencies of tokens in the modeling process. This section discusses its design and implementation.

4.2.1 Design strategies

Let us first explain our design strategies in selecting the kinds of semantic information to be incorporated into sememes (will be formally defined later). First, the programming **role** of a token in a program with respect to the programming language, i.e. whether it represents a variable, data type, operator, function call, field, keyword, etc, is important. This information helps distinguish tokens having the same lexemes, e.g. local variables vs fields. In addition, we construct sememes of code tokens based on their roles.

Local variables and parameters. In a program, we can change names of local variables and parameters easily without affecting the program semantics. However, changing their data types would cause problems. This suggests that, for those tokens, **data type** information is more abstract and important. Therefore, we annotate a local variable or parameter with its data type. For example, token str is annotated as VAR[String].

Literals. A literal in a program can have an arbitrary value and the number of possible values are huge (even infinite, such as for String literals). In addition, an arbitrary value is unlikely to recur (if a value is reused frequently, developers would define it as a constant). Thus, it would be ineffective to predict a literal value, and is inefficient to capture all literal values encountered in code. Therefore, we annotate a literal with its data type only. For example, literal "Hello world" is annotated as LIT[String].

Special literals. There are some special literals, like 0, true, null and "", often recur and associate with some programming patterns, such as checking null: if (x != null), checking empty: if (map.size() == 0), if (str != ""), infinite loop: while (true), etc. Thus, we use special annotations for those literals. For example, "" is annotated as EMPTYSTRING.

Fields. Unlike local variables and parameters, field names are important because they could be referred outside their defining classes. Since different classes might have fields with the same name, we annotate a field with both its name and the name of its class. For example, the field out of the class System is annotated by FIELD[System, out].

Table 4.3 Construction rules of sematic annotation

Role	Sememe	Example
Local variable	Role, data type	str → VAR[String]
Parameter	Role, data type	index → PARA[Integer]
Literal	Role, data type	"Hello world" → LIT[String]
Special literal	*Reserved sememe*	"" → EMPTYSTRING, null → NULL
Field	Role, class, name	next → FIELD[LinkedList, next]
Method	Role, signature	indexOf → CALL[String,indexOf,[String],Integer]
Operator	Role, name	= → OP[assign], (Integer) → CAST[Integer]
Data type	Role, name	String → TYPE[String]
Keyword	*Reserved sememe*	if → IF, while → WHILE
Code separator	*Reserved sememe*	} (of a for loop) → FOREND
Unknown	*Lexeme*	x → LEX[x]

Methods. A method is identified via its signature, including its name, its class's name, the return type, and the parameter list. Therefore, we annotate a method with all those components. For example, a call to the method indexOf of the class String is annotated by CALL[String, indexOf, [String], Integer].

4.2.2 Sememe

Using those design strategies, in this section, we define and present the rule to construct sememes, the semantic information annotations of code tokens.

Definition 20 (Sememe) *The sememe of a code token is a structured annotation representing its semantic information, such as its programming role and data type.*

Table 4.3 lists the construction rules of sememes for code tokens of popular programming roles. For example, indexOf in sequence str.indexOf("Good") has the programming role of a function call, thus, its sememe consists of the role annotation CALL[ ], its class name String, its name length, its parameter list [String], and its returned type Integer. It should be noted that, in practice, sometimes the data types of parameters of a method call are not resolvable. In such case, the method call is annotated with its number of passing parameters, rather than its parameter list. In addition, for the sake of brevity, in the

remaining text, we write sememes of method calls without return types and parameters, such as CALL[String,length] or CALL[Scanner,hasNext].

Sometimes, semantic information of some tokens might be unavailable. For example, in the incomplete code under editing "if (x", it might be undecidable whether identifier x is a variable, data type, or a method name, thus leading to no data type or programming role information. In such cases, the token is annotated with the sememe of type LEX and its lexeme, i.e. LEX[x]. Some code tokens, e.g. semicolons and parentheses, are not associated with semantic information, thus are excluded in the modeling process.

Once individual tokens are annotated with sememes, SLAMC extracts sememe sequences from code sequences and capture programming patterns as sememe sequences with high occurring probabilities. Similar to the lexical model, SLAMC also uses semantic n-gram to represent local context.

Definition 21 (Semantic n-gram) *A semantic n-gram is a sequence of n sememes extracted from a sequence of consecutive code tokens.*

For example, SLAMC will extract from code sequence "if (node != null)" a semantic 4-gram IF VAR[Node] OP[neq] NULL. It should be noted that, since some punctuations such as parentheses are excluded, a code sequence might have less sememes than lexemes.

4.2.3 N-gram topic model

As seen in previous sections, SLAMC represents source code as sequences of sememes annotating the code tokens. As a statistical language model, the core of SLAMC consists conditional probabilities $P(c|p)$ specifying how likely a code token with sememe c will occur next to a code sequence with the corresponding sememe sequence p. SLAMC computes those conditional probabilities based on several factors, including local context, topic, pair-wise association, and scope/dependencies of tokens. In this section, we discuss how SLAMC incorporates local context and topic in this computation.

Prior research shows that the latent topics recovered by topic modeling on source files correspond well to the technical concerns in a system [4, 51]. Inspired by Wallach [72], we have developed an n-gram topic model that integrates the information of both local contexts (via n-grams) and technical concerns (via topics). The key idea is that the occurring probability of a token c in a sequence depends simultaneously on its topic assignment and $n-1$ previously occurring tokens.

Our model assumes a codebase to have K topics (corresponding to its technical concerns). Since a source file might involve several concerns, SLAMC allows each token of a code sequence to be assigned to one among K topics. Thus, a sequence could involve all K topics with often different proportions (some might be zero). SLAMC represents the topics of a sequence p as a multinomial distribution θ sampled from the Dirichlet distribution $\text{Dir}(\alpha, K)$. θ is called *topic proportion* of p and θ_k is the proportion of topic k in p, which can be estimated as the ratio of the number of tokens assigned to topic k over the total number of tokens of p. For example, a code sequence could have 40% of its tokens about I/O, 50% about string processing, and 10% on GUI.

In our n-gram topic model, the occurring probability of a token c is dependent on its topic assignment k and on its local context l. This dependency is modeled by a multinomial distribution $\phi_{k,l}$ (called *token distribution*), which is a sample of the Dirichlet distribution $\text{Dir}(\beta, V)$. $\phi_{k,l}(c)$ specifies how likely a token with sememe c will occur if it is assigned to topic k and its local context is l, a sememe sequence of $n-1$ sememes.

Then, to use this n-gram topic model in SLAMC, i.e. computing the probability $P(c|p)$ for any given code token c and code sequence p, we first need to train it, i.e. estimating the multinomial distributions $\phi_{k,l}$ for all possible topic k and local context l from a training codebase. We have developed a training algorithm based on Gibbs sampling, which is presented below.

```
function Train(B, α, β, K, N)                                              1
  for each source file f in training codebase B                            2
    extract its sememe sequence s                                          3
    collect available sememes into V                                       4
    randomly initiate its topic proportion θ and topic assignment z        5
  repeat                                                                   6
    for each available topic k and n−gram l                                7
      for each token c ∈ V                                                 8
        φ_{k,l}(c) = (count_{(l,c,k)}+β) / (count_{(l,k)}+KVβ)             9
    for each code sequence s in B                                          10
      θ, z = Estimate(s, φ)                                                11
  until φ is stable (i.e. converges)                                       12
return V, φ                                                                13
                                                                           14
function Estimate(s, φ)                                                    15
  repeat                                                                   16
    for each position i in s                                               17
      sample z_i where P(z_i = k) = θ_k · φ_{k,l_i}(s_i)                    18
    for each topic k                                                       19
      θ_k = (count_{(z_i=k)}+α) / (length_{(s)}+Kα)                        20
  until θ is stable                                                        21
return θ, z                                                                22
```

Figure 4.1 Training algorithm for n-gram topic model

4.2.3.1 Training algorithm

Figure 4.1 illustrates our training algorithm. The input includes a codebase B, containing a collection of source files, and other pre-defined parameters, such as the number of topics K, hyper-parameters of Dirichlet distributions α and β, and the maximal size of n-grams N. The output includes the vocabulary V containing all collected sememes and all distributions $\phi_{k,l}$ for every topic k and every possible n-gram l. In addition, for each source file represented as a sememe sequence s, the output also includes its topic proportion θ and the topic assignment z_i of the token at position i in s.

The training algorithm first parses all source files in the codebase, and builds a sememe sequence for each of them. It collects all sememes into the vocabulary V and randomly initiates all latent variables (e.g. θ, ϕ, z) (lines 2-5). Then, it performs two-phase processing as follows.

Phase 1. SLAMC uses the existing (or randomly initiated in the first iteration) topic assignments of all sequences (variables z) to estimate the distributions $\phi_{k,l}$ for every possible topic k and n-gram l. They are estimated as in line 9:

$$\phi_{k,l}(c) = \frac{\text{count}(l, c, k) + \beta}{\text{count}(l, k) + KV\beta}$$

In this formula, function $\text{count}(l, c, k)$ counts every position i in every sequence s where the sememe at position i is c and is assigned to topic k, and its $n-1$ previous sememes make up the sequence l. Similarly, $\text{count}(l, k)$ counts such positions but does not require $s_i = c$. The positive parameter β is added to all the counts for the smoothing purpose for the computation in later iterations.

Phase 2. SLAMC uses the estimated distributions $\phi k, l$ to estimate the topic proportion θ and topic assignment z for every code sequence s (each for a source file) in the codebase (lines 11, 15-22). First, a topic is sampled and assigned for each position i in s. The probability that topic k is assigned to position i is computed as in line 18:

$$P(z_i = k | s, \theta, \phi) \sim \theta_k \cdot \phi_{k,l_i}(s_i)$$

where s_i is the token of s at position i and l_i is the sequence in s of $n-1$ tokens before i. Once topics are assigned for all positions (i.e. z_i is sampled for every i), the topic proportion θ is re-estimated as line 20:

$$\theta_k \simeq \frac{\text{count}(z_i = k) + \alpha}{\text{length}(s) + K\alpha}$$

That means, SLAMC counts the number of tokens assigned to topic k, and approximately estimates the proportion of topic k by the ratio of the number of tokens assigned to topic k over the length of sequence s. The positive parameter α is added to all the counts for the smoothing purpose.

This sampling and re-estimating process is repeated on each sequence until the topic proportion θ is stable (i.e. converged, line 21). When every sequence in the codebase has a stable topic proportion, the algorithm goes back to phase 1. It stops when the latent variables θ and ϕ are stable or the number of iterations reaches a pre-defined threshold.

4.2.3.2 Representation and storage

To save storage costs and improve running time, SLAMC does not directly store the distributions $\phi_{k,l}$. It instead stores all n-grams and their counts in a tree. Each tree node has a pointer to its parent, a sememe in the vocabulary as its label c, a counting vector φ of size K for the counts, and the total count σ. The root node is an empty node. The path from a node to the root corresponds to an n-gram. Let us use l to denote the n-gram from the parent node b of node c to the root. The value φ_k is equal to $\mathsf{count}(l, c, k)$. $\mathsf{count}(l, k)$ is computed by summing over φ_k in all children nodes of b.

This tree is created when the training algorithm constructs the sememe sequences. When a new sememe c is built, the algorithm extracts all possible n-grams l that end right before c (n varies from 1 to $N-1$). Then, for each n-gram l, it traverses the tree to find the corresponding path. If the last node of that path does not have a child with the label c, such a child is created and its total count σ is assigned with the value of 1. Otherwise, its total count is increased by 1. Then, the tree is updated at the beginning of every phase 1 in the training process. The algorithm processes each token in a sequence in the training set similarly to when it creates the tree. However, if the topic assignment for that token is k, it updates φ_k instead of σ.

4.2.4 Pairwise association

Since pairwise association could not be captured with n-gram, SLAMC use separated conditional probabilities to model this factor. In SLAMC, $P(c|b)$ is the probability that a token with sememe c will occur when a token with a sememe b has occurred previously. This conditional probability is estimated as from training data as:

$$P(c|b) \simeq \frac{\text{count}(c, b)}{\text{count}(b)}$$

In this formula, $\text{count}(c, b)$ is the number of times two code tokens with sememes b and c co-occur in a code unit (e.g. a method), while $\text{count}(b)$ is the total occurrences of

tokens with sememe b in the training data.

4.2.5 Scope and dependency

To improve the accuracy of the modeling process, SLAMC uses scope and dependency information to control the training process. First, to avoid pairs of code elements that co-occur by chance (e.g. do not have semantically coupling relationship), SLAMC counts a pair of tokens only if they have data dependencies. For example, if two function calls fopen and fclose are performed on the same file (i.e. having a data dependency), their co-occurrences are counted. If they operate on different files, their co-occurrences might not be semantically related, thus, is avoided.

To reduce the storage and computational cost, we do not compute and store the probability $P(c|b)$ for any pair c and b (it would be a huge cost to compute and store V^2 of such probabilities for the entire vocabulary). We instead consider only the tokens for control structures (including branching, loop, and exception handling statements) and API entities (including classes, methods, functions, and fields). We also consider only the pairs of tokens within the boundary of a method.

Scope and dependency are used in the same manner for n-gram sequences. That is, we only extract n-grams that contain at least a pair of tokens having dependencies. Those dependencies could be either control dependencies (e.g. from a token indicating a while loop to a token belong to its control expression) or data dependencies (from a token used as the input of a method call to the token referring that method call). In addition, n-grams are also extracted only within method boundaries, because code extracted from the end of one method to the beginning of another would have not meaning.

4.2.6 Predicting with SLAMC

Once trained, SLAMC could be used to predict the most likely next code tokens for a given code sequence p. It first extracts the sememe sequence from p. Then, it

uses function Estimate (Figure 4.1) to estimate the topic proportion θ of p. Finally, it computes the highest conditional probabilities predicted by its n-gram topic model and by pairwise associations:

$$P(c|p) \simeq \max(\max_{n}(\sum_{k} \theta_k.\phi_{k,p_n}(c)), \max_{b \in p} P(c|b))$$

In this formula, p_n is the last n tokens of p, i.e. the local context of c. The formula means that SLAMC chooses the best prediction from all factors: local context, topic, and pairwise association. $P(c|p)$ now specifies how likely a token of sememe c will occur after p, thus, tokens with highest probabilities could be ranked and recommended.

4.3 Code Suggestion

As seen in previous sections, once trained on a codebase, SLAMC captures the programming patterns and code regularieis in that codebase, and thus, is able to predict the next tokens of given code sequences. Based on this ability of next-token prediction, we have built a code suggestion engine, which is discussed in details in this section.

4.3.1 Semantic, context-sensitive code suggestion

Overview. Instead of recommending code following a pre-defined template, our engine suggests a sequence of code tokens that is *best-fit to the context* of the editing code and *most likely to appear next*. To help users choose from several relevant suggestions, it provides a ranked list of such sequences. To compute their relevancies, we define a set of suggestion rules that are based on the current context and aim to complete a meaningful code sequence (Table 4.4). The idea is that a useful suggestion would *complete the code at the current position to form a meaningful code unit and likely appear next*. Currently, we implement the rules to define a meaningful code unit in term of a member access, a method call, an infix expression, or a condition expression. For example, if the code context is recognized as an incomplete binary expression such as in "x + ", the suggestions

Table 4.4 Rules of context-sensitive suggestion

Context	Example	Suggestion
Member access	node.	a method or field name, e.g. size or value
Method call	map.get(	a type-compatible expression for next argument, e.g. k
Infix expression	x +	a type-compatible expression for next operand, e.g. y
Condition	if (	a Boolean expression, e.g. x != 0 or !set.isEmpty()
Other	for (int	a next token, e.g. i

Table 4.5 Semantic, context-sensitive completion

	Current code	Suggestions
Lexical tokens	(1) if (node	!= null (4) == null .isRoot()
Semantic tokens	(2) IF VAR[Node]	OP[neq] NULL (3) OP[equal] NULL OP[access] CALL[Node,isRoot]

will be an expression for the remaining operand with a data type compatible with x in the addition. If the context is an incomplete method call, a suggestion will be an expression with a compatible type for the next argument. If it does not match with any pre-defined context, the token with highest probability is suggested.

To illustrate our algorithm, let us consider an example in Table 4.5. Assume that a developer is writing a statement "if (node" and requests a suggestion (see (1)). Our engine first converts the code into a sememe sequence p (see (2)). Analyzing this sequence, our engine recognizes that it matches the rule for an incomplete condition statement. Then, it searches for potential sequences q that connect with the current code to form a boolean expression. Such sequences are ranked based on the score $P(q|p)$. Assume that the search returns a ranked list of three sememe sequences as in (3), which are transformed back to lexical forms and presented to the user as in (4).

Figure 4.2 illustrates our code suggestion algorithm with three main steps. In the first step (lines 2-3), it analyzes the currently editing code and produces its sememe sequence. Since the current code might be incomplete or syntactically incorrect, it uses

```
function Recommend(CurrentCode F, NGramTopicModel φ)                          1
  s = BuildSequence(F) //sequence of semantic code tokens                     2
  θ = EstimateNGramTopic(s, φ) //topic proportion of F                        3
  p = GetCodePriorEditingPoint(s, edpos)                                      4
  L = Search(p, θ)                                                            5
  for each q ∈ L                                                              6
    lex[q] = Unparse(q)                                                       7
  u = UserSelect(lex)                                                         8
  Fillin(u)                                                                   9
                                                                              10
function Search(p, θ)                                                         11
  L = new sorted list of size topSize, Q = new queue                          12
  Q.add(“”, 1) //empty sequence, score = 1                                    13
  repeat                                                                      14
    q = Q.next()                                                              15
    if length(q) ≥ maxDepth then continue                                     16
    C(q) = ExpandableTokens(p, q)                                             17
    for each c ∈ C(q) Q.add(qc)                                               18
    if ContextFit(p, q) then L.add(q, Score(q, p, θ, φ))                      19
  until Q is empty                                                            20
  if L is empty then add the top relevant tokens to L                         21
return L                                                                      22
```

Figure 4.2 Code suggestion algorithm

Partial Program Analysis (PPA) [9] for code analysis and then recognizes the matched code context. PPA parses the code into an AST, which is then analyzed by SLAMC to produce the sememes and other associated semantic information. If PPA cannot parse some tokens, it marks them as Unknown nodes and SLAMC creates the semantic tokens of type LEX for them. Then, SLAMC estimates the topics of the code sequence using n-gram topic model (line 3). In step 2 (lines 4-5, 11-22), it predicts the next code sequences that connect with the current code to form a type-compatible code unit as described in the rule of the matched context. All such sequences are ranked based on their scores using a search-based method. In step 3 (lines 6-9), those sequences are transformed to lexical forms and presented to users for selection and filling up. Let us discuss in details the two steps 2 and 3.

4.3.2 Predicting relevant code

Assume that s is the sememe sequence for the entire source file under editing, and θ is its estimated topic proportion. Since the current editing position edpos might not be at the end of s, the engine starts the search from a sub-sequence p of s, containing all tokens prior to edpos. Then, it searches for sequence(s) $q = c_1c_2...c_t$ with the relevance score of:

$$P(q|p) = P(c_1|p).P(c_2|pc_1)...P(c_t|pc_1c_2...c_{t-1}) \quad (*)$$

This formula suggests that we could expand the sequences token-by-token and compute the score of a newly explored sequence from the previously explored ones. Thus, our engine generates relevant next sequences by searching on a lattice of tokens of which each path is a potential suggestion using a depth-limited strategy. That is, it keeps a queue Q of exploring paths and chooses to expand a path q if it has not reached the pre-defined maximum depth (maxDepth) (lines 15-18). If q satisfies the context rules, its score will be computed and it will be added to the ranked list L of suggested sequences (line 19). If no sequences satisfy the context, the top relevant tokens are added (line 21).

4.3.2.1 Expanding relevant tokens

Theoretically, at each search step, every token should be considered. However, to reduce the search space, we choose only the tokens "expandable" for the current search path q (function ExpandableTokens at line 17). To do that, we use the trained n-gram topic model ϕ to infer *the possible sememes* $V(q)$ *for the next token of* q, and then choose semantic tokens matching those sememes. Assume that the current search path is $q = c_1c_2...c_i$. To find the set of possible sememes $V(q)$ of the next token c, we connect p and q and extract any possible n-grams l ending at c_i (l might have tokens in both p and q). Then, we look for l on the prefix tree of n-grams (see Section 4.2.3.2). If l exists, all sememes of its children nodes are added to $V(q)$.

For each sememe $v \in V(q)$, we create a corresponding code token and put it into the set of expandable tokens $C(q)$. We use the rules in Table 4.3 to infer necessary information, e.g. role or lexeme. For instance, if the sememe is CALL[Node,isRoot], the code token has the role of *function call* and the lexeme of isRoot.

Caching of variables' names. The sememes of variables and literals in n-gram topic model do not have lexemes. Thus, we infer the lexemes for sememes of variables using a caching technique. If v is a sememe for a variable, we select all existing code tokens in the sequence s that have roles of variables. Then, all tokens for variables that belong to the same or enclosing scope of the last code token c_i of the search path and have the same type as specified in the sememe v will be added to $C(q)$. For example, if c_i is within a for and v is a VAR[String], all String variables in the code blocks containing that for loop, including the enclosing method and class, are considered. For a literal sememe, we create a semantic token with the default value for its type (e.g. 0, null).

4.3.2.2 Checking of context fitness

Our engine uses the rules in Table 4.4 to check if a recommended sequence q produced by the above process fits with the context of the current code sequence p (function ContextFit, line 19). For example, from analyzing the current code via PPA to build semantic tokens, our engine knows that the last method call in the current code p has less number of arguments than that of parameters specified in its sememe, the context is then detected as an incomplete method call.

Then, based on the type of context of p, our engine checks if q fits with p as they are connected. If an expression is expected, our engine will check if q is a syntactically correct expression and has the expected type in the context p. If the context is a method call, it will check if q contains the expression that has the correct type of the next parameter for the method in p. If the context is an infix expression, then the result statement of connecting p and q must have the form of $X \diamond Y$, where X and Y are two

valid expressions and have data types compatible with operator $\diamond$. Similar treatment is used for a condition statement in which a boolean expression is expected to be formed. If a context cannot be recognized due to incomplete code, ContextFit returns false.

4.3.2.3 Computing relevance scores

The relevance score of a new path qc is computed incrementally by (*) as $P(qc|p) = R(c).P(q|p)$, in which $R(c)$ is the relevance score of the token c to the current search path. Initially, $R(c)$ is computed as $P(c|pc_1c_2...c_i, \theta)$ using the n-gram topic model ϕ. Since ϕ models only local context and global concern, $R(c)$ is adjusted for other factors. First, if c is a token for a control keyword, or a method call, the maximal pair-wise association probability $P(c|b)$ for every $b \in pc_1c_2...c_i$ is selected for adjusting. Otherwise, if c is a token for a variable, $R(c)$ is adjusted based on the distance r, in term of tokens, from the position of its declaration to the current position. In our current implementation, $R(c)$ is multiplied by $\lambda = 1/log(r + 1)$. That is, the more distant the declaration of a variable, the lower its relevance to the current position.

4.3.3 Transforming to lexical forms

The transformation of a sequence q is done by creating the sequence of lexemes for the tokens in q. This task is straightforward since the lexeme is available in a token. However, our engine also adds the syntactic sugars for correctness (line 7). For instance, in CALL[Node,isRoot], the lexeme is isRoot, and the method call has no argument. Thus, the lexical form ".isRoot()" is created with the added dot and parentheses. Finally, the lexical forms will be suggested in the original ranking.

4.4 Empirical Evaluation

We conducted several experiments to study SLAMC's code suggestion accuracy with various configurations and to compare it with the lexical model. The data set consists

Table 4.6 Subject systems used for the evaluation of SLAMC

Java system	Release time	LOCs	C# system	Release time	LOCs
Ant	01/23/11	254,457	Banshee	01/23/13	166,279
Batik	01/18/11	367,293	CruiseControl	07/25/12	260,741
Cassandra	01/22/11	135,992	db4o	05/22/08	218,481
Log4J	11/19/10	68,528	Lucene.Net	03/08/07	169,413
Lucene	03/19/10	429,957	MediaPortal	01/19/13	922,765
Maven2	11/18/10	61,622	NServiceBus	03/09/12	31,892
Maven3	01/22/11	114,527	OpenRastar	09/28/11	52,018
Xalan-J	12/12/09	349,837	PDF Clown	11/13/11	66,308
Xerces	01/11/11	257,572	RASP Library	01/08/08	62,932

the same Java projects (with the same revisions) as in prior work [26]. We also evaluated on nine C# projects. Table 4.6 lists our subject systems. All experiments were conducted on a computer with AMD Phenom II X4 965 3.0 GHz, 8GB RAM, and Linux Mint.

4.4.1 Experimental procedure

Our experiment is performed by 10-fold cross validation. We first divided the source files of a project into 10 folds (with similar sizes in term of LOCs). Then, each fold was chosen for testing, while the remaining ones were used for training. Suggestion accuracy is measured as follows. For a source file in the test data, our evaluation script uses PPA [9] for partial parsing and semantic analysis and produces a code sequence s. Then the script tests this sequence token-by-token. At position i, it requests the language model under evaluation to suggest the top k most likely code tokens given the previous ones. If the actual token s_i is among those k suggested tokens, we count it as a hit. The top-k suggestion accuracy for a code sequence is the ratio of the number of hits over its length. For example, a code sequence of a test file has 100 tokens and we have 60 hits, the accuracy is 60%. The overall accuracy for a project is computed as the average accuracy over all source files tested in the entire cross-validation process.

Table 4.7 Accuracy (%) with various configurations

Model	Top-1	Top-2	Top-5	Top-10
1. Baseline (Lexical model [26])	53.6	60.6	66.1	68.8
2. Sememe	58.0	65.8	72.7	76.3 (+7.5)
3. Sememe + cache	58.7	66.9	75.7	80.3 (+4.0)
4. Sememe + cache + dependency	58.8	67.0	75.8	80.4 (+0.1)
5. Sememe + cache + dependency + topic	63.0	70.8	77.1	81.8 (+1.4)
6. Sememe + cache + dependency + topic + pair-wise association (SLAMC)	64.0	71.9	78.2	82.3 (+0.5)

4.4.2 Sensitivity analysis on impact of factors

We first evaluated the impact of different factors on code suggestion accuracy on Lucene, our largest Java subject system. Table 4.7 shows accuracy with different combinations of factors. The first row is the baseline, corresponding to the n-gram model on lexeme [26]. The second row shows accuracy of the n-gram model on sememes, i.e. semantic information is added, but only local context is considered. The third model is similar to the second one, however, caching is used to predict variables' names. The forth model is similar to the third one, however, dependencies is incorporated. It extracts only n-grams with dependencies among their tokens. The fifth model adds topic factor by replacing the n-gram model in the forth model by the n-gram topic model. SLAMC is the model in the last row which adds pair-wise association to the fifth model.

As seen in the table, all added factors improve the overall accuracy. For example, comparing the first two rows, we see that, when semantic information is added, the top-10 accuracy increases 7.5%. When caching is added to address the problem of sememes (no variable name), the overall accuracy improves 4% more. When all factors are added, the overall top-10 accuracy of SLAMC is 13.5% higher than the lexical model. Among the factors, sememe adds the most predictive power. In contrast, dependency just adds a slightly improvement of 0.1%.

This analysis suggests that, for practical use, sememe + caching is the most cost-

Table 4.8 Accuracy of code suggestion for Java code

Project	Suggest	Lexical model	SLAMC	Improve	Relative
Ant	Top 1	44.7%	63.5%	18.8%	42.1%
	Top 5	55.4%	79.5%	24.1%	43.5%
Batik	Top 1	44.7%	65.5%	20.8%	46.5%
	Top 5	55.4%	80.7%	25.3%	45.7%
Cassandra	Top 1	44.9%	65.9%	21.0%	46.8%
	Top 5	51.3%	73.5%	22.2%	43.3%
Log4J	Top 1	45.2%	67.4%	18.8%	41.6%
	Top 5	55.5%	79.2%	24.1%	43.4%
Lucene	Top 1	53.6%	64.0%	10.4%	19.5%
	Top 5	66.2%	78.2%	12.0%	18.1%
Maven-2	Top 1	41.3%	64.4%	23.1%	55.9%
	Top 5	51.0%	74.8%	23.8%	46.7%
Maven-3	Top 1	47.7%	65.0%	17.3%	36.3%
	Top 5	59.2%	74.1%	14.9%	25.2%
Xalan	Top 1	48.1%	68.6%	20.5%	42.6%
	Top 5	58.9%	82.4%	23.5%	39.9%
Xerces	Top 1	46.4%	66.6%	20.2%	43.5%
	Top 5	58.1%	81.8%	23.7%	40.8%

effective configuration. Adding dependency would require complex and time-consuming analysis, but the improvement is modest. Topics and pair-wise associations also require huge time and space cost for training and storing additional model parameters. For example, an n-gram topic model for K topics has K times more parameters than a typical n-gram model.

4.4.3 Comparison of semantic and lexical models

Our second experiment was to compare SLAMC with the lexical model in all Java and C# projects. Tables 4.8 and 4.9 show the comparison results. In each table, column Improve shows the absolute improvement of SLAMC in term of overall suggestion accuracy, while column Relative shows the relative improvement. As seen, for Java projects,

Table 4.9 Accuracy of code suggestion for C# code

Project	Suggest	Lexical model	SLAMC	Improve	Relative
Banshee (BS)	Top 1	37.2%	62.5%	25.3%	68.0%
	Top 5	47.8%	72.7%	24.9%	52.1%
Cruise Control (CC)	Top 1	42.8%	64.8%	22.0%	51.4%
	Top 5	54.4%	74.2%	19.8%	36.4%
db4o (DB)	Top 1	44.8%	65.0%	20.2%	45.1%
	Top 5	57.5%	77.3%	19.8%	34.4%
Lucene. Net (LN)	Top 1	47.0%	69.0%	22.0%	46.8%
	Top 5	58.6%	82.0%	23.4%	39.9%
Media Portal (MP)	Top 1	47.1%	66.7%	19.6%	41.6%
	Top 5	58.0%	79.4%	21.4%	36.9%
NService Bus (NB)	Top 1	44.5%	61.4%	16.9%	38.0%
	Top 5	55.6%	69.1%	13.5%	24.3%
Open Rastar (OR)	Top 1	36.3%	59.1%	22.8%	62.8%
	Top 5	46.1%	65.8%	19.7%	42.7%
PDF Clown (PC)	Top 1	44.8%	66.8%	22.0%	49.1%
	Top 5	56.2%	75.7%	19.5%	34.7%
RASP Library (RL)	Top 1	47.2%	68.3%	21.1%	44.7%
	Top 5	57.2%	77.6%	20.4%	35.7%

accuracy with a single suggestion is 41.3–53.6% for the lexical model and 63.5–68.6% for SLAMC. For C# projects, top-1 accuracy with SLAMC is 59.1–69.0%, while lexical model achieves only 36.3–47.2%. For top-5 suggestions, SLAMC's accuracy could be as high as 82.4% (Java) and 82% (C#). Thus, SLAMC is able to absolutely improve over the lexical model up to 25.3% of accuracy. The result also suggests different levels of code repetitiveness in different projects. It could be due to their nature and developers' coding style.

Table 4.10 Training time comparison (in seconds)

Model	BS	CC	DB	LN	MP	NB	OR	PC	RL
Lexical model	46	150	117	80	957	9	14	10	11
SLAMC	300	592	1432	1150	4958	47	32	146	142

Table 4.11 Cross-project prediction accuracy

Java project	Suggest	Lexical model	SLAMC	Improve	Relative
Ant	Top 1	44.5%	64.5%	20.0%	44.9%
	Top 5	56.6%	80.0%	23.4%	41.3%
Batik	Top 1	43.5%	66.5%	23.0%	52.8%
	Top 5	56.1%	81.1%	25.0%	44.6%
Cassandra	Top 1	45.4%	66.2%	20.8%	45.8%
	Top 5	57.7%	77.4%	19.7%	34.1%
Log4J	Top 1	47.5%	68.4%	20.9%	44.0%
	Top 5	59.6%	82.1%	22.5%	37.8%
Lucene	Top 1	53.6%	65.0%	11.4%	21.3%
	Top 5	66.2%	79.2%	13.0%	19.6%
Maven-2	Top 1	56.5%	70.4%	13.9%	24.6%
	Top 5	71.0%	83.9%	12.9%	18.2%
Maven-3	Top 1	54.2%	67.0%	12.8%	23.6%
	Top 5	68.6%	77.7%	9.1%	13.3%
Xalan	Top 1	49.6%	70.4%	20.8%	41.9%
	Top 5	61.0%	84.4%	23.4%	38.4%
Xerces	Top 1	46.6%	66.8%	20.2%	43.3%
	Top 5	59.5%	81.9%	22.4%	37.6%

Table 4.10 shows the training time of both models in all folds in the entire cross-validation process. As seen, SLAMC is much more computational expensive (2–15 times). However, it is still within a couple of hours for the largest system.

4.4.4 Cross-project training and prediction

We performed another experiment to study SLAMC's accuracy when it is trained and used for prediction with data across projects. For each Java project in Table 4.8, we re-performed 10-fold cross-validation as in Section 4.4.3. However, to predict for one fold, we used not only the other nine folds but also the other eight Java projects for training. As seen, when both models used the training data from other projects, SLAMC relatively improves over the lexical n-gram model from 13.3%–52.8% for top-1 and top-5

accuracy. This is consistent with the relative improvement of 18.1%–55.9% in Table 4.8 when training data was from only a single project.

Comparing SLAMC's accuracy in Tables 4.11 and 4.8, we can see that prediction accuracy is not improved much with using additional cross-project data for training. This is also true for the lexical model (also reported by Hindle *et al.* [26]). The similar levels of accuracy of within and cross-project settings imply that the degree of regularity across projects is similar to that in a single project.

4.4.5 Threats to validity and limitation

The biggest threat to validity of our results is that our code suggestion procedure is simulated on existing code and is not in real-world code editing settings. In addition, we re-implemented the lexical model in [26], since their implementation is not available to us. On the subject systems, we have used the dataset evaluated in the prior work, and also collected a new set of data containing nine projects. However, our selected projects might still not be representative.

The current design and implementation of SLAMC does not consider class inheritance and sometimes cannot correctly resolve types and programming roles due to incomplete code. In addition, like other statistical language models, SLAMC also faces the problem of out-of-vocabulary, i.e. it would fail to predict code elements (e.g. data types, methods) that have never been encountered in the training data.

CHAPTER 5. RELATED WORK

This section discusses the work related to the studies we presented in this dissertation. As the core of this dissertation is about capturing programming patterns in source code, we will first discuss the literature on pattern mining. Then, we will discuss the techniques focusing on detecting bugs and recommending bug fixes with patterns and similar code. We also mention the studies using statistical models to analyze source code to suggest code to be written and for other purposes.

5.1 Pattern Mining

5.1.1 Mining patterns from source code

There exist several techniques and tools for mining patterns from source code. The closest research to GrouMiner is JADET [73]. For each Java object in a method, JADET represents its usage model as finite state automaton (FSA) with anonymous states and transitions labeled with feasible method calls. This usage model is similar in spirit to our GROUM model. However, such a usage model is built for a single object and does not involve control structures. In contrast, GROUMs represent usages of multiple objects, which might involve control structures. In addition, our tool GrouMiner detects programming patterns as frequent sub-graphs of GROUMs extracted from source code. JADET represents patterns as frequent sets of pairs of method calls.

Chang *et al.* [8] also mine patterns as frequent sub-graphs. However, their work focuses on patterns involving neglected conditions. Therefore, they abstract source code

as Program Dependence Graphs (PDG) and use a maximal frequent subgraph mining algorithm to find patterns around condition nodes of those graphs. While PDG is a general-purpose model of source code, GROUM is more specialized towards API usages and GrouMiner focusses more on programming patterns involving method calls.

Several approaches focus on mining patterns involving method calls. For example, Dynamine [42] represents patterns as pairs of method calls, such as start/stop, addListener/removeListener, etc. It analyzes code changes to extract the sets of inserted methods. Then, it applies frequent subset mining on those sets to infer those patterns, as pairs of method calls occurring frequently. Acharya *et al.* [1] also express API usage patterns in term of ordered pairs of method calls. Engler *et al.* [15]'s approach is also limited to patterns as pairs of method calls. It should be noted that a pair of method calls corresponds to an edge in a GROUM. In addition, sets of method calls would be limited for representing complex object usages, such as ones involving three or more method calls or involve two or model identical objects or method calls.

PARSEWeb [69] models programming patterns as sequences of method calls. It accepts a query as a pair of "source object type" and "target object type", and searches for frequent sequences of method calls that produce a target object from a source object. MAPO [78] and MSeqGen also capture patterns as sequences of method calls. MAPO uses patterns to recommend relevant code examples, while MSeqGen uses patterns to support the automated generation of test cases.

Several approaches represent programming patterns as association rules, i.e. in the form of $A \rightarrow B$, in which A and B are sets or sequences of code units (e.g. method calls, or checks, i.e. Boolean expressions, data types/classes, etc.). For example, CAR-Miner [70] represents programming patterns of exception-handling as sequence association rules, such as [getConnection, createStatement, executeUpdate] $\rightarrow$ rollback. Thus, the components of rules (i.e. A and B) are sequences of method calls. PR-Miner [41] uses similar representation where the components are sets of function calls, variables, data types

that frequently appear in same methods. An example of those rules is {fscsi_host_alloc, scsi_add_hostg} → {fscsi_scan_hostg}. CodeWeb [46] detects patterns in term of associate rules among classes.

5.1.2 Mining patterns from other artifacts

Execution traces are frequently mined to detect programming patterns and rules (often called temporal specification mining). Gabel *et al.* [16] mine temporal properties between method calls in execution traces and express a specification as a regular expression, such as fopen fread* fwrite. This pattern indicates a usage specification of a FILE object. First, fopen is used to open the file. Then, fread could be called several times and finally, fclose is applied on the file.

Several approaches represent patterns and specification as finite state automata (FSA), of which states represent method calls. For example, an FSA represents the use of a StringTokenizer might have three states: init, hasMoreToken, and nextToken. Two latter states have edges to each other, i.e. we could call hasMoreToken, then call nextToken, and repeat. There is an edge from init to hasMoreToken, but no edge from init to nextToken, suggesting that, we should not call nextToken directly after creating the StringTokenizer.

Using this representation, Shoham *et al.* [66] applied static inter-procedural analysis for mining API specifications. Ammons *et al.* [2] analyzes execution traces and mine usage patterns in term of probabilistic FSAs, i.e. the edges are associated with probabilities of occurring. Both approaches require the alphabet of an FSA specification to be known. Pradel and Gross [61] also mine patterns as probabilistic FSAs with a mining technique that can scale up to hundreds of millions of events.

DyGen [68] mines execution traces and produces patterns as sequences of events (e.g. method calls). Its mining techniques can also scale up to huge amounts of execution events. Yang *et al.* [75] find behavioral patterns that fit into user-provided templates. Chronicler [62] uses inter-procedural analysis to find and detect violations of function

precedence protocols. Kremenek *et al.* [38] use a factor graph, a probabilistic model, to mine API method calls.

Dallmeier *el al.* [10] analyze method call sequences between successful and failing executions to detect defects. Similarly, Fatta *et al.* [12] find frequent subtrees in the graphs of calls in passing and failing runs. Dickinson *et al.* [13] cluster bugs based on their profiles to find error patterns. Fugue [11] allows users to specify object typestates and then checks for code conformance. Weimer *et al.* [74] mine method pairs from exception control paths.

Zhong *et al.* [79] infer usage specification from API documentation. Their tool, Doc2Spec, use natural language processing techniques to analyze API documentation (in textual forms) and recognize the specification for resource objects. Such specification is matched against a usage template of several phases: *creation*, *lock*, *manipulation*, *unlock*, and *closure*. For example, a usage of a SocketImpl has three phases: connect, getInputStream, and close. Similarly, a usage of a Document also has three phases: open, addTitle, and close.

There are several key differences of GrouMiner from those approaches. First, they detect patterns as FSAs with very limited automaton forms and sizes. For example, Doc2Spec's patterns often involve two or three method calls, while GrouMiner's patterns can have tens of nodes. In addition, since patterns are mined from execution traces and documents, they do not contain conditions and branching nodes and often involve usages of a single object. This line of research for mining patterns and specifications from execution traces and non-code artifacts can complement well to our graph mining approach. For example, they can detect patterns that appear only once in source code but were executed frequently execution traces.

5.1.3 Using of patterns

Most pattern mining tools use mined patterns to detect bugs. For example, JADET finds usage anomalies, i.e. rare deviations of common patterns, and reports them as potential bugs. PR-Miner finds bugs as code snippets missing some function calls from the programming patterns. CAR-Miner detect bugs as sequences of method calls that do not have the exception handing parts. Chang *et al.* [8] determines conditions that missing some checks as bugs. Representing usage patterns as graph-based models that do not limit to only conditions, GrouMiner can detect bugs related to the more complicated misuses of APIs.

Several tools use patterns for recommending code examples. MAPO analyses general code and recommends API usage patterns, as sequences of method calls, that are similar to ones are used in the code. PARSEWeb [69], XSnippet [65], and Prospector [43] provide code examples as the sequences of method calls that produce an object of a target type from an object of a source type. In contrast, Grapacc represents usage patterns as graph-based models and computes the relevancy of recommended code based on several factors, including the graph-based similarity of the editing code and the pattern, as well as, the context-sensitive location of the editing point. More importantly, Grapacc can fill in the missing part of the code following the selected pattern.

Patterns are also used to support the automated generation of test cases. MSeqGen [71] analyzes source code and mines object usage patterns as frequent sequences of method calls. Those sequences are then incorporated into Pex, a test case generation tool. Pex uses those sequences to generate objects under test and their behaviors (via different sequences of method calls). DyGen is similar in use as MSeqGen, but it mines patterns from execution traces, rather than from source code. Supporting test case generation with programming patterns is a direction we will explore in the future.

5.2 Recurring Bugs

5.2.1 Recommending recurring bug fixes

There exist the methods that aim to record recurring bug fixes. The close research to FixWizard is BugMem [36]. BugMem uses the line-based *textual* differencing approach to identify the changed textual areas (called *hunks*). BugMem's atomic fixing change is a pair of textual hunks: bug hunk (in the older revision) and fix hunk (in the new one). For each line in a hunk, BugMem extracts the syntactic units (with type information) and uses their textual values as the features of the hunk. An atomic fix is characterized by the features existing in the bug hunk but not in the fix hunk. The first difference between BugMem and FixWizard is the *program context* that is used to extract the features of a fix. Because examining only the changed area, BugMem misses the recurring fixes that involve the addition of any new code. In these cases, BugMem faces empty bug hunk and cannot detect the fix. That means, BugMem cannot handle well the recurring fixes on with the impact areas lying outside of the changed regions. In contrast, FixWizard could detect these recurring fixes because it examines also the impact usage of the change.

FixWizard is able to detect and recommend global recurring fixes due to another advantage over BugMem: the *semantics level of abstraction* for the extracted features. Instead of extracting the syntactic information from only the changed area, FixWizard performs program/data analyses from the changed area in its enclosing method. It extracts semantic features based on the object usages. The features also incorporate the relationship between code peers, thus, capturing recurring bug fixes at a higher more level of abstraction (i.e. fixes occurred and applied in *similar* code units). Another advantage is that FixWizard can recommend better in both fixing *locations* and *operations* (at method calls and statement levels, even in code that is not identical).

In Patch Miner [29], after making a fixing change, developers could use the tool to find all code snapshots with similar snippets (i.e. cloned fragments) to the fragment that

was fixed. It uses largest common subsequences of program tokens to find such cloned code. Since the level of abstraction of extracted features is at lexical tokens, it could not handle the cases that require complex program analysis (e.g. the similarity of object usages). Another difference is that Patch Miner detects *code clones* to suggest a fix, while FixWizard can also detect *similar fixes*. Thus, Patch Miner could not suggest a fix even though a similar fix occurred in the past to a peer of the current fragment.

5.2.2 Detecting similar code and bugs

Many approaches for detecting cloned code and clone-related bugs have been proposed. For example, CP-Miner [40] detects code clones as frequent sequences of tokens and detects bugs caused by inconsistent editing to cloned code. Jiang *et al.* [32] detect clone-related bugs via formulating context-based inconsistencies. Those approaches often define code clones via the similarity in their textual, lexical, and syntactic representation. In contrast, FixWizard looks for code units with similarity in term of object usages and interactions (e.g. expressed via method calls, temporal usage orders, data sharing etc.) This helps FixWizard capture similar code at a higher level of abstraction. In addition, FixWizard is able to detect not only similar code, but also similar code changes.

There exist tool supports for tracking and consistent editing of cloned code. CloneTracker [14] annotate detected code clones with their markers. When a system evolves, CloneTracker uses those markers to recognize changes to cloned code and clone groups. CReN [30] is interactive synchronization plugin for clones within Eclipse. It tracks copy-and-paste activities and and helps developers to consistently rename identifiers in copy-and-paste code. Libra [24] uses token-based clone analysis to search for cloned fragments and recommend simultaneous changes. While those tools derive recommended changes based on mappings of code tokens, FixWizard maps code and derives changes based on object usages (e.g. adding or moving a method calls).

As discussed in the previous section, there exist several approaches finding bugs based

on programming patterns (e.g. JADET, PR-Miner, CAR-Miner). They focus only on a small set of patterns and related bugs (e.g. object usages [73, 42], error-handling [70], condition checking [8]). In contract, FixWizard detects bugs via the similarity of the enclosing code context, thus, is not limited to any pattern. However, FixWizard is able to recognize new fix patterns (e.g. embedded in recurring bug fixes). Once recognized and recorded, those fix patterns can be used in for bug fixing recommendations.

Several approaches have been proposed to help users localize buggy code units [37, 23, 45, 49, 5]. Some leverage the project's historical information: the amount of changed LOC over the total in a time period [49], frequently/recently modified/fixed modules [23], code co-changes and bug locality [37], change and complexity metrics [48, 67], or social interactions among developers [47, 3, 5, 60]. Although they have achieved the good level of accuracy (60%–80%), the granularity levels of buggy area are still coarse, ranging from packages to files or methods. With slightly lower accuracy, FixWizard is able to detect bugs at statement level and provide useful operations as recommended fixes.

5.3 Statistical Modeling of Code

5.3.1 Modeling repetitiveness of code

Statistical language models have been successfully for analyzing source code. Hindle *et al.* [26] use n-gram model with lexical tokens to show that source code has high repetitiveness. This n-gram, lexical model has good predictability and is used to support code suggestion. However, SLAMC has several key advances over the lexical model. First, it annotates code tokens with semantic information, thus providing better predictability. Second, SLAMC's n-grams are also complemented with topics and pairwise associations of code elements. It allows the representation of co-occurring pairs of tokens that cannot be efficiently captured within n consecutive tokens. In addition, a novel n-gram topic model is developed in SLAMC to enhance its predictability via a global view on current

the topics (i.e. technical concerns functionality) of code.

Code repetition is also observed by Gabel *et al.* [17]. They studied 420 million LOCs in 6,000 software projects and reported *syntactic redundancy* at levels of granularity from 6–40 tokens. However, this approach considered only syntactical tokens and lexical information (e.g. identifiers) in the code sequences, while SLAMC operates at the semantic level. Han *et al.* [22] have used Hidden Markov Model (HMM) to infer the next token from user-provided abbreviations. Abbreviated input is expanded into keywords by an HMM learned from a corpus. HMM captures only local contextual information, while SLAMC has also topics and pairwise associations. An n-gram model has been used to find code templates relevant to current task with n-grams built from clone groups [31].

5.3.2 Enhancing code completion with statistical properties

Bruch *et al.* [7] propose three algorithms to suggest the method call for a variable v based on a codebase. First, FreqCCS suggests the most frequently used method in the codebase. Second, ArCCS is based on mined associate rules where a method is often called after another. The third algorithm, best-matching neighbors, uses as features the set of method calls of v in the current code and the names of the methods that use v. The features of methods in examples are matched against those of the current code to find the relevant suggestions.

Precise [76] completes the parameter list of a method call. It mines a codebase to build a parameter usage database. Upon request, it queries the database to find best matched parameter candidates and concretizes the instances. Omar *et al.* [57] introduce active code completion in which interactive and specialized code generation interfaces are integrated in the code completion menu to provide additional information on the APIs in use.

Other strategies have been proposed to improve code completion. Hill and Rideout [25] use small *cloned fragments* for code completion. It matches the fragment under

editing with small similar-structure code clones. Robbes and Lanza [64] introduced six strategies to improve code completion using *recent histories* of modified/inserted code during an editing session and on the methods and class hierarchy related to the current variable. Hou and Pletcher [28] found that ranking method calls by frequency of past use is effective. Eclipse[1] and IntelliJ IDEA[2] support *template-based* completion for common constructs/APIs (for/while, Iterator). Strathcona [27] extracts *structural context* of the current code and finds its relevant examples. Mylyn [34], a code recommender, learns from a developer's personal usage history and suggests related methods

[1]http://www.eclipse.org - Accessed at 12:25 on 12/02/2013
[2]http://www.jetbrains.com/idea - Accessed at 12:26 on 12/02/2013

CHAPTER 6. FUTURE WORK AND CONCLUSIONS

As seen from previous chapters, GROUM and SLAMC, as abstract models of source code, could be used to represent and recover patterns and regularities of code. The inferred patterns are both general and specific to the codebases where those models are extracted from. Thus, a direction for future work is to study programming patterns and code regularities of specific codebases, e.g. on code changes, bugs, or bug fixes. Another direction is to study the relationship of code in different codebases via their patterns, e.g. mapping APIs of different frameworks like Android and iOS. In the next sections, we will discuss some promising on-going projects that we are investigating.

6.1 Future Work

6.1.1 Personalized and domain-specific code modeling

In this direction, we address the question that whether our models could be personalized or specialized for different projects or domains? It is possible that different developers have different areas of expertise and programming styles/preferences. For example, a developer works on back-end aspects of the systems might use different APIs, and thus, express different concepts, concerns, and patterns in code than another works on front-end aspects. Or, one might prefer to use enhanced `for` loops or new language features, while another prefers classic constructs. Similarly, different domains, projects, or components would involve different programming patterns and idioms. For example, usage patterns of Database APIs would have different elements and structures from Graphics

APIs. A project could have project-specific patterns that are not likely to be used in other projects. We are interested to study whether language models trained separately for individual developers or projects could recover such personalized or project-specific aspects and and are more effective in modeling corresponding patterns and regularities. If such, personalized/specialized models could predict and recommend patterns or activities more relevant for each individual in a specific context. Let us discuss some of such applications of those personalized/specialized models in the following scenarios.

6.1.1.1 Adaptive code recommendation

Personalized and specialized models could be used to enhance the quality of code recommendation. For example, a SLAMC-based language model could be trained on code written by an individual developer to learn his/her expertise, preferences, and styles, thus, code recommendations based on that model will be more personalized towards him/her. If (s)he prefers using StringBuilder to StringBuffer, the occurrences of patterns involving StringBuilder in his/her code would be higher. The language model personalized for him/her will have higher probability for patterns involving StringBuilder, and thus, the recommendations would be driven towards StringBuilder more.

Similarly, domain or project-specific programming patterns could be mined from corresponding domains and projects and be used for corresponding contexts. For example, when a developer are writing code for back-end functionality like database access, programming patterns involving Database APIs (e.g. Java JDBC Connection or Statement classes) are more relevant and likely to be used than the patterns involving front-end functionality like Graphics APIs (e.g. Graphics or Image).

Personalized and specialized models and patterns should be iteratively updated over time to accommodate the evolution of the software projects as well as the change in personal experience and preferences of software developers. In other words, we need adaptive methods to recover patterns and regularities for improving the quality of rec-

ommendation. Such adaptive methods will need incremental algorithms for training language models and mining programming patterns to avoid the cost of training/mining from scratch and the lost of adaptive information collected over time.

6.1.1.2 Task assignment

Personalized language models could be used to recommend task assignment, i.e. finding developers most suitable for a development task, e.g. fixing a bug or implementing a feature request. We are interested in studying whether a language model trained on code written/maintained by an individual developer could measure his/her experience. That is, given a code unit of interest s and several language models personalized for several developers. If the generation probability of s by the personalized model of a developer x is higher than those of other developers, it is possible that x is more familiar with that code unit than others, and thus, could perform some tasks such as refactoring or bug fixing on that code unit better than the others.

6.1.1.3 Code authorship

Another related direction is to study whether personalized language models for source code could be used to measure code authorship. That is, given a code unit of interest and two programmers, we need to determine who is more likely to be the author of that code unit. Similar to the problem of task assignment, we could train two personalized language models on the code that is known to be written by those two developers, respectively. Then, we use those models to compute the respective generating probabilities of the given code unit. If a probability is significantly higher than the other, the corresponding programmer is more likely to write that code unit.

Specialized language models might be used to solve the similar problem of code plagiarism. That is, we want to determined whether a piece of code is copied from a codebase of a software project. To do that, we could train two language models, one

specialized for the given codebase and another for a very large code corpus, which could be considered to contain all general programming knowledge. Then, if the generating probability of the given piece of code by the specialized language model trained on the given codebase is significantly higher is significantly higher than the probability by the general model, we could conclude that the given code is likely to be copied and modified from that codebase than written from scratch using general programming knowledge.

6.1.2 Finding and fixing programming errors

6.1.2.1 Detecting bugs

Statistical language models have been used to detect and correct spelling and syntax errors in natural text. For example, in English text, *"I are"* is unlikely to appear than *"I am"* or *"I have"*. Or similarly, *"have went"* is less likely compared to *"have gone"* or *"have to go"*. Thus, based on the generating probability of a phrase or a sentence by a language model trained on a appropriate corpus, (compared relatively to similar ones), we could determine whether that phrase or sentence is unlikely, thus, potentially has spelling or syntax error. Similar phrases or sentences which have much higher generating probabilities would then be recommended for corrections.

Following the same idea, we would like to study how language models specialized for source code could be used to detect and correct programming errors. For example, to find erroneous API usages, we could extract code related to API usages to train a language model specialized for such code. Then, this model could be used to compute the generating probability for a given API usage, and if that probability is significantly low, the usage could be reported as a possibly wrong one. The most likely usages deviated from this usage could be recommended as the corrections.

6.1.2.2 Change and fix patterns

Change and fix patterns are code changes and error repairs that occur frequently. Similarly to usage patterns, those patterns could be used to detect bugs or recommend changes or bug fixes. Preliminary studies show promising results. For example, we have used GROUMs to represent API usages and their changes are analyzed to infer change patterns of API usages, which then are used to recommend API adaptation [53]. Kim *et al.* used GROUM to analyze bug patches of small programming errors [35]. By clustering changes of GROUMs of original and patched versions, they have found several bug fixing patterns. Those patterns are then used to automatically generate bug patches for other erroneous programs.

Those studies focus on special changes (e.g. API adaptations) or small patches (often one-liner), collected from small scale repositories. To improve the effectiveness of the recommendations, recommended changes/fixes should be combined from smaller mined patterns. Those patterns should be mined from larger scale repositories and not be specialized to some special kinds of changes. Compared to GROUM, SLAMC represents more code elements and could capture patterns at finer-grained levels, thus, is more suitable for modeling those change/fix patterns. We are investigating in new techniques for mining change and fix patterns from large scale corpora and using search-based methods to find the best recommended combinations of those patterns for a given program that need to be changed or fixed.

6.1.3 Statistical program transformation

Nowadays, to address business needs, the same software might need to work on multiple platforms and environments. For example, with the fast growing market of mobile devices, an mobile app originally developed for Apple iOS platform (Objective-C) might need to be migrated to Android (Java) or Windows Phone (C#). Manually migrating code written in one language/platform to another is time-consuming and error-prone. Ex-

isting (semi-)automated migration approaches are based on predefined translation rules for the constructs and APIs between two languages. Existing methods expect users to manually specify such rules, which is also a tedious and error-prone task. Since there are a large number of mappings such as those among APIs provided in different languages, manual rule definition is time-consuming, insufficient, and not scalable.

Thus, we are also interested in using statistical models for source code for program transformation, such as API or source code migration (e.g. from Java ME to Android, or from Java to C#). Adopting statistical machine translation, we will develop statistical models that present the transformation process of source code in one API or a programming language to a closely similar one. Then, the trained statistical models could be used to guide or automate the migration process.

The core idea of our approach is based on statistical machine translation. That is, we build a translation model T, which could specify how likely a code sequence s in the source API/language is transformed to a new code sequence t in the target API/language. Statistically, this model allows us to computes the translation probability $P(t|s)$ for any source sequence s and and target sequence t. Then, for a given source code sequence, we support the transformation task by generating and recommending the most likely translated code sequences.

6.1.4 Automated code translation

For automated code translation, code sequences should contain all possible code tokens. However, using lexical tokens (i.e. lexemes) would cause a high level of syntactic errors resulted from statistical machine translation, since syntactic information is lost. Unlike a natural language, a programming language has well-defined syntactic rules and unambiguous semantics. Thus, the grammar of two programming languages could be used to direct the translation process. Syntactic units could be translated and used as placeholders for a later migration process for the tokens within each of them.

Our results in Chapter 4 suggest that adding semantic information to code tokens, rather than lexical tokens, has improved the overall quality of the language model. Therefore, a future direction would be to use program analysis to extract the semantic information of program elements such as their types, roles, etc. Such information could help the translation model better align the code within syntactic units and improve the quality and efficiency when translating. Finally, post-processing could also be applied to correct the resulting code. Such a process could use program analysis techniques to make sure the migrated code correct. Test cases could also be used to validate the resulting code.

6.1.5 API mapping

In language migration and software evolution, one common task is to identify the mappings of elements in two API frameworks/libraries. For example, to migrate a mobile app from Android to Windows Phone, one needs to identify the API elements (e.g. functions, data types, classes, methods) that provide the same/similar functionality between two frameworks. This API mapping task is mainly manual [19].

To reduce manual effort in building the API mapping rules, researchers have propose several approaches that automatically infer such API mappings from already migrated code [77] or software applications having the same functionality [19] in corresponding frameworks. Despite their differences in details, they all share the principle that two corresponding API elements in two languages have textually similar names, the calling structures and parameters are similar in the execution traces are similar.

We are working towards a new approach that uses statistical models to learn the mappings between API usages from the corpus of the corresponding client code of API elements in two frameworks or languages. Instead of using heuristics on the textual, structural similarity between single API elements in two framworks/languages to derive the API mappings, our approaches use a statistical machine translation model to derive the alignments API usages, modeled as sequences of method invocations. GROUM is

originally used to represent API usages and then sequences of method invocations are extracted from GROUMs. As statistical machine translation models use only co-occurrence frequencies and do not require the aligned sequences to have textual or structural similarity, we expect that this approach can detect API usage mappings with higher accuracy than the state-of-the-art approach.

6.2 Final Conclusions

Software has an important impact to our economy and society. However, due to the high complexity and the ever-changing nature of software systems and development processes, programming errors and software failures, often called bugs, occur unavoidably. Bugs are causing huge financial losses and creating high risk to human lives. Therefore, reducing the incidence of bugs and improving the efficiency of the bug-fixing process is a main research area in software engineering.

This dissertation proposes two abstract models of source code: GROUM (for object usages) and SLAMC (for code sequences with semantic annotations) and several automated techniques and tools using those models to recover programming patterns and language regularities from written code, to detect programming errors and security vulnerabilities, and to guide developers writing code faster and less error-prone following those programming patterns. Those techniques employ efficient and scalable graph-based and statistical algorithms to analyze source code and the proposed abstract models.

Empirical evaluation results show that our models and techniques are highly expressive and effective. Our techniques have been able to recover programming patterns of high quality and accurately detect many errors and vulnerabilities. In addition, they are also highly efficient and scalable, processing large software systems within reasonable time. When using the recovered programming patterns for code recommendation and completion, our tools are useful and outperform the state-of-the-art techniques.

This line of research also has many other potential applications. A future direction is to focus on specialization, i.e. studying programming patterns and code regularities of specific codebases, e.g. on code changes, bugs, or bug fixes. Another direction focuses on association, i.e. studying the relationship of code in different codebases via their patterns, e.g. mapping APIs of different frameworks such as Android and iOS. The last direction focuses on generalization. Because code and other software artifacts are continuously produced and released, codebases and software repositories are getting larger and larger. Thus, scaling the mining and training algorithms to such large-scale datasets is also a necessary and promising research direction.

BIBLIOGRAPHY

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 25–34. ACM, 2007.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM, 2002.

[3] E. Arisholm and L. C. Briand. Predicting fault-prone components in a Java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 8–17. ACM, 2006.

[4] P. Baldi, C. V. Lopes, E. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 2008 SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 543–562. ACM, 2008.

[5] C. Bird, D. Pattison, R. D. Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–35. ACM, 2008.

[6] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software: Quantify the time and cost saved using reversible debuggers. *University of Cambridge*, 2013.

[7] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 213–222. ACM, 2009.

[8] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, 2008.

[9] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the 2008 SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 313–328. ACM, 2008.

[10] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *Proceedings of the 2005 European Conference on Object-Oriented Programming*, pages 528–550. Springer, 2005.

[11] R. DeLine and M. Fahndrich. Typestates for objects. In *Proceedings of the 2004 European Conference on Object-Oriented Programming*. pages 465–490. Springer, 2004.

[12] G. Di Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, pages 62–69. ACM, 2006.

[13] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 2001 International Conference on Software Engineering*, pages 339–348. IEEE, 2001.

[14] E. Duala-Ekoko and M. P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20(1):3:1–3:31, 2010.

[15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72. ACM, 2001.

[16] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 339–349. ACM, 2008.

[17] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 147–156, ACM, 2010.

[18] M. Gallaher and B. Kropp. Economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[19] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 82–91. IEEE, 2013.

[20] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 119–130. ACM, 2010.

[21] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[22] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. IEEE, 2009.

[23] 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*, pages 263–272. IEEE, 2005.

[24] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 262–269, 2007.

[25] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 2004 IEEE/ACM International Conference on Automated Software Engineering*, pages 228–235. IEEE, 2004.

[26] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 837–847. IEEE, 2012.

[27] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 2005 International Conference on Software Engineering*, pages 117–125. ACM, 2005.

[28] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 233–242. IEEE, 2011.

[29] Pattern Insight. Eliminate previously fixed bugs to improve your software quality. http://genius-cs.com/lpe/lpe/4443/lp188.html - Accessed at 12:15 on 12/02/2013.

[30] P. Jablonski and D. Hou. CREN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20. ACM, 2007.

[31] F. Jacob and R. Tairas. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, pages 104:1–104:6. ACM, 2010.

[32] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 55–64. ACM, 2007.

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

[34] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 1–11. ACM, 2006.

[35] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE, 2013.

[36] S. Kim, K. Pan, and E. Whitehead Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–45. ACM, 2006.

[37] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 2007 International Conference on Software Engineering*, pages 489–498. IEEE, 2007.

[38] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12. USENIX Association, 2006.

[39] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41. IEEE, 1993.

[40] Z. Li, S. Lu, and S. Myagmar. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* , 32(3):176–192, 2006.

[41] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 306–315. ACM, 2005.

[42] V. B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 296–305. ACM, 2005.

[43] 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*, pages 48–61. ACM, 2005.

[44] C. D. Manning and H. Schutze. Foundations of statistical natural language processing. *MIT Press*, 1999.

[45] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[46] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 167–176. ACM, 2000.

[47] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2002.

[48] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 2008 International Conference on Software Engineering*, pages 181–190. ACM, 2008.

[49] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 2005 International Conference on Software Engineering*, pages 284–292. ACM, 2005.

[50] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 69–79. IEEE, 2012.

[51] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference Automated Software Engineering*, pages 70–79. IEEE, 2012.

[52] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 2009 International Conference on Fundamental Approaches to Software Engineering*, pages 440–455. Springer, 2009.

[53] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of 2010 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 302–321. ACM, 2010.

[54] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 2010 International Conference on Software Engineering*, pages 315–324. ACM, 2010.

[55] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134. IEEE, 2009.

[56] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 383–392. ACM, 2009.

[57] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 859–869. IEEE, 2012.

[58] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 2009 International Conference on Software Engineering*. ACM, 2009.

[59] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering*, pages 447–456. ACM, 2010.

[60] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2–12. ACM, 2008.

[61] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 371–382. IEEE, 2009.

[62] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 2007 International Conference on Software Engineering*, pages 240–250. IEEE, 2007.

[63] R. Read and D. Corneil. The graph isomorph disease. *Journal of Graph Theory*, 1:339–363, 1977.

[64] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE, 2008.

[65] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proceedings of the 2006 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 413–430. ACM, 2006.

[66] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 174–184. ACM, 2007.

[67] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 107–110. ACM, 2005.

[68] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proceedings of the 4th International Conference on Tests and Proofs*, pages 77–93. Springer, 2010.

[69] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the Web. In *Proceedings of the 2007 IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213. ACM, 2007.

[70] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 2009 International Conference on Software Engineering*, pages 496–506. IEEE, 2009.

[71] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 193–202. ACM, 2009.

[72] H. M. Wallach. Topic modeling: Beyond bag-of-words. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 977–984, ACM, 2006.

[73] A. Wasylkowski, A. Zeller, , and C. Lindig. Detecting object usage anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–44. ACM, 2007.

[74] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 2005 International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[75] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 2006 International Conference on Software Engineering*, pages 282–291. ACM, 2006.

[76] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 826–836. IEEE, 2012.

[77] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 2010 International Conference on Software Engineering*, pages 195–204. IEEE, 2010.

[78] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 2009 European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

[79] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring specifications for resources from natural language API documentation. *Automated Software Engineering Journal*, 18(3–4):227–261, 2011.