EXPLORING ACTION UNIT GRANULARITY OF SOURCE CODE FOR SUPPORTING SOFTWARE MAINTENANCE

by

Xiaoran Wang

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2017

EXPLORING ACTION UNIT GRANULARITY OF SOURCE CODE FOR SUPPORTING SOFTWARE MAINTENANCE

by

Xiaoran Wang

Approved: ____
Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: ____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: ____
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

ACKNOWLEDGEMENTS

It would not have been possible to write this doctoral thesis without the help and support of the people in my personal and professional life. Foremost, I would like to thank my wife for her great patience and personal support all the time. My parents have given me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice. I would like to express my sincere gratitude to my advisors, Dr. Lori Pollock and Dr. Vijay Shanker, for their continuous support of my Ph.D study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. Besides my advisors, I would like to thank the rest of my thesis committee: Dr. Emily Hill and Dr. James Clause. Special thanks go to my fellow labmates and friends in University of Delaware: Benwen Zhang, Cagri Sahin, Irene Manotas, Chen Huo, Ningjing Tian, Yifan Peng, Yuanfang Chen, Gang Li, Ruoyao Ding, Samir Gupta, Ashique Mahmood, Yang Guan and Xiaolong Wang, for the stimulating discussions and for all the fun we have had in the last few years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Because resources for today's software are used primarily for maintenance and evolution, researchers are striving to make software engineers more efficient through automation. Programmers now use integrated development environments (IDEs), debuggers, and tools for code search, testing, and program understanding to reduce the tedious, error-prone tasks. A key component of these tools is analyzing source code and gathering information for software developers. Most analyses treat a method as a set of individual statements or a bag of words. Those analyses do not leverage information at levels of abstraction between the individual statement and the whole method. However, a method normally contains multiple high-level steps to achieve a certain function or execute an algorithm. The steps are expressed by a sequence of statements instead of a single statement. In this dissertation, I have explored the feasibility of automatically identifying these high level actions towards improving software maintenance tools and program understanding.

Specifically, methods can often be viewed as a sequence of blocks that correspond to high level actions. We define an action unit as a code block that consists of a sequence of consecutive statements that logically implement a high level action. Rather than lower level actions represented by individual statements, action units represent a higher level action, for example, "initializing a collection" or "setting up a GUI component". Action units are intermediary steps of an algorithm or sub-actions of a bigger and more general action. In this dissertation, I (1) introduce the notion of action units and define the kinds of action units, (2) develop techniques to automatically identify actions for loop-based action units, (3) automatically generate natural language descriptions for object-related action units, and (4) automatically insert blank lines into methods based on action units to improve source code readability.

Chapter 1
INTRODUCTION

Reports indicate that as much as 60-90% of software engineering resources are spent on maintenance [27]. *Software maintenance* is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [58]. Software maintenance involves the integrated use of source code and other software artifacts [3]. A large portion of the effort is spent in reading source code [24, 33, 38, 48, 49, 74, 91, 94] to gain the necessary understanding needed to make any modifications.

To reduce the effort of program comprehension and software maintenance, many automated and semi-automated analyses and associated tools have been developed to support the tedious and error-prone tasks [107, 113]. Many of these tools support the software maintenance process by analyzing source code and gathering information for software developers. Most analyses treat methods as a set of individual statements or a single unit. For instance, most existing concern location tools treat a method as a "bag of words" [61], i.e., a method is viewed as one document containing a set of words during information retrieval. Some source code summary generators process methods as a set of individual statements and then select a subset of statements for which to generate a summary [102]. Others use methods as a "bag of words" and select a subset of words for the summary [37]. The current analyses do not leverage information at levels of abstraction between the individual statement and the whole method. Programming language design restricts the way we write code in such a way that more than one statement is often needed to implement a given step of an algorithm that a given method is implementing. The step is too small to write as a single method, but also takes more than one statement. One indicator that these higher level steps are important is that humans often delineate them by blank lines between the blocks or document them as internal comments above the blocks. In this dissertation, I explore the feasibility of automatically identifying the high level steps, or actions, of a given method and how to use the high-level action information towards improving programming understanding. Specifically, methods can often be viewed as a sequence of blocks that correspond to high level actions. We define an **action unit** as a code block that consists of a sequence of consecutive statements that logically implement a high level action. Rather than lower level actions represented by individual statements, action units represent higher level actions, for example, "initializing a collection" or "setting up a GUI component". These high level actions are intermediary steps of an algorithm or subactions of a bigger and more general action.

Utilizing high level action information can benefit existing software maintenance tools. For instance, automatic comment generators [102, 105] for method summaries frequently fall short for long methods because they generate comments based on individually selected statements. The generated summary either contains too much detailed information or is too brief to describe the method accurately. However, key steps are often implemented by a sequence of statements instead of a single statement. With the identification of high level action information, intermediary steps could be captured and composed for the summary. This may lead to summaries with higher accuracy and in a more concise form.

Another example is concern location tools. Current concern location tools that use a "bag of words" approach focus on the individual words in the code. However, a sequence of statements might implement an "update" or "check" for some variable, but there may be no such words in the code. In this situation, relying only on words in source code would not locate the code related to an "update" query word. Integrating high level action information could help concern location tools find such features by including the high-level action descriptions with the code.

In addition, analyses of high level actions could potentially enable creation of new tools for software maintenance. Internal comments are frequently used by developers, and when they are written, they often represent a block of code, or a high-level action [40]. Internal comments describe sub-actions or steps represented by the block. Unfortunately, developers rarely write internal comments [105]. By using the approach of action unit identification and natural language processing (NLP) techniques, internal comments can be generated in useful locations to describe actions of intermediary steps. Intuitively, human-inserted comment locations can be used to guide the identification of candidate insertion points for automatically generated internal comments.

Lastly, developers frequently use blank lines to separate different logical sections to make the code more readable. If action units can be identified automatically, blank lines can be automatically inserted.

The main contributions of this dissertation include (1) introducing the notion of action units and defining the kinds of action units, (2) developing techniques to identify actions for loop-based action units, (3) generating natural language descriptions for object-related action units, and (4) inserting blank lines into methods based on action units to improve source code readability. Our techniques are developed for Java, since Java is reported as the most popular programming language [45].

Chapter 2 introduces the related work of text analysis for software maintenance tools. Chapter 3 presents the notion of action units, their kinds and similar notions. Chapter 4 describes an automatic approach to identify loop-based action units and label their high level actions with a verb phrase. Chapter 5 describes an approach to generate natural language descriptions for object-related action units. Chapter 6 shows the usefulness of action units by applying the notion to insert blank lines for improving code readability. Chapter 7 presents a study of reader perspectives of action units. Chapter 8 summarizes this thesis and outlines directions for future work.

Chapter 2

TEXT ANALYSIS FOR SOFTWARE MAINTENANCE TOOLS

Despite decades of developing software engineering techniques, as much as 60-90% of software life cycle resources are still used for software maintenance [27]. Building effective software tools is important to reduce these high maintenance costs. Information retrieval (IR) techniques have been adopted to search source code and reconstruct documentation traceability in source code [39, 43, 65, 84, 89, 117, 132]. In recent years, researchers have integrated natural language processing (NLP) techniques to analyze the natural language clues in program literals, identifiers, and comments [22, 26, 41, 42, 54, 60, 97]. In both cases, text analysis is leveraged to increase the effectiveness of many software tools. This chapter presents the related work in text analysis for software maintenance tools. Information retrieval on source code is discussed first, followed by natural language processing on source code and formatting for program understanding. Finally, the shortcomings and remaining issues are summarized.

2.1 Information Retrieval on Source Code

As software systems continue to grow and evolve, locating code for software maintenance tasks becomes increasingly difficult. To modify an application, developers must identify the high-level idea, or concept, to be changed and then locate, comprehend, and modify the concept's concern in the code [64]. There are mainly two types of searches that are used to locate concerns: lexical-based and information retrieval-based.

Programmers commonly use lexical searches to locate concepts in code using regular expression queries. The problem with lexical search tools like *grep* [34] is that regular expression queries are often fragile, causing low recall. Many natural language features cause regular expression queries to exhibit low recall, including morphology changes, synonyms, line breaks, and reordered terms [97]. For example, a user's search for the concept "take" will fail if he/she uses a regular expression search for "take" and the concern is implemented using a different morphological form of the word, such as "took". Expert developers often solve this problem by searching for more general terms, which leads to large result sets with low precision and no ranking of relevance within the large result sets [97].

IR technology uses the frequency of words in documents to determine similarity between documents and queries. Because IR calculates a similarity score, the results of a query according to relevance are ranked. In addition, IR also gracefully handles multiple word queries [97]. IR does not, therefore, suffer from all the difficulties of regular expression queries. IR's search queries are not fragile and it does not return unranked result sets. Researchers have successfully applied IR to locate concepts [39, 43, 65, 83, 84, 89, 115, 132] and reconstruct documentation traceability links in source code [8, 63, 68, 77]. Recently proposed approaches to bug localization and feature location also integrate the *positional proximity* of words in the source code files and the bug reports to determine the relevance of a file to a query [98].

However, IR-based concern location tools treat source code as a "bag of words" and focus on the individual words in the code [43]. The effectiveness of IR still depends on the individual words of the query or its synonyms also appearing in the document. In a typical Java method, a sequence of statements might implement a concept or concern that is not expressed in the words that the developer uses to query. In this situation, relying only on individual words in source code would not locate the code related to the query word.

Researchers have developed techniques to reformulate queries and use word relations to improve the queries [31]. Recent techniques take advantage of *phrasal concepts*, which are concepts expressed as groups of words such as noun phrases (NP) and verb phrases (VP). Phrasal concepts improve search accuracy [44] and suggest alternate query words [97]. Although these techniques reduce the requirements of exact word matching, concern location techniques still largely rely on individual query words.

Action unit could provide a new perspective to view the source code. Instead of viewing a method as a set of individual statements, a method is viewed as a set of high level actions. Our hypothesis is that integrating this new higher level information would provide additional representative words in the document and more succinct abstractions for existing concern location tools.

2.2 Natural Language Processing on Source Code

As discussed in Section 2.1, IR-based techniques have been applied to locate concepts and reconstruct documentation traceability in source code. However, IR-based methods do not consider crucial information regarding relationships between terms that NLP analysis can provide. It has been demonstrated that many tools that help program understanding for software maintenance and evolution rely on, or can benefit from, analyzing the natural language embedded in identifier names and comments using NLP techniques [2, 11, 97, 105, 129, 130, 131].

Automated analysis of program identifiers begins with splitting the identifier into its constituent words and expanding abbreviations to tokenize the name. Unlike natural languages, where space and punctuation are used to delineate words, identifiers cannot contain spaces. One common way to split identifiers is to follow programming language naming conventions. For example, Java programmers often use camel case, where words are delineated by uppercase letters or non-alphabetic characters. However, programmers also create identifiers by concatenating sequences of words together with no discernible delineation, which poses challenges to automatic identifier splitting. Several approaches have been developed to automatically split the identifiers into their constituent words and expand the abbreviations [22, 26, 42, 54, 60].

Many text-based tools for software engineering use part-of-speech (POS) taggers, which identify the POS of a word and tag the word as a noun, verb, preposition, etc. and then chunk (or parse) the tagged words into grammatical phrases to help distinguish the semantics of the component words. Automated POS tagging and parsing in software is complicated by several programmer behaviors and the way that POS taggers typically work. POS taggers for English documents are built using machine learning techniques that depend on the likelihood of a tag given a single word and the tags of its surrounding context data; these taggers are trained on typical natural language text such as the Wall Street Journal or newswire training data [114]. These taggers work well on newswire and similar artifacts; however, their accuracy on source code reduces as the input moves farther away from the highly structured sentences found in traditional newswire articles. To improve the POS taggers for software artifacts, several techniques take program identifiers as input and generate sentences or phrases to input to a classic POS tagger for English text. Abebe and Tonella [2] applied natural language parsing to sentences generated from the terms in a program element. Depending on the program element being named (class, method, or attribute) and the role (noun/verb) played by the first term in the identifier, different templates are used to wrap the words of the identifier to form sentences, which are input to a parser to construct parse trees. Binkley et al. [11] presented a POS tagger for field names. They followed the approach of using templates like Abebe and Tonella. Although they use fewer templates (List, Sentence, Noun and Verb) than Abebe and Tonella, they produce sentences with each template, which are then input to a POS tagger developed for the general English domain. Unlike the Abebe and Tonella system, they do not output a unique tag for each word, but rather produce four different tags for each template. Falleri et al. [29] simply use the TreeTagger [96] which is trained on English text to perform the POS tagging. This closely relates to the sentence template used by [2] except the use of a different language parser. Gupta et al. [35] presented a POS tagger and syntactic chunker for source code names that takes into account programmers' naming conventions to understand the regular, systematic ways a program element is named. This POS tagger is able to handle the common as well as the less common naming conventions equally well and not limited to method names.

Because the software writer and reader who is searching the code are often different, the effectiveness of search tools is often improved by adding related words to textual artifact representations [46]. Synonyms are particularly useful to overcome the mismatch in query to document vocabularies, as well as other word relations that indicate semantic similarity. Sridhara et al. [104] investigated whether six publicly available, well known semantic similarity techniques that perform well on English text [15, 67, 82] are directly applicable to the software domain. Their study indicated that all six English text-based approaches perform poorly when applied to the software domain. Their qualitative study also suggested that one promising way to customize semantic similarity techniques for software is to augment WordNet with relations specific to software, as some pairs were identified due to WordNet being augmented with some very common software word relations, such as (*remove*, *clear*). Yang and Tan [126] developed an approach to automatically identify semantically related words by leveraging the context of words in comments and code. The approach is based on the key insight that if two words or phrases are used in the same context in comment sentences or identifier names, then they likely have syntactic and semantic relevance. Howard et al. [46] presented an approach to automatically mine word pairs that are semantically similar in the software domain, with the goal of automatically augmenting WordNet for use in software engineering tools. The key insight of their approach is to map the main action verb from the leading comment of each method to the main action verb of its method signature. This work is complementary to Yang and Tan [126].

Many software related words are not in the source code itself, but instead are in the various associated textual artifacts, such as forum posts, bug reports, commit logs, etc. Therefore, Tian et al. [112] developed an automatic approach that builds a software specific WordNet like database by leveraging the textual contents of posts in Stack Overflow. They measure the similarity of words by computing the similarities of the weighted co-occurrences of these words in the textual corpus. In addition to the work of mining semantically-similar Words, Wang et al. [125] infer semantically related tags from FreeCode. Falleri et al. [29] showed how to extract a network of identifiers connected by is-more-general-than or is-a-part-of relationships from source code.

Some researchers have developed automatic techniques to capture co-occurring word pairs [59, 62], but co-occurrences do not capture any information about the nature of the relationship between words beyond that the words occur together in the same context. However, others have leveraged the POS tagging to analyze identifiers and build models of the usage of words in identifiers. Shepherd et al. showed that representing method identifiers as verb and direct object pairs can improve search by focusing on the actions and what they action upon [97]. Hill [41] extended this approach to the Software Word Usage Model (SWUM) in which a set of identifier grammar rules for Java identifiers were developed. SWUM has been used for several software tools [41, 43, 70, 103, 105].

One significant use of SWUM was for comment generation. Sridhara et al. developed a technique to automatically generate summary comments for Java methods based on structural and linguistic clues captured by SWUM [102]. The premise behind the technique is that a method's key actions that would be content for a summary can be represented by selecting important single statements to represent key actions of the method and used as a basis for a natural language description of the method's functionality. They developed heuristics to choose individual isolated statements and generate a natural language phrase for a sequence of these selected statements to serve as the method's summary comment. The major drawback of this approach is that individual statements often do not represent the main action of the method. To begin to address this problem, they developed a technique to generate natural language descriptions for high level actions [105]. They manually identified a set of key patterns of code sequences and developed a template-based approach to recognize the occurrence of these patterns in code and then generate a phrase description of the main action based on knowing the high level action that those patterns implement. This work on high level action identification and description motivated this dissertation work. Study of the template-based approach indicated that it identifies high level actions for a small fraction of code [120]. The goal of this dissertation is to develop a scalable approach that can identify any action units without manually defining the templates.

Complementary to this work, the same authors also presented a technique to generate comments for parameters and integrated those descriptions with method summaries [106]. Moreno et al. developed an approach that generates summaries for Java classes [72].

Wong et al. mined question and answer sites for automatic comment generation [123]. They first extracted code-description mappings from the question title and text, and then used code clone detection to find source code snippets that are almost identical to the code of the code-description mappings. Their technique does not leverage the higher level abstraction of source code, which leads to only generating comments for a small number of code fragments.

There has also been research into extracting topic words or verb phrases from source code [43, 68, 77] to identify code fragments that are related to a given action or topic. Other research clusters program elements that share similar phrases [53]. These approaches rely solely on the linguistic information to determine the topic of the code segment, which we found is not adequate for describing many action units where the action is not expressed as a word within the source code explicitly.

2.3 Formatting for Program Understanding

The readability of a program is related to its maintainability, and is thus a key factor in overall software quality. Readability is so significant that Elshoff and Marcotty proposed adding a development phase in which the program is made more readable, because many commercial programs were much more difficult to read than necessary [25].

Readability metrics help to identify potential areas for improvement to code readability. Through human ratings of readability, Buse et al. [17] developed and automated the measurement for a metric for judging source code snippet readability based on local features that can be extracted automatically from programs. More recently, Daryl et al. [86] formulated a simpler lightweight model of software readability based on size and code entropy, which improved upon Buse's approach. Daryl et al. observed that indentation correlated to block structure has a modest beneficial effect on readability. Other earlier works measured program characteristics with different definitions of readability [3, 12].

In addition to the code's size and complexity, the readability of source code can be affected by the identifier names, comments, and overall appearance [19]. This observation led to the invention of prettyprinters [50], which automatically reformat a source code to improve the appearance or fit a specific coding style. It is capable of pretty-printing just a part of a program, corresponding to an editor view. The running time of the pretty-printer is independent of the full length of the program, and only the size of the editor's view matters. The algorithm gives the same result as if the entire program was pretty-printed, avoiding reformatting artifacts as the user scrolls the editor view. Bagge and Hasu [9] developed a pipeline approach to format source code. Their approach has a pipeline of connected components to format code, where the various concerns of producing pretty code are separated into different processors; one for inserting horizontal space, one for breaking lines, and one for adding colors. As separate tools or within IDEs, these prettyprinters perform textual transformations such as placing certain kinds of statements on separate lines, indenting and left-justifying certain lines, inserting blank lines before specific syntactic units such as certain keywords, or removing extra blank space. These transformations are purely textual, not semantic, and typically use the keywords and maybe the syntax to identify and perform transformations. While code readability involves the syntactic appearance of code, poor readability is perceived as a barrier to program understanding, which focuses on the semantics [86].

Organizations employ coding standards to maintain uniformity across code written by different developers, with the goal of easing program understanding for newcomers to a code [47, 55, 108, 109, 134]. Some guidelines [36, 108, 134] have clearly specified the number of blank lines that should be used to separate methods in a class and class fields. They suggest that blank lines should be used to separate different logical sections within methods. However, a logical section is left ill-defined as it is difficult to define precisely. Identifying the high level actions or steps in a method and separating the logical sections by blank lines is complementary to these efforts.

2.4 Shortcomings and Remaining Issues

Information at the level between the individual statement and the whole method is not leveraged by current source code analyses. One possible reason is that the information is not easily available beyond any internal comments describing the code blocks implementing them. Instead, current source code analyses driving software maintenance tools today treat methods as either a single unit or a set of individual statements or words. Our hypothesis is that the abstraction at the level between individual statements and methods could potentially provide a new perspective for software analyses. For example, with potentially more appropriate words associated with action units, concern location tools would have more descriptive words to use; method summary generators could integrate the high level steps to create a summary in a more descriptive and succinct manner; and segmenting the code to a sequence of high level steps could greatly improve source code readability.

Chapter 3
THE NOTION OF ACTION UNITS

This chapter presents the notion of action units. Motivating examples are shown first followed by the definition of action units and the major types. At the end of this chapter, similar notions are compared with action units.

3.1 Motivating Examples

The Java method in Listing 3.2 illustrates how code blocks can represent high level actions. The method is extracted from an open source project - *comet* [21]. The developer has partitioned the method by blank lines and internal comments. The main action is building a menu, which basically contains 3 steps. The first step creates and sets up a queue menu item. The second step creates and sets up a stop menu item. The last step builds the menu. These three steps are implemented as sequences of statements, each requiring more than one statement to implement the action. The first block (Lines 3-12) initializes a `MenuItem` object and then uses it as a parameter and invokes methods to set up the object. Similarly, the second block (Lines 15-24) also initializes and sets up another `MenuItem`. The third block (Lines 26-27) creates another `MenuItem` and builds the menu. Neither the method name, `buildMenu`, nor any individual statement capture the essence of the three high level steps.

```
public void buildMenu(Menu menu) {

        // Queue
        final MenuItem itemQueue = new MenuItem(menu, SWT.PUSH);
        Messages.setLanguageText(itemQueue, "MyTorrentsView.menu.
           queue");
        Utils.setMenuItemImage(itemQueue, "start");
        itemQueue.addListener(SWT.Selection, new Listener() {
                public void handleEvent(Event e) {
                        ManagerUtils.queue(download, splash);
                }
        });
        itemQueue.setEnabled(ManagerUtils.isStartable(download));


        // Stop
        final MenuItem itemStop = new MenuItem(menu, SWT.PUSH);
        Messages.setLanguageText(itemStop, "MyTorrentsView.menu.
           stop");
        Utils.setMenuItemImage(itemStop, "stop");
        itemStop.addListener(SWT.Selection, new Listener() {
                public void handleEvent(Event e) {
                        ManagerUtils.stop(download, splash);
                }
        });
        itemStop.setEnabled(ManagerUtils.isStopable(download));

        new MenuItem(menu, SWT.SEPARATOR);
        super.buildMenu(menu);
}
```

Listing 3.1: Motivating Example 1

Consider another example in Listing 3.2 which is from the open source project of *orbisgis* [79]. The method is partitioned by blank lines and internal comments. As the method name indicates, this method refreshes xml, which contains 3 steps. The first step creates a `source` and adds the `source` to `sources`. The second step adds `dependencies` and the last step adds `dependencies` to other `sources`. These three steps are implemented as sequences of statements, each requiring more than one statement to implement the action.

```
private void refreshXml() throws DriverException {
        Source source = new Source();
        source.setName(name);
        source.setDefinition(def.getDefinition());
        sources.getSource().add(source);

        // Add this dependencies
        List<String> depNames = def.getSourceDependencies();
        List<String> referencedSources = source.getReferencedSource
            ();
        referencedSources.addAll(depNames);

        // Add dependencies to other sources
        List<Source> srcList = sources.getSource();
        for (Source src : srcList) {
                if (depNames.contains(src.getName())) {
                        src.getReferencingSource().add(name);
                }
        }
}
```

Listing 3.2: Motivating Example 2

As shown in the two examples above, a method is designed to implement a major action such as “build menu” and “refresh xml”, which are indicated by their method names. To achieve the major action of the method, the statements inside the method implement several high level steps or sub-actions of the major action. Those steps or sub-actions are implemented by sequences of statements. Each sequence implements a higher level action than any individual statement and they together achieve the method’s major action.

3.2 Defining High-level Action Units

We define an **action unit** as a code block that consists of a sequence of consecutive statements that logically implement an algorithmic step within a method body. A method body normally consists of a sequence of statements and the major action of the method consists of several steps. Therefore, the sub-sequences of statements correspond to these steps.

Consider the code fragment in Listing 3.3. The code block checks if the *projects* collection contains a project whose *identifier* is equal to the given *projectKey*. This code block contains a declaration of a boolean variable and a loop iterating over a collection. Inside the loop, there are conditional statements, variable assignment statements and a break statement. None of those individual statements can represent the high level action. Instead, they together implement the high level action.

Listing 3.3: Action unit as loop.

```
boolean found=false;
for (Project project : projects) {
  if (project.getIdentifier().equals(projectKey)) {
    found=true;
    break;
  }
}
```

In a more vivid analogy, the action unit to a method is like a paragraph to an essay. In an essay, a paragraph is a series of sentences that are organized and coherent, and are all related to a single topic. Almost every piece of writing that is longer than a few sentences is organized into paragraphs. This is because paragraphs show a reader where the subdivisions of an essay begin and end, and thus help the reader see the organization of the essay and grasp its main points.

Similar to an essay, a method contains a series of lines. Developers write code line by line and organize the related lines together to implement a small step or a sub-action of the method. The key idea behind action units is action units are the steps or sub-actions of the method. An action unit is to a method what a paragraph is to an essay.

3.3 Types of Action Units

We performed a preliminary study in which we analyzed a large number of Java methods and learned where developers think each step or sub-action is. The clues we used include the blank lines and the internal comments left by the developers. Blank lines and internal comments naturally reflect how developers partition a method into smaller sub-actions of the method. This section describes the major kinds of action units based on the study.

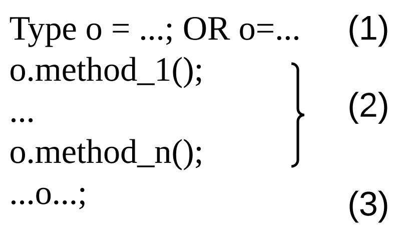
3.3.1 Object-related

We define *object-related action units* as action units that consist of only non-structured consecutive statements associated with each other by an object(s). Non-structured statements are variable declarations/assignments or method invocation statements. For example, the code fragment in Listing 3.4 is an object-related action unit. The five statements are associated with each other through the object `mainFunc` and implement a high level action *add the main function to the file.*

Listing 3.4: Action unit as a sequence.

```
// Add the main function to the file
CFunction mainFunc = CFunction.factory.create("main", "int", null);
mainFunc.appendCode(className + " obj;");
mainFunc.appendCode("cout << obj.print() << \"\\n\";");
mainFunc.appendCode("return 0;");
file.add(mainFunc);
```

In general, an object-related action unit contains 3 parts as shown in Figure 3.1. Part (1) is a declaration of or an assignment to an object reference `o`. Part (2) is one or more statements where each statement is a method call invoked on the object `o`. Part (3) is a statement that *uses* the object `o`. Specifically, *use* means `o` appears on the right-hand side of "=" if there is a declaration/assignment. If there is no declaration/assignment, then `o` appears as an argument of a method call and the method call is not invoked on `o`.



```
Type o = ...; OR o=...
o.method_1();
...
o.method_n();
...o...;
```

Figure 3.1: General format of object-related action units.

Each of the three parts of the action unit is optional. For example, Listing 3.5 and 3.6 are two examples of object-related action units. Listing 3.5 does not contain

Part(3), and Listing 3.6 does not contain Part (1) and (3).

Listing 3.5: Object-related action unit without Part (3).

```
Iterator[] classes = new Iterator[4];
classes[0] = sNode.eIterator("class");
classes[1] = sNode.eIterator("subclass");
classes[2] = sNode.eIterator("joined");
classes[3] = sNode.eIterator("union");
```

Listing 3.6: Object-related action unit without Part (1) and (3).

```
bs.setBrowseIndex(bi);
bs.setOrder(SortOption.DESCENDING);
bs.setResultsPerPage(Integer.parseInt(count));
bs.setBrowseContainer(dso);
```

3.3.2 Loop

In programming, a loop contains a sequence of instructions that is continually repeated until a certain condition is reached. Typically, the repeated action is a process, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. We refer the immediately preceding lines that initialize or define a variable that is used in the condition controlling the loop as preamble. The loop statement with its preamble together implement a high level action and we call those action units *loop action units*.

Consider the `for` loop in Listing 3.7. The high level action of this code fragment is deleting all the bad files that exist in the bad file array. The loop iterates over the array, uses each element to create a file and deletes the file if the file exists. The preamble `badFiles` and the loop together implement the high level action.

Listing 3.7: Action unit as a loop - file example.

```
String[] badFiles={CaptureActivity.OSD_FILENAME + ".gz.download",
    CaptureActivity.OSD_FILENAME + ".gz",CaptureActivity.
    OSD_FILENAME};
for (String filename : badFiles) {
  File file=new File(tessdataDir,filename);
  if (file.exists()) {
    file.delete();
  }
}
```

Consider another example in Listing 3.8. This `while` loop is a common implementation of a binary search. The loop finds the position of a target value within a sorted collection. It compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until the loop breaks. The statements of the loop together implement a binary search. Similarly, the loop in Listing 3.9 implements a bubble sort.

Listing 3.8: Action unit as a loop - search example

```
while(i<j){
   int mid = (i+j)/2;
   if(list.get(mid) < num){
       i=mid+1;
   }else{
       j=mid;
   }
}
```

Listing 3.9: Action unit as a loop - sort example

```
// sort : best first in resulting list
for (int i=0; i < (numPhrases - 1); i++) {
  for (int j=i + 1; j < numPhrases; j++) {
    if (scores[i] < scores[j]) {
      float x=scores[i];
      scores[i]=scores[j];
      scores[j]=x;
    }
  }
}
```

3.3.3 Conditional

Conditional statements perform different computations, or actions, depending on whether a programmer-specified boolean condition evaluates to true or false. A sequence of statements composing a conditional statement often together implement a higher level action. In Listing 3.10, `width` is given a value initially and then based on comparison with a minimum frame width its value is updated accordingly. The statements composing the whole conditional body together set the value for `width`. Similarly, `height`'s value is set in the next conditional statement. Therefore, we define *conditional action unit* as condition statements with the immediately preceding lines that initialize or define a variable that is used in the condition controlling the conditional statement. We call those immediately preceding lines as preamble.

Listing 3.10: Action unit as a conditional.

```
int width = screenResolution.x * 3/5;
if (width < MIN_FRAME_WIDTH) {
    width = MIN_FRAME_WIDTH;
} else if (width > MAX_FRAME_WIDTH) {
    width = MAX_FRAME_WIDTH;
}

int height = screenResolution.y * 1/5;
if (height < MIN_FRAME_HEIGHT) {
    height = MIN_FRAME_HEIGHT;
} else if (height > MAX_FRAME_HEIGHT) {
    height = MAX_FRAME_HEIGHT;
}
```

3.3.4 Exception Handling

In Java, an exception (or exceptional event) is a problem that arises during the execution of a program. When an exception occurs, the normal flow of the program is disrupted and the program terminates abnormally, which is not recommended, therefore exceptions need to be handled by the program. An exception can occur for many different reasons such as invalid data, non-exist files, and network disconnection. To handle the code that may cause an exception, Java introduces the `try, catch` and `finally` mechanisms. The `try` block contains a sequence of statements that implement a high-level action involved with a potential exception, and the `catch` and `finally` blocks catch and process the potential exception, respectively. We call the `try, catch` and `finally` block *exception handling action units*.

Consider the code in Listing 3.11. The first line initializes `file` which is used in the `try` block. The `try` block then reads all bytes from the `file`, followed by the `catch` block. The code fragments together (Line 1-8) implement the higher level action "read all bytes from the given file" with the potential exceptions handled.

Listing 3.11: Action unit as exception handling.

```
1 File f = new File(filename);
2 try {
3     byte[] bytes = Files.readAllBytes(f.toPath());
4 } catch (FileNotFoundException e) {
5     e.printStackTrace();
6 } catch (IOException e) {
7     e.printStackTrace();
8 }
```

3.4 Summary

In summary, there are two major categories of action units. One type is object-related and the other type is related to structured statements. We refer to the set of structured statements {Switch, While, If, For, Try, Do} as SWIFT statements. An extended-SWIFT action unit (E-SWIFT) includes a SWIFT nest extended with the preamble statements.

In addition to these AUs, there are other types of sequences of code that are like action units in that they serve a particular purpose. One such example would be a sequence of statements, often found in the beginning of methods, whose primary purpose is to declare and/or initialize variables. A somewhat similar situation arises with statements found at the end of methods which release resources (such as close a database connection or clear up a collection) after the main task of the method has been completed. In addition we can consider single statements found in methods that stand by themselves. That is, they appear not to be connected to the action unit immediately before and after them. Yet, since they will be performing an action, we could consider them as individual action units. Several return statements appear to be of this kind. We do not focus on these types of code sequences, as the detection of the actions they perform, while crucial, does not appear to be very interesting.

3.5 Syntactically-similar Statement Sequences

In addition to object-related and E-SWIFT action units, we also found that statements with similar syntax are frequently grouped by blank lines. We name such sequences Syntactically-Similar segments (SynS). One example is that a method often starts with a set of declarations or initializations of some variables. Those variables are used in the following main logic of the method that performs the key action of the method. When necessary, those declarations or initialization can also be placed in the middle of the method code. These declarations or initializations are a pre-step of other steps in the method, and they prepare the variables required in the later steps or sub-actions. Listing 3.12 shows a method starting with a sequence of declaration statements. `dataLength` and `pageLength` form the preamble for this method. Similarly, there are a lot of method calls grouped together without having an object associated with each other. The SynS sequences not fall into our definition of action unit, Chapter 6 considers SynS segments when dealing with blank lines.

Listing 3.12: Method with a preamble.

```
public void decode(byte[] header, ByteBuffer buffer) throws
    IOException{
     int dataLength;
     int pageLength;

     this.parametersSavable = ((header[0] >>> 7) & 0x01) == 1;
     this.subPageFormat = ((header[0] >>> 6) & 0x01) == 1;
     this.pageCode = (byte) (header[0] & 0x3F);

     if (this.subPageFormat){
          this.subPageCode = header[1];
          pageLength = ((int) header[2] << 8) | header[3];
          dataLength = pageLength - 2;
     }else{
          this.subPageCode = 0;
          pageLength = header[1];
          dataLength = pageLength - 2;
     }

     DataInputStream dataIn = new DataInputStream(new
         ByteBufferInputStream(buffer));
         decodeModeParameters(dataLength, dataIn);
}
```

3.6 Comparison to Similar Notions

While action units have not been previously defined, previous reseaerchers have defined similar notions in software engineering.

There is early work on “beacons” [13, 23], where a beacon can be a well-known coding pattern (e.g., 3 lines for swapping array elements), meaningful identifiers, program structure, or comment statements, that signal some specific functionality to help the code reader. Beacons represent names given to a visually recognizable pattern, such as swap. There is no precise definition and there are no techniques presented to automatically recognize them.

Sridhara et al. [105] developed a technique to automatically identify groupings of statements that collectively implement high level actions and then synthesize a succinct natural language description to express each high-level abstraction. A *high-level action* is defined to be a high-level abstract algorithmic step of a method. Their approach finds blocks that correspond to a manually predefined set of templates that represent high-level actions. Their goal was to identify these high level actions to improve their method summary generator. Our notion of action units extends the idea of high-level action and addresses a wider range of high-level actions.

An action unit is essentially a code fragment. Yoshida et al. [128] proposed a cohesion metric approach to dividing source code into functional segments in the form of code fragments to improve maintainability. The code fragments that have a cohesion metric above a certain threshold are detected and presented as functionalities. The code fragments identified as functionalities are not necessarily consecutive sequences of statements and do not represent high level actions of the method. For example, all code fragments that are related to “reading protein data” in a method are recognized as one functionality.

Allamanis and Sutton presented an approach to mine code idioms from source code [7]. A code idiom is a code fragment that recurs in many projects. They present a wide range of evidence that the resulting idioms are semantically meaningful, demonstrating that they do indeed recur across software projects and that they occur more frequently in illustrative code examples collected from a Q&A site. In contrast to code idioms, action units are more specific to a local method and they do not necessarily occur in many projects.

Some compiler-based tools work at the basic block level for simplified, faster analysis [4]. Gil and Maman [32] presented a catalog of 27 micro patterns. Micro patterns are similar to design patterns, but lower level abstractions. Micro patterns classify the class-level abstraction into eight categories, including idioms for a particular and intentionally restricted use of inheritance, immutability, wrapping and data management classes. Batarseh [10] proposed an approach to identify Java nano patterns, which are sets of reusable method invocations that are frequently used in Java software development. The nano patterns do not represent high level actions within a method. A nano pattern is recognized based on predefined object nature (e.g., database or file) and method functionality (e.g., insert or update). Another related notion to action units is a feature or concern, which is associated with the user-visible functionality of the system [85]. Features are bigger than action units and typically scattered across methods.

Margulieux et al. have shown that subgoal labeled worked examples improve problem solving performance in programming [66]. Subgoals are the building blocks of procedural problem solving. Each subgoal contains one or more steps. Subgoal labels are believed to be effective for learning because they visually group the steps of worked examples into subgoals and meaningfully label those groups. To promote deeper processing of worked examples, worked examples are formatted to encourage subgoal learning by emphasizing the subgoals of problem solving procedures to highlight the structural components of the problem solving process [20]. Action units emphasize the algorithmic steps of methods and can potentially be used to label the different parts of a method as subgoal labels.

In summary, several researchers have studied or invented ways to capture the parts of a method into features, functionalities, micro-patterns. Action units are unique in that the goal is to automatically identify the algorithmic steps of a method where each step is some consecutive sequence of statements.

Chapter 4
LOOP ACTION UNITS

This chapter presents our technique to automatically identify and describe action units implemented by loops [120]. We identified a set of features that together can determine the action of a loop and developed an action identification model by learning from a large corpus of open source projects. Our system takes loop source code as input, extracts its features and identifies the high-level action using the action identification model.

4.1 Problem and Motivation

To identify high-level actions from method code, Sridhara et al. developed a template-based technique by manually examining code [105]. While their technique identifies code fragments that implement high-level algorithmic steps, the technique is limited in the kinds of action units that it will identify, mostly based on a manually established set of templates for known multi-statement actions such as a loop construct for *compute max*. Their evaluation study indicated that only 24% of switch blocks, 40% of if-else blocks, and 15% of iterator loops implemented one of the templates.

Our goal was to develop an approach to automatically identify high-level actions without manually defining templates. In this chapter, we focus on identifying action units that are implemented by loop structures. Loops are a major part of source code, and many algorithmic steps (e.g., count, compare pairs of elements, find the maximum) are implemented as loop structures. Loop structures lack documentation in general. Our study of 14,317 projects revealed that internal comments on loops occur less than 20% of the time. Automatically identifying loop action units could alleviate this situation.

We focus on loops that contain exactly one conditional statement; we call these structures a loop-if structure. Although this may seem very specific, we found that 26% of loops in a Java project are loop-if on average. From 14,317 open source projects extracted from GitHub [6], we counted 674,800 code fragments with this structure, indicating that each project has on average 48 loop-if structures.

The main insight behind an automated technique is as follows. A loop can be represented by a set or sequence of feature values that are stated in terms of structural and data flow elements and linguistic characteristics of the loop. We developed a model called the *action identification model* that can associate actions with loops based on a vector of new feature values. Given the action identification model, the action for any loop-if in a project can be determined by the process illustrated in Figure 4.1. The source code representation of the loop-if is analyzed to extract its representative feature vector. The action identification model is then referenced to determine the high-level action associated with the loop's feature vector.

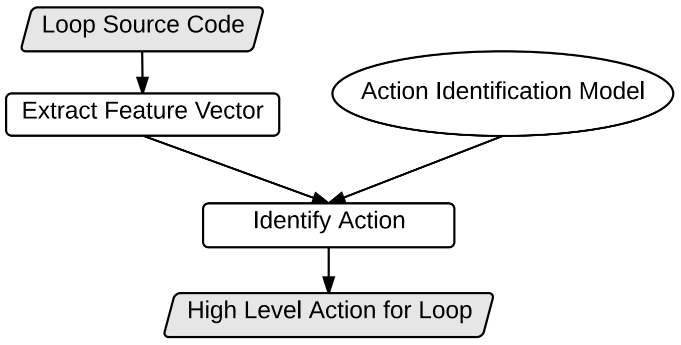


Figure 4.1: *Action Identification Process*

4.2 Characterizing Loops as Feature Vectors

In this section, we describe the features we will extract from a loop and their potential values. We begin with terminology that we use throughout this chapter.

4.2.1 Targeted Loops and Terminology

Our target is a Java loop that uses any one of the loop formats in Java: for, enhanced-for, while or do-while. We require that these loops have a single if-statement that is also the last lexical statement within the loop body. We call such a loop a **loop-if**. Such restrictions enable us to focus on identification of a single action. Currently, we also do not consider any nested loops. We collected all loops from 14,317 open source projects [6] and obtained almost 1.3 million loops. An automatic analysis of these loops revealed that 26% of them fit our loop-if criteria. In the rest of this chapter, we use loop and loop-if interchangeably to describe features and the model development process.

We analyzed a large number of loop-ifs and examined the available features that can possibly determine the action of a loop, such as structure, data flow, and names. We use the following terminology to describe the features that are used to determine the loop actions.

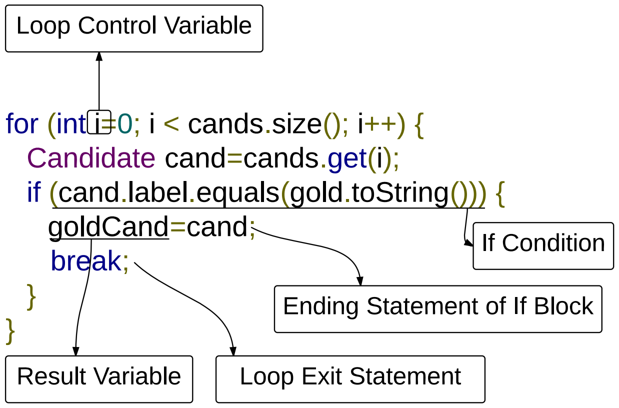
- **If condition**. The if condition refers to the conditional expression in the if statement of the loop-ifs.
- **Loop exit statement**. Loop exit statements transfer control to another point in the code by exiting when control reaches the loop exit statement, such as a break or return. Since they affect the number of iterations that are executed, we are interested in the existence of the branching statements "break", "return", and “throw” in characterizing loop-ifs.
- **Ending statement of if block**. Since the last statement inside a loop-if is an if, the last executed statement of the loop is the last statement of either the then or else block of the if statement. We are interested in the last statement on the branch that is most frequently executed, thus we approximate this as the then block unless the if condition is the null case (such as null checking, empty checking, or other validity checking), in which case, we will identify the last statement

of the else block as the ending statement of the if block. Sometimes the last executed statement of the loop is a branching statement (break, return or throw). In this case, the ending statement is designated to be the statement immediately preceding the branching statement. For the remainder of the chapter, we use ending statement to refer to the ending statement of the if block.

- **Loop control variable**. The loop condition determines the maximum number of iterations that will be executed. In `while`, `do`, and `for` loops, the loop control variable is the variable defined in the loop condition. For enhanced-for loops, the loop control variable is each element in a given collection.

- **Result variable**. The intent of the result variable is to capture the resulting value of the loop's action (if one exists). We look for the result variable in the ending statement. If the ending statement is an assignment, the result variable is the left-hand-side variable. If the ending statement is an object method invocation, it is the object that invokes the method.

Figure 4.2 shows an example loop annotated to demonstrate the terminologies used throughout the chapter.



```
for (int i=0; i < cands.size(); i++) {
  Candidate cand=cands.get(i);
  if (cand.label.equals(gold.toString())) {
    goldCand=cand;
    break;
  }
}
```

Figure 4.2: Example of terminology

4.2.2 Features

We now describe the features that we extract from a loop-if structure. These features are extracted from the ending statement or from the if condition.

4.2.2.1 Features Related to Ending Statement

Sridhara et al.[102] observed that methods often perform a set of actions to accomplish a final action, which is the main purpose of the method. Similarly, in our analysis, the ending statement also plays an important role toward indicating the action of a loop. We have identified five loop features that are related to the ending statement.

F1: Type of ending statement. The syntactic type of ending statement can be a strong indicator of what the overall loop does. We distinguish several types of ending statements for this purpose: assignment, increment, decrement, method invocation, or object method invocation. Further, we separately distinguish assignments that are boolean assignments. The type of ending statement is important in the perspective of determining actions. For example, an increment ending statement is a strong indicator of counting.

```
// Count the number of selected methods
int c=0;
for (int i=0; i < controls.length; i++) {
  if (controls[i].getChecked()) {
    c++;
  }
}
```

F2: Method name of ending statement method call. When the ending statement is a method invocation, the verbs comprising the method name often reflect the loop's actions. Two loops are not likely to do the same high-level action if one has add method and the other has remove method at the end.

We process all loop-ifs to extract the method names in ending method calls. We use Java method naming conventions to extract verbs by splitting with camel case and extracting the first word. Although many verbs are found in method names of ending statements from our data set, Table 4.1 only shows verbs eventually used in our action identification model.

F3: Elements in collection get updated.

Consider the following example:

```
for (VirtualDiskDescType vDiskDesc : disks) {
  if (diskFileId.equalsIgnoreCase(vDiskDesc.getFileRef())) {
    vDiskDesc.setCapacity(String.valueOf(bundleFileSize));
  }
}
```

The `set` method is invoked on qualified elements in the collection `disks`, which is the loop control variable of the loop. Since the result variable is the loop control variable, the method is invoked on every element that satisfies the criteria. But if the result variable is not the loop control variable, that is not the case. So this feature has the potential to differentiate between different actions. We set F3=1, when the result variable is the loop control variable; otherwise, F3=0. For example, the following loop has F3=0, because the loop control variable (`paramName`) is not the result variable (`res`).

```
Enumeration<String> paramNames=req.getParameterNames();
while (paramNames.hasMoreElements()) {
  String paramName=(String)paramNames.nextElement();
  if (!paramName.equals(SCRIPT_URL)) {
    res.setRenderParameter(paramName,req.getParameter(paramName));
  }}
```

F4: Usage of loop control variable in ending statement. Normally, we expect the loop control variable to appear in the ending statement, as the loop goes through a collection and uses elements in some way. We have already considered whether the loop control variable is the result variable or not (F3). Here we consider whether it is directly used or some variable derived from it is used in the ending statement. We also consider if it never appears in the ending statement.

Consider the following 2 examples. In the first loop, the loop control variable is used in the ending statement, and the loop filters out some elements and adds the qualified elements to another collection. In the second loop, the loop control variable is on the def-use chain of a variable used in the ending statement.

```java
for (String button : oldList) {
  if (newList.contains(button)) {
    mergedList.add(button);
  }
}
```

```java
for (Order order : orderList) {
  if (order.getPrice() > price) {
        String name = order.getName();
    newList.add(name);
  }
}
```

We set F4=0, when the loop control variable never appears in the ending statement; F4=1, when the loop control variable is directly used in the ending statement; F4=2, when the loop control variable is on a def-use chain to a use in the ending statement.

F5: Type of loop exit statement. It is important to know whether there is a control flow disruption (i.e., all elements in the collection are considered or not). For example, in a *find* action, there has to be a disruption of the loop. In addition, the type of disruption might be related to different actions and we want to know the type. So values for F5 are none, break, return, return boolean, return object, and throw. Intuitively, a return of boolean indicates the loop checks if there is any element in the collection that satisfied the condition, while a return of an object indicates finding the first qualified element in the collection.

The first example below shows that when the loop exit statement is a break and the second one shows that when the loop exit statement is a throw. The `throw` statement ensures that the desired condition is always satisfied; otherwise, an exception will be thrown.

```java
for (Field field : type.getDeclaredFields()) {
  if (field.isAnnotationPresent(LODId.class)) {
    idField=field;
    break;
  }
}
```

```java
for (BaseField field : tableToRemove.getFields()) {
  if (!(field.equals(tableToRemove.getPrimaryKey()) || field.
      getHidden())) {
```

```
    throw new CantDoThatException("Please remove all fields before
        removing the table");
  }
}
```

4.2.2.2 Features Related to the If Condition

We consider three features that are related to the if condition. We examine whether multiple collections are compared in the if condition, the role of result variable in the If condition, and the type of if condition expression.

F6: Multiple collections in if condition. This feature is a boolean that indicates whether multiple collections are compared in the if condition. We believe loops that manipulate multiple collections are likely to accomplish very different actions than those that manipulate only one.

Consider the following two examples:

```
// compare the MD5 hash array .
for (int i=0; i < 16; i++) {
  if (correctResponse[i] != actualResponse[i]) {
    return false;
  }
}
```

```
// Check if the user already exists
for (a=0; a < userCounter; a++) {
  if (ident[a] == tident)   return false;
}
```

In the first example, each pair of elements from the two arrays are compared. In the second example, each element is compared with a fixed value that does not change within the loop, the action is checking if the array contains a certain value.

We set F6=1 if there are two synchronized collections in the if condition; otherwise, F6=0.

F7: Result variable used in if condition. Within the if statement, the result variable will be updated. So determining whether the result variable is part of the if condition is equivalent to determining whether the current value of the result variable needs updating. For example, in finding the maximum or minimum, the current value determines whether it needs to be updated.

Consider the following two examples.

```
for (Iterator i=m_specialPages.entrySet().iterator(); i.hasNext(); )
    {
  Map.Entry entry=(Map.Entry)i.next();
  Command specialCommand=(Command)entry.getValue();
  if (specialCommand.getJSP().equals(jsp)) {
    return specialCommand;
  }}
```

```
float z=Float.MAX_VALUE;
for (int i=2; i < vertices.length; i+=3) {
  if (vertices[i] < z) {
    z=vertices[i];
  }}
```

In the second loop, the value of the result variable `z` is updated each time when the condition is satisfied, and the new value is then used in the if condition during the next iteration of the loop. The result variable in the first loop is not used in if condition.

We set F7=1, if the result variable appears in the if condition; Otherwise, we set F7=0.

F8: Type of if condition. This feature indicates the type of if condition. The if condition can be a numeric value comparison (e.g., "<" and ">"), or a user-defined method that returns a boolean value. A numeric value comparison combined with the feature that the result variable is used in the if condition is a strong indicator of finding the maximum/minimum element in a collection. We set F8=1, when the if condition is numeric value comparison; F8=2, when it is not.

4.2.2.3 Other Considered Features

We started with the loop control variable, result variable, if condition, and ending statement, and considered their various usages as different features. Many features turned out to be not helpful in determining high level actions. This determination is based on the empirical method we used for developing the action identification model. We just described those that are useful; however, many others (such as the type of loop, whether the if statement contains any else blocks, whether the loop control variable appears in the if condition, and the number of statements in the if block) were omitted. Table 4.1 details the possible values for each feature.

Table 4.1: Semantics of Feature Values

Label	Feature	Possible Values and Their Semantics
F1	Type of ending statement	0: none 1: assignment 2:increment 3:decrement 4:method invocation 5:object method invocation 6: boolean assignment
F2	Method name of ending statement method call	0:none 1:add 2:addX 3:put 4:setX 5:remove
F3	Elements in collection get updated	0: false 1: true
F4	Usage of loop control variable in ending statement	0: not used 1:directly used 2:used indirectly through data flow
F5	Type of loop exit statement	0:none 1:break 2:return 3:return boolean 4:return object 5:throw
F6	Multiple collections in if condition	0: false 1: true
F7	Result variable used in if condition	0: false 1: true
F8	Type of if condition	1: >/</>=/<= 2: others

4.2.3 From Loop to Feature Vector: An Example

A feature vector for a given loop-if is constructed by extracting the features F1 through F8 from the loop's source code representation using simple static analysis. The feature vector for the example code fragment in Figure 4.2 is:

(F1:1, F2:0, F3:0, F4:2, F5:1, F6:0, F7:0, F8:2)

F1 indicates that the ending statement is an assignment. F2 indicates there is no method name from an ending method call. F3 indicates that not every element in the collection is updated. F4 indicates that the loop control variable is on the def-use chain to a use in the ending statement. F5 indicates that the type of loop exit statement is a break. F6 indicates that there is only one collection in if condition. F7 indicates that the result variable is not used in the if condition. F8 indicates that the type of the if condition is not numeric comparison.

4.3 Developing an Action Identification Model

Our goal is to identify the action of a loop from its feature vector representation. More than one feature vector can correspond to the same action. Some of the features may be relevant to a specific action, while others may not be relevant to the same action. Therefore, our goal is to determine the combinations of features that are relevant to a specific high-level action and to find the groups of vectors that perform the same actions.

To characterize the high-level action performed by a specific feature vector, we could examine several loops corresponding to that loop feature vector that have comments associated with them. If these comments describe what the loop code is accomplishing, we can analyze the action descriptions in these comments, and associate them with the feature vector.

Thus, our first task was to automatically find loops for each feature vector that have descriptive comments associated with them. We call these comment-loop associations. We found that the verbs in the comments are not sufficient to characterize the actions. Verb phrases with the same verb can describe different actions based on their arguments. Furthermore, verb phrases with different verbs can describe the same action. For example, an action of "finding" is often described by verbs "search", "check", etc. While verbs alone from these descriptive comments are not enough, our further analysis revealed that the distribution of verbs can be a good indicator of the action of a specific loop feature vector.

Our hypothesis is that while different verbs may be used in comments to describe the action of loops with the same loop feature vector, the verbs associated with a loop feature vector are related to each other and can not be any arbitrary subset (of verbs). This hypothesis yields a new opportunity. The distribution of verbs associated with individual loop feature vectors can be the basis of clustering loop feature vectors that perform similar actions. Different sets of verbs associated with a pair of loop feature vectors clearly indicate that the actions performed by the pair are very different. For example, some vectors have associated comment verbs "check", "find", "search", "look", "try", "return", "get", "see", while other vectors have verbs "remove", "filter", "clean", "check", "clear", "prune", "delete". Clearly, they perform different actions. However, two loop feature vectors may just correspond to two different ways of programming the same action, and in this case, we should expect a similar distribution of verbs associated with them.

Thus, we cluster loop feature vectors by the distribution of verbs. Indeed, for our data set, we found that the top 100 most frequently occurring loop feature vectors cluster into just 12 groups on this basis. However, it is desirable to use a phrase to capture the action rather than using the associated set of verbs. While every step thus far is completely automated, choosing the representative verb phrase to describe the 12 groups was done manually using the procedure described in Section 4.3.5. To summarize, our approach to developing the action identification model follows the process illustrated in Figure 4.3.

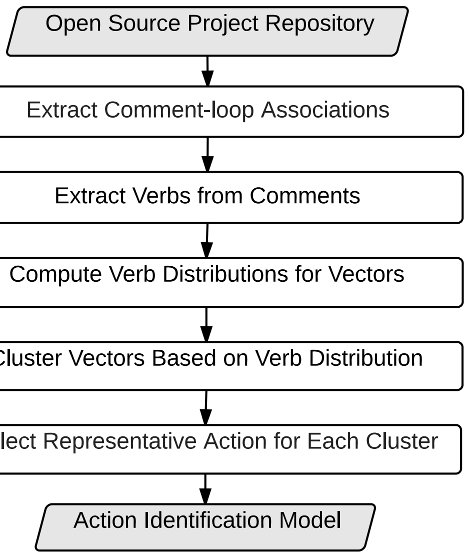

Open Source Project Repository
Extract Comment-loop Associations
Extract Verbs from Comments
Compute Verb Distributions for Vectors
Cluster Vectors Based on Verb Distribution
Select Representative Action for Each Cluster
Action Identification Model

Figure 4.3: *Process of developing the Action Identification Model*

4.3.1 Collecting Comment-Loop Associations

We extracted our comment-loop associations from the corpus of 14,317 open source Java projects originating from GitHub [6] . We used half of the repository (7,158 open source projects) to develop our action identification model, so we could use the remaining half for evaluation of the model.

To extract all available comment-loop associations from the corpus, we traverse the abstract syntax tree of each project and collect all loop-ifs with any associated comments. We identify the internal comments associated with loop-ifs by following the general convention that blank lines indicate separation of blocks of code that logically fit together and the comment immediately preceding the block is describing that code block[71, 118]. The end of a block could also be the end of a method or the end of a statement block where the comment resides. Thus, we extract any loop-if that has a preceding internal comment and ends with a blank line or end of method.

4.3.2 Extracting Verbs from Associated Comments

Previous researchers have developed classifications of comments based on the information conveyed by the comments [28, 71, 80, 92]. Comments can be descriptive, explanatory, evolutionary, and conditional. We are only interested in descriptive comments as they describe what the code following the comment is doing, at a higher level of abstraction than the code. We begin by filtering out all other kinds of comments using heuristics similar to others. From our observations, descriptive comments tend to be verb phrases, while non-descriptive comments are more likely to be sentences. However, a descriptive comment does not necessarily start with a verb; it may start with "for", "now", "if", "the", "first", etc. In each case, we still need heuristics for identifying verbs in the natural language text of the comment. For example, the comments starting with the word "for" are often in the format: "for each/all ..., verb ...". We extract the verb by selecting the first word in the second clause. Similarly, we extract verbs for other cases. After applying the heuristics, we also check if the 'extracted 'verb" can be found in the verb dictionary[124]. We base the verb extraction process on the work of Howard et al.[46].

From our repository that contains 7,158 open source Java projects, we extracted 30,089 distinct qualified comment-loop associations from which we could extract verbs. Each comment-loop association is related to a single feature vector based on the feature vector representing the loop in the comment-loop association.

4.3.3 Computing Verb Distributions for Feature Vectors

At this point, each feature vector has an associated set of comment-loop associations composed of the commented loop-ifs in the development set that are represented by that feature vector. Each comment-loop association has a comment verb, so each feature vector has an associated set of comment verbs. For example, the set of {"check", "find", "get", "search", "look for"} and the set of {"set", "update", "make", "add", "change", "reset", "replace"} contain verbs that have strong correlation.

The top 100 most frequent feature vectors cover more than 59% of loops in the data set. Thus, to give us good coverage, we generalize by performing the clustering on the top 100 most frequent feature vectors. In addition, the number of loops associated with a given feature vector starts dropping after the top 100 feature vectors, such that clustering on the basis of verb distributions would become less reliable. Most of the feature vectors have 5-6 verbs that occur repeatedly in the comment-loop associations for that feature vector. In addition, the distribution of verbs associated with each feature vector has a long tail. Thus, the verb distribution for a given feature vector is based on the distribution of the top 10 verbs most frequently associated with that feature vector.

We extract all verbs that are associated with the top 100 feature vectors. There are 230 unique verbs in total, which becomes the dimension of the verb probability distribution. For each feature vector, the verbs are transferred to a normalized probability distribution of verbs associated with that feature vector.

4.3.4 Clustering Feature Vectors Based on Verb Distribution

To help us group together the same actions, we use hierarchical clustering analysis [93]. Hierarchical clustering does not require us to specify the number of clusters. We use the complete linkage method for hierarchical clustering. This particular clustering method defines the cluster distance between two clusters to be the maximum distance between their individual components. At every stage of the clustering process, the two nearest clusters are merged into a new cluster. The process is usually repeated until the whole data set (100 loop vectors) is agglomerated into one cluster. However, since the cluster quality drops as we create larger clusters, we chose to stop the process when the distance between clusters to be merged reached a threshold value of 0.5. This resulted in 10 clusters.

Given two feature vectors, let p and q be their corresponding verb probability distributions, respectively. Each distribution will be the assignment of each verb $verb_i$, from 1 to the total number $m = 230$ unique verbs. The Euclidean distance between verb distributions p and q is:

$$d(p, q) = \sqrt{\sum_{i=1}^{m}(p(verb_i) - q(verb_i))^2}$$

Kullback-Leibler divergence and Jensen-Shannon divergence are two additional popular methods of measuring the similarity between two probability distributions and could be used.

4.3.5 Creating Action Identification Model

The last step is to create the action identification model based on the clusters. This step is manual because we want to discover and verify what action each cluster performs. This can not be automated.

Every cluster corresponds to a group of feature vectors that have some common feature values and some values different for a few features. In our development set, we manually analyzed up to 33 vectors per cluster. For any single cluster, we began by examining the common feature values and expressed them as a loop-if template. If a difference in feature value indicated a possible difference in action, we chose to divide the cluster on the basis of that feature.

For example, the major differences between members of a particular cluster were the value for feature F5 (type of loop exit statement), F7 (result variable used in if condition) and F8 (type of if condition). Since the presence or absence of loop exit is critical in the action computed by a loop (because it determines whether or not all elements in the collection will be processed), we chose to divide the cluster on the basis of whether or not the loop-if has a loop exit statement. Immediately,

Table 4.2: Action Information for 100 Most Frequent Loop Feature Vectors in Development Set

Action	# of Feature Vectors	# of Loops
find	33	5312
get	16	2967
determine	8	2192
add	11	940
remove	9	818
copy	2	714
count	3	512
max/min	4	500
ensure	2	395
set_all	4	330
set_one	2	223
compare	1	157

this resulted in two clusters that suggest the following two loop-if templates. These templates correspond to the ”min/max” and ”find” actions in Table 4.4, respectively.

```
for (loop_control_variable for a collection) {
  if (numeric comparison) {
        //result variable used in if condition
    result_variable = loop_control_variable;
   // no loop exit statement
}  }
```

```
for (loop_control_variable for a collection) {
  if (numeric / non-numeric comparison ) {
    result_variable = loop_control_variable;
    break/return/return result_variable;
}  }
```

The manual analysis of the 10 feature vector clusters led to splitting some clusters such that there are 12 distinct actions identified from the original 10 clusters associated with top 100 most frequent feature vectors. To give an insight into the amount of manual analysis, Table 4.2 shows the number of top 100 most frequent feature vectors and the total number of loops in the development set associated with each action. For each cluster, we analyzed the template and created a verb phrase description. Table 4.3 shows the 12 identified action stereotypes and their verb phrase descriptions. Table 4.4 constitutes a model where each row shows an action and its corresponding combination of feature values. Some actions appear in multiple rows

Table 4.3: Identified actions with their verb phrase descriptions

Label	**Action Phrase**
count	count the number of elements in a collection that satisfy some condition
determine	determine if an element of a collection satisfies some condition
max/min	find the maximum/minimum element in a collection
find	find an element that satisfies some condition (other than max/min)
copy	copy elements that satisfy some condition from one collection to another
ensure	ensure that all elements in the collection satisfy some condition
compare	compare all pairs of corresponding elements from two collections
remove	remove elements when some condition is satisfied
get	get all elements that satisfy some condition
add	add a property to an object
set_one	set properties of an object using objects in a collection that satisfy some condition
set_all	set a property for all objects in a collection that satisfy some condition

because the different feature value combinations could not be merged into one. For example, if a loop has combination of value 0 for F1 and 2 or 3 for F5, or value 6 for F1 and 0 or 1 for F5, then the model will label this loop with action determine.

4.4 Example of Automatic Action Identification

To exemplify the automatic action identification process shown in Figure 4.1, consider the following source code:

```
for (Subunit s : subunits) {
  if (s instanceof Department) {
    Department subDepartment=(Department)s;
    Department result=subDepartment.findDepartment(id);
    if (result != null) {
      return result;
} } }
```

With the action identification model and the given loop-if source code, the automatic identifier first extracts the features as described in Section 4.2. The extracted feature vector is (F1:1, F2:0, F3:0, F4:2, F5:2, F6:0, F7:0, F8:2). By mapping the feature vector against the action identification model in Table 4.4, the identified action is ”find”. Thus, we can identify the action as ”find an element that satisfies some condition”.

Table 4.4: Action Identification Model

Feature / Action	F1	F2	F3	F4	F5	F6	F7	F8
count	2				0			
determine	0				2,3			
determine	6				0,1			
max/min	1			1,2	0	0	1	1
find	1			1,2	1,2,4			
find	0				4			
copy	5	1	0	1	0			
ensure					5			
compare					3	1		
remove	5	5	1	1	0			
get	5	1,3	0	2	0			
add	5	2	0	1,2	0			
set_one	5	4	0	1,2				
set_all	5	4	1	1,2	0			

4.5 Evaluation

We implemented the automatic action identifier, and designed our evaluation to answer the two main questions:

- *RQ1 - Effectiveness.* How effective is the automatic action identifier?
- *RQ2 - Prevalence.* How prevalent are the actions we are able to identify in Java software?

4.5.1 Effectiveness of Automatic Action Identification

Procedure. We asked 15 human evaluators to judge the output of our prototype in identifying and describing high-level actions targeted by our technique. To reduce potential bias, we told the judges that the action descriptions were identified from different systems, and that our goal was to know which system is better.

The programming experience of this group ranges from 5 to 15 years, with a median of 9 years; 13 of the evaluators consider themselves to be expert or advanced programmers, and 4 evaluators have software industry experience ranging from 1 to 3 years. None of the authors participated in the evaluation.

For evaluation, we randomly selected 5 loops for each of the 12 actions that we are able to identify. To account for variation in human opinion, we gathered 3 separate judgments for each loop. Hence, each human evaluator evaluated 12 loops. In total, we obtained 180 independent judgments on 60 loops, by 15 developers working independently with 3 judges per loop.

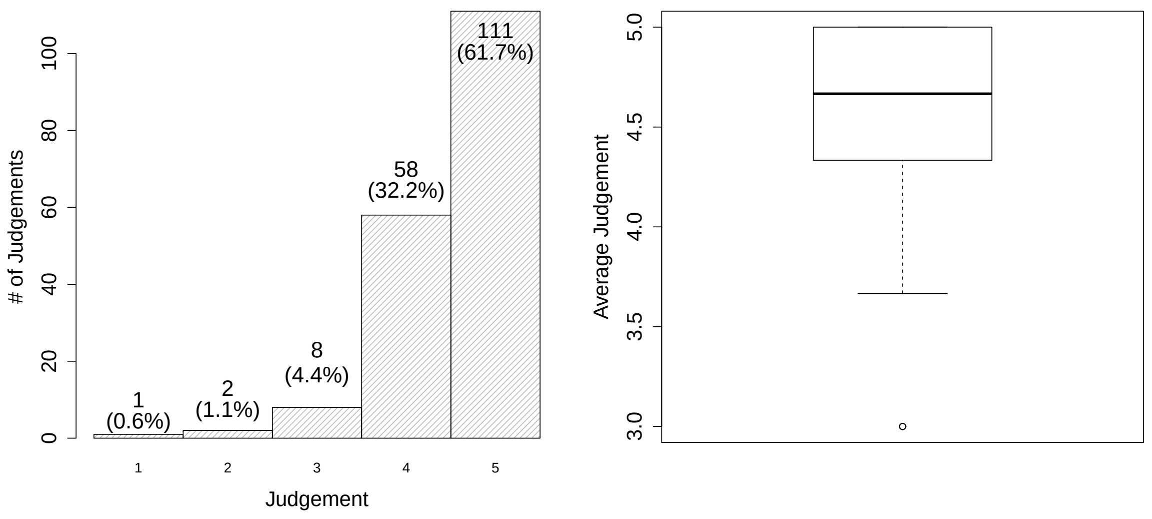
In a web interface, we showed evaluators a code fragment and asked them to read the code. Then, they were instructed to click a button to answer questions. This process let the evaluator read the code first and avoid their opinion being affected by the provided options. We asked evaluators two questions about the high-level action of the code:

1. How much do you agree that the loop code implements this action?
2. How confident are you in your assessment?

The evaluators were asked to respond to the first question via the widely used five-point Likert scale: 1:Strongly disagree, 2:Disagree, 3:Neither agree or disagree, 4:Agree, 5:Strongly agree. Similarly, evaluators were asked to respond to the second question with values 1:Not very confident, 2:Somewhat confident, or 3:Very confident.

Results and Discussion. Figure 4.4a shows the number of individual developers' responses for each point in the Likert scale for the first question. The results strongly indicate that the code fragments that we automatically identify as high-level actions are indeed viewed as high-level actions by developers. In 93.9% (169 out of 180) of responses, judges strongly agree or agree that the identified actions represent the high-level actions of the loops, within which 61.9% correspond to strong agreement. Furthermore, out of 180 responses, 176 judges are confident or very confident about their opinions.

For each of the 60 loops given to evaluators, we also computed the average opinions given by 3 evaluators for each loop. Figure 4.4b shows the box plot of average opinions for the 60 loops. The average opinions per loop range from 3 to 5, with their median being 4.33. Also 91.7% (55 of 60) of the average opinions were 4.0 or above, where 4.0 indicates agreement that the identified action represents the high-level action of the loop.



(a) **Individual judgements** (b) **Average judgements per loop**

Figure 4.4: Human judgements of identified actions (1:Strongly disagree, 2:Disagree, 3:Neither agree or disagree, 4:Agree, 5:Strongly agree)

We analyzed the three loops with the average opinion below 3: Neither agree or disagree. One interesting case is that when an if block contains many statements and there are multiple actions expressed by the statement block, the last statement itself is not sufficient to summarize the major action. Similarly, when there are multiple then clauses in the if statement and each of them does a different action, the ending statement we select from the first clause is not sufficient to represent all of them. In another case, when the last statement is an object method invocation and the object is a class, the method is then a static method. In such cases, the class that the static method is invoked on is not the object that gets updated. Instead, the static method updates the object passed into the method. We can improve our implementation to capture this case.

4.5.2 Prevalence of Identifiable Actions

Procedure. To determine the potential impact of automatically identifying the high level actions of loop-ifs, we ran the action identifier on all 7,159 open source projects that we had saved as our test data set. Cumulatively, these programs contain 9,358,179 methods, with a median of 150 and maximum of 206,175 methods per project. We gathered data on the frequency of each high level action that was automatically identified.

Results. In the test data set, there are 1.3 million loops, of which 337,294 are loop-ifs. For those loop-ifs, we were able to automatically identify 195,277 high level actions (i.e., 57.9%). The frequency distribution of each identified high level action is shown in Figure 4.5.

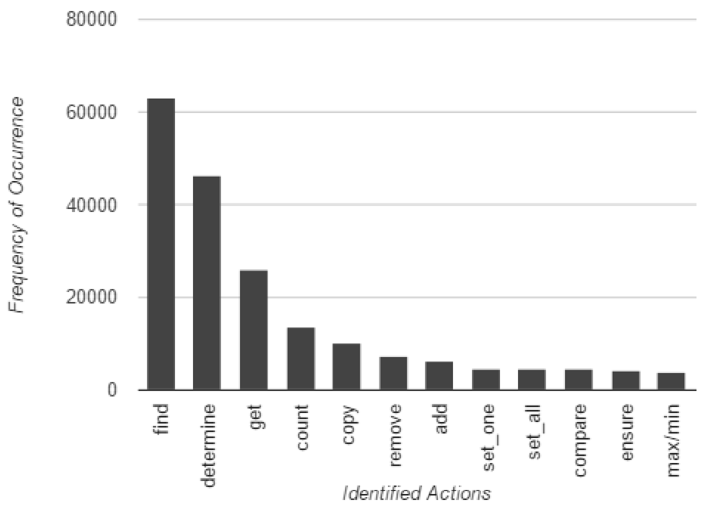


Figure 4.5: *Identified high level action distribution over 7,159 projects*

For this large corpus of Java projects, we believe these numbers are high enough to demonstrate that our action identification technique has wide applicability.

4.5.3 Summary of Evaluation Results

Human judgments by 15 developers strongly suggest indeed that they view our automatically identified descriptions as accurately expressing the high level actions of loops. Our study of the prevalence of detected high level actions in over 7000 Java open-source projects indicates that our algorithm for automatically identifying loop-ifs that implement high-level actions has wide applicability.

4.5.4 Threats to Validity

Our results may not generalize to other Java programs. To mitigate this, we used a repository that contains 14,317 projects from GitHub. We use half of the repository to develop our technique and the other half for testing. Although our extracted features are not specific to the Java programming language, they may not generalize to other programming languages. Our evaluation relies on human judges, which could be subjective or biased; to mitigate this threat, each action description was judged by three evaluators, and the evaluators were told that the output they were judging was from two different tools that we were comparing. The reported prevalence numbers could be slightly overestimated because we did not manually check all 195,277 high level actions identified for accuracy; however, our accuracy results show that it should be a good approximation.

4.6 Improving Client Tools

Another measure of this work's contribution is the potential impact on client tools for software maintenance. Here, we discuss refactoring, code search, internal comment generation, and automatic code completion.

The Java API is updated and improved regularly. On March 18, 2014 Java 8 was released. One of the major features introduced is the Stream API, which supports functional-style operations on streams of elements, such as map-reduce transformations on collections. Our technique can help developers identify loops that can be migrated to the new Stream API. For example, Figure 4.6 *(a)* shows a method that contains a loop. Our automatic action identifier can identify that the loop implements a *find* action. With the Steam API in Java 8, this action can be written in a more concise way.In addition, our technique can help developers to identify the actions for refactoring.

```
public Article getFirstJavaArticle() {
  for (Article article : articles) {
     if (article.getTags().contains("Java")) {
         return article;
     }
  }
  return null;
}
```

(a) Original Method

```
public Optional<Article> getFirstJavaArticle() {
  return articles.stream()
    .filter(article -> article.getTags().contains("
        Java"))
    .findFirst();
}
```

(b) After Refactoring

Figure 4.6: *Using our system for refactoring*

The automatic action identifier has the potential to increase the effectiveness of code search tools by providing the action phrase with the associated loop. To illustrate, in our data set of 63,265 loops that we identified automatically as ”find”, the word ”find” appears in the loop bodies only 1442 times (2% of the “find” loops).

Studies have shown the utility of comments for understanding software [110, 111]. However, few software projects contain many internal comments [52]. In our data set, less than 20% of loop-ifs are commented. Considering that some of the comments are poor or nondescriptive, the actual percentage is even less. Developers can use the automatic action identifier to generate internal comments by customizing the verb phrases using identifiers from the specific loops.

Major IDEs [51, 99] currently allow programmers to define code snippets and easily reuse them. Manually defining the templates is tedious and time consuming. When a developer wants to perform a specific action that we have identified as a high-level action in our action identification model, we can provide the code template.

4.7 Summary and Conclusions

In this chapter, we presented a novel technique that uses loop structure, data flow and word usage to automatically identify loop-if action units. Based on 15 experienced developers’ opinions in which 93.9% of responses indicate that they strongly agree or agree that the identified actions represent the high-level actions of the loops, we conclude that characterizing loops in terms of their features learned from a large corpus of open source code is enough to accurately identify high-level actions. The results also show that the technique needs only a few comment-loop associations to exist in a large corpus to support the approach.

Chapter 5
OBJECT-RELATED ACTION UNITS

In Chapter 4, we developed a model of loop action unit stereotypes without manually creating templates and used the model to identify actions for loops. While our technique advances previous research [105] that identifies the high-level actions, there are still many high-level actions remaining unaddressed. One of the most significant ones that has not been considered is *object-related action units*. In this chapter, we introduce a technique to automatically identify object-related action units and synthesize a succinct natural language description to express the actions performed by these action units [121].

5.1 Problem and Motivation

In Listing 5.1, there are three action units. The first action unit (Lines 2-7) is a loop action unit as defined in Chapter 4. The action unit determines if the given bitstream is already in a collection. The last action unit (Lines 11-14) is the type we focus on in this chapter. We call such action units object-related action units. Notice in this action unit, the statements are related to each other through an object - `mappingRow`. The purpose of the action unit is to add a newly created mapping row to the database. Line 9 is also an object-related action unit by itself. We define *object-related action units* as action units that consist of only non-structured consecutive statements associated with each other by an object or multiple objects. Non-structured statements are variable declarations/assignments or method invocation statements. Structured statements such as loops and conditionals are generally action units by themselves as discussed in Chapter 3.

Listing 5.1: A method with multiple action units.

```
public void addBitstream(Bitstream b) throws SQLException,
    AuthorizeException {
  for (int i = 0; i < bitstreams.size(); i++) {
    Bitstream existing = (Bitstream) bitstreams.get(i);
    if (b.getID() == existing.getID()) {
        return;
    }
  }

  bitstreams.add(b);

  TableRow mappingRow = DatabaseManager.create(ourContext, "
      bundle2bitstream");
  mappingRow.setColumn("bundle_id", getID());
  mappingRow.setColumn("bitstream_id", b.getID());
  database.add(mappingRow);
}
```

Our algorithm takes a Java method as input and outputs natural language descriptions associated with sequences of statements identified as object-related action units in the method. To delineate a sequence of statements that together implement an identifiable high-level action, we leverage both programming language structural information and natural language clues embedded in the developers' naming of entities. To generate natural language descriptions to express the identified high-level action, we select a representative statement and use the natural language clues embedded in the developers' naming of entities. Our automatic system involves analysis of source code only, requiring no execution information, and thus, can be applied to even incomplete and unexecutable legacy systems. Our system can be easily integrated to an IDE to provide up-to-date descriptions as the software developer begins working on a Java method. The key insight of this work is to utilize today's available large source of high-quality, open source software projects and learn common patterns of object-related action units.

More precisely, we address the following problem in this chapter:

Given a Java method M, automatically discover each object-related action unit that implements a high-level action comprising the overall algorithm of M, and accurately *express each object-related action unit as a succinct natural language description.*

Consider the code fragments in Listing 5.2. This code fragment consists of 2 AUs (Line 3-7 and Line 10-13). Notice the comments above each AU. The comments are the descriptions of the AU's high level action. They are the types of descriptions we want to generate. For example, the first description contains the action "add", the theme "number jpanel", and the secondary argument "content panel".

Listing 5.2: Action unit as a sequence.

```
 1 ...
 2 // add number jpanel to content panel
 3 numberJPanel = new JPanel();
 4 numberJPanel.setLayout( null );
 5 numberJPanel.setBounds( 16, 62, 176, 224 );
 6 numberJPanel.setBorder(new BevelBorder( BevelBorder.LOWERED ) );
 7 contentPanel.add( numberJPanel );
 8 
 9 // add one button to number jpanel
10 oneJButton = new JButton();
11 oneJButton.setText( "1" );
12 oneJButton.setBounds( 16, 16, 48, 48 );
13 numberJPanel.add( oneJButton );
14 ...
```

The two main challenges are identification of the object-related action unit and then generation of the description. There exists a range of different forms of object-related action units; however, the forms differ primarily in syntax. Finding the boundary of where the action unit begins and ends is not as trivial as identifying a loop structure. The bigger challenge is in learning how to describe the different forms of possible object-related action units. To determine an appropriate description, we need to analyze both the syntax and the data flow through the action unit. A good description should include the action, theme and other optional arguments.

5.2 State of the Art

The closest related work to action unit identification is by Sridhara et al. which automatically detects and describes high-level actions within methods [105]. One of the three major cases that technique handles is what they called "sequence as single action." The key difference between object-related action units and their "sequence as a single action" is that they consider a very restricted kind of sequence, that is, sequences of consecutive method calls that are highly similar to each other, while we broaden that scope of work to action units that are object-related. For example, lines 3-7 in Listing 5.2 might be detected as a sequence with a single action.

Other research towards identifying code fragments that are related to a given action or topic extracts topic words or verb phrases from source code [43, 68, 77]. Related research clusters program elements that share similar phrases[53]. These approaches rely solely on the linguistic information to determine the topic of the code segment, which we found is not adequate for many action units where the action is not expressed as a word within the source code explicitly.

Since our system can be used to generate internal comments for the identified high-level actions or used as a basis for method summaries, our work is related to comment generation. Sridhara et al. developed a technique to automatically generate summary comments for Java methods based on structural and linguistic clues [102]. The same authors also developed a technique to generate comments for parameters and integrate with method summaries [106]. Moreno et al. developed an approach to generate summaries for Java classes [72]. Ying and Robillard present a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary [127]. Our work goes beyond these techniques by automatically identifying action units within a method, at a granularity between individual words or statements and full method unit.

Wong et al. automatically generate comments by mining question and answer sites for code examples with related natural language text [123]. They extract code-description mappings from the question title and text for a post containing a code snippet, apply heuristics to refine the descriptions and use code clone detection to find source code snippets that are almost identical to the code-description mapping in the application to be documented. Their evaluation results showed that few clones of code snippets in the forum could be identified in real applications, indicating that the current approach has limited applicability. One possible cause is that it does not leverage an abstraction of source code such as the descriptions of object-related action units. Our work also takes the approach of directly describing the action unit based on the identified templates and focal statement identification.

There are also techniques for automatically summarizing and documenting different content or software artifacts. McBurney and McMillan presented a source code summarization technique that generates descriptions of the context of Java methods (i.e., the callers and callees) [69]. Rahman et al. presented a mining approach that recommends insightful comments about the quality, deficiencies or scopes for further improvement of the source code[88]. Panichella et al. presented an approach to mine source code descriptions from developers' communications [81]. Buse and Weimer presented an automatic technique for summarizing program changes [18]. Li et al. presented an approach to automatically document unit test cases [56]. Rastkar et al. developed a technique to automatically summarize bug reports [90]. Oda et al. developed [76] a statistical machine translation technique to generate pseudocode for Python code. To suggest method and class names, Allamanis et al. [5] introduced a neural probabilistic language model for source code that is specifically designed for the method naming problem.

There has been much research on mining code fragments across projects. Buse and Weimer developed an approach to mine API usage examples [16]. Their approach uses data flow to identify sequences of statements related to a certain API. Our approach also uses data flow. However, we focus on algorithmic steps or high-level actions, not the data flow across nonconsecutive statements, while they focus on the longest commonly used data flow chains of a target API. Allamanis and Sutton presented an approach to mine code idioms from source code [7]. There is other related work of mining code examples or patterns [30, 73, 75, 122, 133]. None of these works focus on identifying object-related action units.

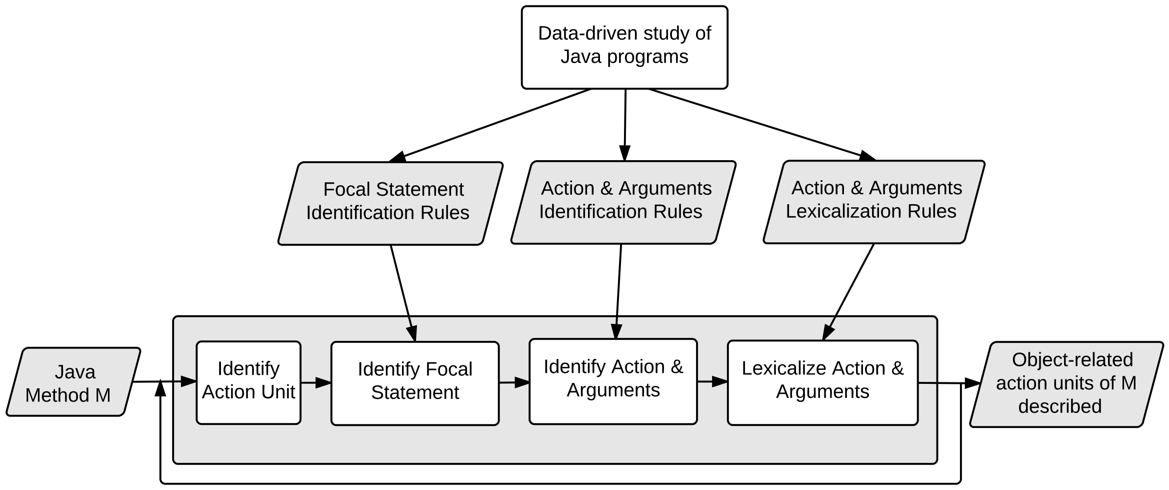

Data-driven study of Java programs
Focal Statement Identification Rules
Action & Arguments Identification Rules
Action & Arguments Lexicalization Rules
Java Method M
Identify Action Unit
Identify Focal Statement
Identify Action & Arguments
Lexicalize Action & Arguments
Object-related action units of M described

Figure 5.1: Overall process of generating descriptions for object-related action units.

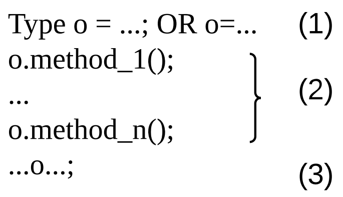
5.3 Overview of Approach

Figure 5.1 depicts the main steps of automatic identification and generation of natural language descriptions for object-related action units. Given a Java method as input, we build an abstract syntax tree (AST) and use both AST and operations performed on related objects to identify object-related action units in the method. For each object-related action unit, we identify the statement that represents the main action of the action unit, which we call the action unit's *focal statement*. The focal statement provides the primary content for the natural language description of the action unit. We identify the action and arguments that should be included in the generated description from the focal statement, and then lexicalize the action and arguments to create a natural language phrase.

The identification of focal statement, action and arguments, and lexicalization are all performed through a set of rules that we developed through a data-driven study of a large set of Java methods. The major bulk of our work is the development of these rules. We manually analyzed a large number of statement sequences from 1000 randomly selected open source projects and developed the template of object-related action unit. In our analysis, we used blank lines to provide us examples to learn the templates since Java developers conventionally use blank lines to separate methods into logically related sections[78, 118, 119]. Therefore, blank lines are an important source of learning how developers break up a method into multiple algorithmic steps. While we used blank lines to learn the templates for identifying object-related action units, our template-based approach to identification does not rely on blank lines in the methods. The algorithm will use blank lines if they exist, but can identify object-related action units despite the absence of blank lines. Sections 5.4 - 5.6 describe the development of the identification rules.

5.4 Action Unit and Focal Statement Identification

In general, an action unit contains three parts as shown in Figure 5.2. Part (1) is a declaration of or an assignment to an object reference `o`. Part (2) is one or more statements where each statement is a method call invoked on the object `o`. Part (3) is a statement that *uses* the object `o`. Specifically, *use* in this work means `o` appears on the right-hand side of "=" if there is a declaration/assignment. If there is no declaration/assignment, then `o` appears as an argument of a method call and the method call is not invoked on `o`. Each part is optional.



```
Type o = ...; OR o=...    (1)
o.method_1();
...                       (2)
o.method_n();
...o...;                  (3)
```

Figure 5.2: General format of object-related action units.

Our analysis of the same 1000 randomly selected open-source Java methods resulted in three major cases for the location of the focal statement.

Case 1: Part (3) Exists. When there is a Part (3) in the action unit, Part (1) and Part (2) prepare the object that is used in the last statement. That is, the statements in Part (1) and (2) form a necessary step to perform the action in the last statement.

In the example action unit shown in Listing 5.3, the first and second statements set the time for the `cal` object, and the last statement gets the time. The main action is in the last statement.

Listing 5.3: Focal statement is the last.

```
cal.set( 2002, 9, 14, 2, 32, 20);
cal.set( Calendar.MILLISECOND, 0);
testDate = cal.getTime();
```

This will be true even if part (1) is present, such as Listing 5.4.

Listing 5.4: Focal statement is the last.

```
Element size = new Element("size", PREMIS_NS);
size.setText(String.valueOf(...));
ochar.addContent(size);
```

Case 2: Part (3) Does not Exist. When an action unit does not have Part (3), it may have Parts (1) and (2), or only Part (2).

When an action unit has Parts (1) and (2) and all method calls in Part (2) set properties or add initial values for the object in Part (1), the action is determined by the first statement. We call the methods that set properties or initialize value for an object *setup methods*. If $method_n$ is not a setup method and $method_1 ... method_{n-1}$ are setup methods, then the action is determined by $method_n$. In Listing 5.5, the first statement creates a `PromReportFactory` object, and the next three statements set properties for the object. Therefore, the main action is in the first statement.

Listing 5.5: Focal statement is the first.

```
PromReportFactory factory = new PromReportFactory();
factory.setStoreFactory(storeFactory);
factory.setMxmlStoreFactory(mxmlStoreFactory);
factory.setEventTransformator(transformer);
```

In Java, when an operation is related to a database, a network connection, or a file writer, there normally is a statement for releasing or closing the resource at the end. These statements perform auxiliary tasks and are not the main action of an action unit. We call such statements *auxiliary statements*. If the last statement is an *auxiliary* *statement*, the last statement is ignored and the next to last statement becomes the actual “last”. The method name of an *auxiliary statement* is often `close()`, `flush()`, `commit()`, `rewind()`, `clear()` or `refresh()`. In Listing 5.6, the first statement creates an `OutputStreamWriter` object, the second statement uses the object to write data, and the last statement flushes the writer. The last statement is auxiliary, so the second statement becomes the “last”. It does something other than setting up the object, so the action is determined by the second to last statement.

Listing 5.6: Focal statement is the last.

```
// Write the data
final OutputStreamWriter writer = new OutputStreamWriter(
    connection.getOutputStream());
writer.write(data.toString());
writer.flush();
```

When there is no Part (3) or Part (1), the code block contains only a sequence of method invocations. The action units in this case can be written in the following form:

```
o.v1t1()
o.v2t2()
...
o.vntn()
```

Our preliminary studies suggest that there are basically two types of such code blocks. We consider only these two types of code blocks to be object-related action units and we discuss how the focal statement is produced for only these two types.

In the first type of code block, each method name has a verb “v” and optionally a noun “t”. If v_1 through v_{n-1} are all *setup* verbs (such as `add`, `set`, `put`, `append`, `init`, etc.) and v_n is not, then the action is determined by v_n. The reason is that the first n-1 statements set up the object that is used in the last statement to perform the main action. In the example Listing 5.7, the first two statements set properties for “g2”, and the last statement performs the “draw” action. In this case, the focal statement is the last statement.

Listing 5.7: Focal statement is the last.

```
g2.setColor(ColorUtil.mix(bgColor, Color.black));
g2.setComposite(AlphaComposite.Src);
g2.draw(backgroundRect);
```

In the second type of code block, a sequence of method calls are all setup methods or very similar to each other, and there is no focal statement of the action unit. The reason is that they all together set up properties for an object or perform an action with different arguments, so it is inappropriate to use a single statement to represent the whole action unit. This case does not fit into our overall process of the system. Describing this kind of action unit requires determining the verb and the arguments directly. The following describes three subcases that our system handles: 1) If v_1, v_2, ..., v_n are the same and t_1, t_2, ..., t_n are the same, then the action is given by the same verb and the theme is the plural form of the noun. Listing 5.9 shows such an example. 2) If v_1, v_2, ..., v_n are the same and t_1, t_2, ..., t_n are different, then the action is given by the same verb and we use the noun "properties" to represent the theme. For example, in Listing 5.8, each of the six statements sets a property for the object "clone", the action verb is obviously "set" and the theme can be represented by using the word "properties". We select the word based on developers' preference in internal comments. 3) If v_1, v_2, ..., v_n are different setup verbs, we use "set up" to represent the action. Listing 5.10 shows such an example.

Listing 5.8: Example action unit.

```
// set properties for clone
clone.setPlayerId(getPlayerId());
clone.setName(getName());
clone.setActive(getActive());
clone.setScore(getScore());
clone.setBirthday(getBirthday());
clone.setDescription(getDescription());
```

Listing 5.9: Example action unit

```
// check discs (they should be sorted by release date)
checkDisc(1, "Sally Can't Dance", "1974", EMPTY_DISC_DESCRIPTION);
checkDisc(2, "Metal Machine Music", "1975", EMPTY_DISC_DESCRIPTION);
checkDisc(3, "Rock and Roll Heart", "1976", "A sensitive and revealing look into the prince of darkness.");
```

Listing 5.10: Example action unit

```
list.setCellRenderer(legendListRenderer);
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
list.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent evt) {
                jList1ValueChanged(evt);
                refresh();
        }
});
```

Case 3: Multiple Data Flows. Case 1 and 2 are object-related action units involving only one object. From our analysis, we found that in many sequences, the statements are associated with each other by multiple objects, and those sequences perform a single high-level action. Figure 5.3 depicts the three subcases.

The first subcase is two distinct data flows appearing in a sequence. Recall each action unit has 3 parts - Part (1), (2) and (3). As shown in Figure 5.3a, the object `o1` is used in the Part (1) of the data flow of `o2`. `o1` is on the right-hand side of the declaration/assignment statement, and is used to create or get the object `o2`. The focal statement is on the chain of object `o2`. For example, in the action unit shown in Listing 5.11, the first statement creates a `file` object which is used in the second statement as an argument to create a `PrinterWriter`. In addition, the last statement is auxiliary and should be ignored. Therefore, the focal statement is the second-to-last statement.

Listing 5.11: Example of Case 3-1.

```
file = new File(secondWC, "A/B/lambda");
pw = new PrintWriter(new FileOutputStream(file));
pw.print(value);
pw.close();
```

The second subcase occurs when the first object `o1` is used to set up the object

```
Type o1 = ...; OR o1=...
o1.method_1();
...
o1.method_n();
Type o2 = ...o1...; OR o2=...o1...
o2.method_1(...);
...
o2.method_n(...);
...o2...;
```

(a) **Case 3-1.**

```
Type o2 = ...; OR o2=...
Type o1 = ...; OR o1=...
o1.method_1();
...
o1.method_n();
o2.method_1(...);
...
o2.method_n(...);
...o1...o2...;
```

(b) **Case 3-2.**

```
Type o1 = ...; OR o1=...
o1.method_1();
...
o1.method_n();
Type o2 = ...; OR o2=...
o2.method_1(...);
...
o2.method_n(...);
...o1...o2...;
```

(c) **Case 3-3.**

Figure 5.3: Templates of multiple data flows.

`o2`, as shown in Figure 5.3b. The focal statement is on the data flow chain of the object `o2`. For example, in Listing 5.12, there are two data flows. One starts with the declaration of `newNumFormat` and the other starts with the declaration of `newNumberFormats`. The data flow starting from `newNumFormat` only creates and sets up an object that is used in the data flow of `newNumberFormats`. Therefore, the data flow starting with `newNumberFormats` contains the main action of the action unit.

Listing 5.12: Example of Case 3-2.

```
List<NumberFormat> newNumberFormats = new ArrayList<
    NumberFormat>();
NumberFormat newNumFormat = new NumberFormat();
newNumFormat.setPattern("(\\d{3})(\\d{3})(\\d{4})");
newNumFormat.setFormat("($1) $2-$3");
newNumberFormats.add(newNumFormat);
```

In the third subcase, as shown in Figure 5.3c, the object o1 and o2 are declared and set up in their own Part (1) and (2). Both of the objects are used in Part (3). The focal statement is in Part (3). For example, in Listing 5.13, the first 4 statements prepare two objects, `p1` and `p2`, and the last statement check difference between the two objects. The action is in the last statement.

Listing 5.13: Example of Case 3-3.

```
LineString l1 = gf.createLineString(coords2D);
p1 = ValueFactory.createValue(l1);
LineString l2 = gf.createLineString(coords3D);
p2 = ValueFactory.createValue(l2);
checkDifference(p1, p2);
```

5.5 Identifying Action and Arguments

As defined, the focal statement contains the action and the arguments that should be included in the description of the action unit. The first step of description generation is identifying the related arguments in the focal statement. The action and arguments identification are focused on method calls. Given a method call in the syntactic form of `receiver.verbNoun(arg)`, we identify *action*, *theme*, and *secondary argument* from the four parts of the method call: `receiver`, `verb`, `Noun`, and `arg`.

We leverage observations from previous work of action and arguments identification on source code [41, 102]. To enable the identification of the four parts of a method call, identifiers must be split into component words. We use camel case splitting, which splits words based on capital letters, underscores, and numbers (e.g., "setBookmarks" would be split into "set bookmarks"), and aspects of more advanced splitting [26]. As in any system that uses linguistic information, our technique will be hindered if the naming of entities in the source code is not meaningful. We believe this requirement is reasonable, given that developers tend to choose descriptive names for program entities such as methods and types [57].

In the method call of the form `receiver.verbNoun(arg)`, the action is the `verb` of the method. The theme could be the `noun`, the `arg`, or the `receiver`. If the split `arg` shares the same last word with `Noun`, which means the `arg` is a concrete instance of `Noun`, then the theme is the `arg` and the secondardy argument is the `receiver`. For example, in `service.setContext(securityContext)`, the last word in the split method argument names is `context`. That indicates that the method argument is an instance of `Noun`, so the theme is `securityContext`. If the `arg` is not an instance of `Noun`, then the theme is `Noun` and the `arg` becomes the secondary argument. For example, in `getResource(name)`, the method argument is not an instance of `resource`, so `resource` is the theme and the `name` is the secondary argument. If there is no `Noun` or `arg` in the method call, the `receiver` is the theme. For example, in `name.trim()`, `name` is the theme. When there are multiple arguments in the method call, we use the argument that is involved with the main data flow.

If the focal statement of an action unit is a variable declaration/assignment statement, the action comes from the right-hand side (RHS) of “=”. The RHS can be a class instance creation expression or method call. We use the method verb as the action, and the variable name on the left-hand side of “=” as the theme. When the RHS of “=” is a class instance creation expression, the action is “create”.

As a full example, given the action unit in Listing 5.14 and its focal statement, and based on the rules of identifying action and arguments, the action is “set”, the theme is “information”, and the secondary argument is “config”.

Listing 5.14: Example code fragment.

```
Information info = new Information();
info.setDescription("network description");
info.setType("network");
info.setName("networklayer1.0");
info.setVersion("1.0.0");
config.setInformation(info);
```

5.6 Lexicalizing Action and Arguments

From the focal statement, we are able to identify the action and the arguments. Next, we generate natural language descriptions that include the action and the arguments related to it as the content. We leverage observations from previous lexicalization work on source code [41, 102]. Given the action and arguments, we generate a verb phrase following the template:

action theme (secondary argument).

As shown in the previous phase, the action is from the verb of the method name for a method call. As a coding convention, method names normally start with a verb. From a randomly sampled 100 focal statements, all statements contained an action verb, and our approach for action verb selection works for all cases. Identifying those verbs is straightforward. We are aware that method names do not always start with a verb even when code is written by following coding conventions. However, methods that do not represent actions are highly unlikely to be the focal statement of an object-related action unit [41]. For example, `actionPerformed(ActionEvent e)` is a call back method that can not be a focal statement.

On the other hand, describing the secondary argument and their connection to the action and theme is usually very challenging. One issue is the connection through prepositions. We examined developers' comments that start with one of the most frequently used verbs to see which ones are used. We analyzed the 100 most commonly used verbs. Those 100 verbs cover 93.1% of all focal statements. We developed templates for those verbs. For example, add ... to ..., set ... for ..., put ... to ..., etc. For the rest, we use "using". For example, for `dao.save(document)`, we generate "save document using dao".

Listing 5.15: Focal statement is not enough.

```
TaskPropertiesDialog dialog = new TaskPropertiesDialog(
    getShell());
dialog.setResource(resource);
dialog.open();
```

The second challenge is using a phrase to describe the theme and the secondary argument. From the focal statement, we identify the action and theme. However, using words from the focal statement does not lead to a descriptive phrase for the theme and the secondary argument. Consider the example in Listing 5.15. The theme is "dialog" which is located in the focal statement. We could simply generate a phase "open dialog". However, this gives us very little information about the dialog. We need to get more information for the argument if we want to generate a more descriptive phrase. By conducting a preliminary study, we found that the description can benefit from the information elsewhere in the action unit. For example, it can be the variable declaration type, the type of class instance creation, or the string parameter of a setup method. Based on the preliminary study, we have developed a set of rules to improve the description for the theme and secondary argument. The remainder of this section describes the places in the code where we extract more valuable information.

Class Name. In object-oriented programming languages, a class is the blueprint from which individual objects are created. The name of the class normally provides the succinct name with whole words and avoids acronyms/abbreviations. The class name can be from the declaration name of a declaration statement or a class instance creation expression. For example, in the following action unit, the declaration type `StringBuilder` tells us that the object `b` is not just a builder by abbreviation expansion; it is a string builder.

Listing 5.16: Improve arguments with class names.

```
//create a string builder
StringBuilder b = new StringBuilder();
b.append(proxy.getNamespace());
b.append("/");
b.append(proxy.getActionName());
b.append("!");
b.append(proxy.getMethod());
```

Method Argument. When a method argument is a string literal, the string often gives the name of the object being set up or created. If the argument of a constructor method is a simple string, the string normally gives the name of the object being created. For example, in `JButton b = new JButton("Add");`, the argument is a simple string literal which provides the name of the button. If an object has a `setName()` method and the argument of the method is a string, the string provides the name of the object being set up. We use the string literal to decorate the theme or secondary argument. For example, in the following action unit, the argument of the method `setName()` gives the name of the label "user uid". Therefore, we generate "add user uid label to form panel".

Listing 5.17: Improve arguments with string literals.

```
//add user uid label to form panel
xLabel2.setBorder(javax.swing.BorderFactory.createLineBorder(new
    java.awt.Color(153, 153, 153)));
xLabel2.setCaption("User ID");
xLabel2.setCaptionWidth(105);
xLabel2.setDepends(new String[] {"selectedItem"});
xLabel2.setName("user.uid");
xLabel2.setPreferredSize(new java.awt.Dimension(0, 19));
formPanel.add(xLabel2);
```

Program Structure. Program structure provides information on how an object is created. For example, the first line in Listing 5.15 creates a new dialog, and we can provide information about the dialog by integrating the structural information. For "dialog", we add "newly created" to include such information.

To generate adequate descriptions that include more valuable information, in the first step, we add the most general information, which is from the variable name and its type name. We expand abbreviations for variable names if necessary [42]. The single character or number resulting from camel casing splitting are ignored. For example, in the variable name "xLabel1", "label" is extracted and x and 1 are ignored. In an English noun phrase, the right side of the description of a noun contains more general information than the left side. The names from the method arguments provide more specific information about the argument. So, in the second step, we prepend this information. When the focal statement is different from the variable declaration statement which has a class instance creation on its right-hand side, we add "newly created" to all of the descriptions to include this structural information. In the case of overlapping words, we only integrate words that have not been used.

5.7 Evaluation

We designed our evaluation to answer the three research questions:

- *RQ1* - How effective is the action and arguments selection?
- *RQ2* - How effective is the text generation?
- *RQ3* - What are the costs in terms of execution time of the description generator?

To evaluate the approach, we implemented the object-related action unit identification and description generator. The first two research questions are answered by two human studies. We asked 10 developers to participate in our evaluation studies. Their programming experience ranges from 5 to 10 years, with a median of 7 years; 6 of the evaluators consider themselves to be expert or advanced programmers, and 2 evaluators have software industry experience ranging from 1 to 3 years. None of the authors participated in the evaluation, and none of the participants read this chapter.

We ran our system on 5,000 open source projects that we randomly selected from a GitHub repository [116]. Cumulatively, these projects contain 6,684,407 methods, with a median of 150 and maximum of 206,175 methods per project. We randomly selected 100 object-related action units from all of the action units identified in the 5000 open source projects. Based on the distributions of the three major cases of object-related action units, 39 belong to case 1, 42 belong to case 2, and 19 belong to case 3. The 100 action units were shuffled and divided into ten groups. Those ten groups are used in both studies described below.

The remainder of this section discusses our evaluation design and analyses of the results.

5.7.1 RQ1: Effectiveness of Action and Arguments Selection

Evaluation Design. For the first evaluation, we obtained 200 independent annotations on 100 action units, by 10 developers working independently with 2 judgements per code fragment. We randomly assigned each of the 10 evaluators one group of code fragments. In a web interface, we first showed evaluators a code fragment and asked them to read the code. Then, they were instructed to answer the two questions.

- *With a single phrase, summarize the high-level action that the code fragment implements. Write your description so it is adequate to be copied from this local context and included in the method summary description located at the top of the method.*

- *Based on your description, what would you consider as the main action of this code fragment?*

The first question is used to help our evaluators to understand the code and write a complete phrase description. The second question is used for us to get the main action and arguments. When an annotator only provided a verb in the second question, we could also infer the arguments from the answer of the first question since that answer is always a phrase. To avoid biasing evaluators, we did not tell them each code fragment is an action unit. To account for variation in human opinion, each code fragment was annotated by two developers. If two human opinions disagree on a code fragment, we asked the third person to annotate that code fragment. To determine if a third annotator was necessary, we manually examined the annotations of the two developers. We considered the action, theme and secondary argument individually. In 89% of the cases, the two annotators agreed exactly on the action. This means we had to give 11 action units to a third evaluator.

With respect to the theme, the two annotators used exactly the same phrase 27 times. In another 29 cases, they used the same head noun but differed in the adjectives, for example, "texture attributes" and "attributes". We did not consider this to be a disagreement because from the responses to the two questions, it is clear that they are referring to the same target in the code. Recall this evaluation is concerned with determination of the arguments rather than their natural language description (which is considered in the next evaluation). Similarly, we considered 20 more cases as agreement even though theidescriptions used by the two annotators were different. The following example typifies this case. Consider Listing 5.18, both agreed the action was "add". One of them described the theme as "new JMenuItem object" and the other described it as "elements item". Again we see that the two annotators selected the same argument although the two phrases are very different. They were referring to the different parts of the code to describe the target, but clearly referring to the same object. So there were only 24 cases that we treated as a real disagreement between the two annotators. This was resolved by giving the 24 action units to a third annotator.

Listing 5.18: Disagreement on Argument.

```
    elementsItem = new JMenuItem("Edit Items");
    elementsItem.addActionListener(this);
    popup.add(elementsItem);
```

For the secondary argument, in 53 of the 100 instances, neither of the annotators gave a secondary object. In the remaining 47 cases, our analysis revealed that they agreed on 27 cases (19 using the same phrase and 8 using the same head words) and disagreed on 20 cases. All of these 20 disagreements were due to the fact that one of them specified the secondary argument and the other did not. Thus, we had 20 action units that required annotation by a third developer. Eventually, 45 of 100 instances included the secondary object in our final gold set.

Altogether, there were 35 action units where a third developer was asked to resolve the differences. In 34 cases, the third response indeed allowed us to create the gold set for the action, theme, and secondary argument. One action unit that we did not get agreement on action is Listing 5.19. Annotators used "create", "write", "output" as the action. We asked a fourth person to choose one of them and used the one that is selected.

Listing 5.19: Disagreement on Action.

```
_logger.info("init " + cnt + " batches");
_logger.info("init position=" + getPosition());
_logger.info("init batch=" + _batch);
```

Results and Discussion. The accuracy of our system for determining the action is 97%. There were only 3 cases where our system did not agree with the human annotated gold standard. In only 2 of the 100 cases, our system output for the theme did not agree with the annotation. Finally, there were 6 cases where there were differences between the gold set and the system output. In 3 of these cases, the system produced a secondary argument whereas the gold standard did not include one. In the 3 remaining cases, the gold standard included a secondary argument whereas our system did not produce one.

In Listing 5.20, the human majority annotation does not include the secondary argument "session", while our system picks up that argument. On one hand, the human annotation indicates that the argument "session" is not necessary to be included. On the other hand, the missing argument could be caused by the inconsistency among human developers. Our system always extracts the secondary arguments when they are available. Since the selection by our system is automatic, it never misses the argument when it is available.

Listing 5.20: Disagreement on Argument.

```
ServerMessage SS = new ServerMessage(ServerEvents.
    CurrentSnowStormPlays);
SS.writeInt(0);
SS.writeInt(0);
SS.writeInt(1);
SS.writeInt(15);
session.sendMessage(SS);
```

5.7.2 RQ2: Effectiveness of Text Generation

Evaluation Design. In the second evaluation, we evaluated the effectiveness of our text generation technique. Specifically, we assessed whether the system-generated description is as good as human-written descriptions when taken out of the context as a representative of that action unit, potentially to be used as part of a method summary. We asked humans to compare human-written descriptions against our system-generated descriptions without knowledge of how they were generated. To obtain human-written descriptions, we showed each annotator a code fragment in a web interface and asked two questions:

- *If somebody told you that the main action of this code fragment is "X", please give us a phrase that completes the description of the code's main action.* (X is the action verb given by our system)

- *If you were to take your description out of context (for example, to put as part of a method summary), would you add/modify this English phrase in any way? If yes, how?*

We gathered 100 human-written descriptions by randomly assigning 10 code fragments to each of our evaluators, for which we had generated descriptions with our prototype. We then assigned a third person to each of 10 code fragments that they had not viewed previously. To reduce bias on directly comparing two copies of descriptions (one from a human and one from our system), we showed both the system-generated description and the human-written description for each code fragment, in a random order, to each human. In a web interface, we showed the code fragment and asked the following question:

If the description is to be taken out of its context in the code to be part of a method summary, what is your agreement with the statement: The description serves as an adequate and concise abstraction of the code block's high-level action.

The evaluators were asked to respond to the above proposition via the widely used five-point Likert scale, (1) Strongly Disagree, (2) Disagree, (3) Unsure (Neither agree nor disagree), (4) Agree, (5) Strongly Agree.

	System-generated Descriptions				Human-written Descriptions			
Response	**Case 1**	**Case 2**	**Case 3**	**All**	**Case 1**	**Case 2**	**Case 3**	**All**
1:S Disagree	1	0	1	2	0	1	0	1
2:Disagree	0	1	0	1	1	1	0	2
3:Neutral	9	4	5	18	4	2	3	9
4:Agree	11	10	8	29	14	10	5	29
5:S Agree	18	27	5	50	20	28	11	59
Total	39	42	19	100	39	42	19	100

Table 5.1: Distribution of human judgements of descriptions.

Results and Discussion. Table 5.1 shows the number of individual developer responses along the Likert scale. In the 100 system-generated descriptions, humans agree or strongly agree that the description serves as an adequate and concise abstraction of the code block's high-level action 79% of the time. In 3% of the cases, humans disagree or strongly disagree. In contrast, in the 100 human-written descriptions, humans agree or strongly agree 88% of the time and disagree or strongly disagree 3% of the time. Listing 5.21 shows an instance that the humans strongly disagree. For this code fragment, our system generates "set properties." On the 5-point Likert scale, the average score of the 100 system-generated descriptions was 4.24, while the average score of the 100 human-written descriptions was 4.43. Our automatic system was rated as adequate and concise abstraction of the code block's high-level action only slightly lower than the rating of human-written descriptions.

Listing 5.21: Disagreed action unit 1.

```
setButton(context.getText(android.R.string.ok), this);
setButton2(context.getText(android.R.string.cancel), (
    OnClickListener) null);
setIcon(R.drawable.ic_dialog_time);
```

Listing 5.22: Disagreed action unit 2.

```
removeLastButton = new IcyCommandButton("Remove last", new IcyIcon
    (ResourceUtil.ICON_SQUARE_DOWN));
removeLastButton.setActionRichTooltip(new RichTooltip("Remove last
     Z slice",
        "Remove the last Z slice the sequence"));
removeLastButton.addActionListener(...);
addCommandButton(removeLastButton, RibbonElementPriority.MEDIUM);
```

Table 5.2 presents the confusion matrix for the opinions of human-written and system-generated descriptions. The cells below the diagonal are numbers of instances that our system's generated descriptions are rated better than human-written descriptions, the cells above the diagonal are numbers of instances that human-generated descriptions are rated higher than system-generated descriptions, and the cells along the diagonal are numbers of instances when the system and human-written descriptions are rated equally. Out of the 100 code fragments, our system's descriptions are rated better than or equally well with human-written descriptions in 63 cases. Although human-written descriptions are rated better than system-generated systems in 37 cases, over half of these human-written descriptions were ranked 5 (Strongly Agree) while the system-generated description for the same case was ranked 4 (Agree), indicating that the evaluators still agree that the system-generated description serves as an adequate and concise abstraction of the code block's high-level action. In addition, when we run the Wilcoxon signed-rank test to show whether there is a statistical difference between the opinions of human and system-generated descriptions, the two-tailed P=0.12088, which means the difference between human and system is not significant.

System \ Human	1	2	3	4	5
1	0	0	0	1	1
2	0	0	0	0	1
3	0	0	3	10	5
4	0	2	3	5	19
5	1	0	3	13	33

Table 5.2: Confusion matrix of system-generated scores and human-written scores

5.7.3 RQ3: Cost of Execution

The purpose of the third research question is to evaluate the execution cost of using our object-related action unit description generator. Our automatic identification and description generator for object-related action units is meant to be run offline and depends on ASTs only. To obtain an estimate of execution time, we ran our description generator on a desktop machine running Ubuntu 14.04.2 LTS with a 3.4GHz Intel Core i7-2600 CPU and 16 GB memory. Java version 1.8.0_66 was used. The description generator took 90 minutes to process 5000 randomly selected open source Java projects. Thus, on average, each project took 1.08 seconds. We believe this time cost is reasonable and our technique could easily be run on any individual project or a large number of projects.

5.7.4 Threats to Validity

Our results may not generalize to other Java programs. To mitigate this, we used a repository that contains 6000 projects from GitHub. We use 1000 projects to develop our technique and 5000 for testing. Although our analysis is not specific to the Java programming language, the evaluation results may not generalize to other programming languages. Our evaluation relies on human judges, which could be subjective or biased; to mitigate this threat, each evaluator is assigned a randomly selected set of code fragments, the evaluators were shown each code fragment and question in a random order, and each code fragment was annotated by 2 different developers.

5.8 Summary and Conclusions

In this chapter, we presented a novel technique to automatically generate natural language descriptions for object-related action units, which implement intermediary steps of a method and commonly occur in method bodies. Our evaluation results demonstrate that we can effectively identify the action, theme and secondary arguments as the content for the description. Human opinions of our automatically generated descriptions showed that they agree that the automatically generated descriptions are an adequate and concise abstraction of the code block's high-level action, and compared to human-written descriptions, they are viewed within .2 on a 5-point Likert scale (4.24 versus 4.43) in adequacy and conciseness. The ability to automatically identify and generate descriptions for object-related action units has several important applications, including generating internal comments, improving method summaries, and identifying potential refactored code fragments and naming for those refactored units.

Chapter 6

EXAMPLE APPLICATION: AUTOMATIC BLANK LINE INSERTION

This chapter explores the relation between action units and blank lines, and presents a blank line insertion tool based on recognition of action units. Our tool, SEGMENT, takes as input a Java method, and outputs the method with blank lines inserted in appropriate places.

6.1 Problem and Motivation

Blank lines help readability by visibly separating code segments into logically related segments [108]. Software engineering books [14, 95] suggest using blank lines to break up big blocks of code. Recent studies with humans judging software readability [17] even suggest that simple blank lines may be more important than comments for readability. We have also observed that developers use vertical spacing inconsistently. Integrated development environments can easily indent code based on syntax, but do not currently support automatic blank line insertion.

Blank lines are clearly not inserted into code at arbitrary locations. We hypothesize that most blank lines are used to separate individual steps of a method. Therefore, blank lines are related to action units. We use this assumption to enable blank line insertion.

6.2 State of the Art

There are readability metrics that help to identify potential areas for improvement to code readability. Through human ratings of readability, Buse et al. [17] developed and automated the measurement of a metric for judging source code snippet readability based on local features that can be extracted automatically from programs.

Among of the many features is indentation. More recently, Daryl et al. [86] formulated a simpler lightweight model of software readability based on size and code entropy, which improved upon Buse's approach. Daryl et al. observed indentation correlated to block structure to have a modest beneficial effect. Other earlier works measured program characteristics with different definitions of readability [3, 12].

Organizations employ coding standards to maintain uniformity across code written by different developers, with the goal of easing program understanding for newcomers to the code [47, 55, 108, 109, 134]. Some guidelines [36, 108, 134] clearly specify the number of blank lines that should be used to separate methods in a class and class fields. They suggest that blank lines should be used to separate different logical sections within methods. Automatic blank line insertion is complementary to these efforts.

6.3 Algorithm for Automatic Blank Line Insertion

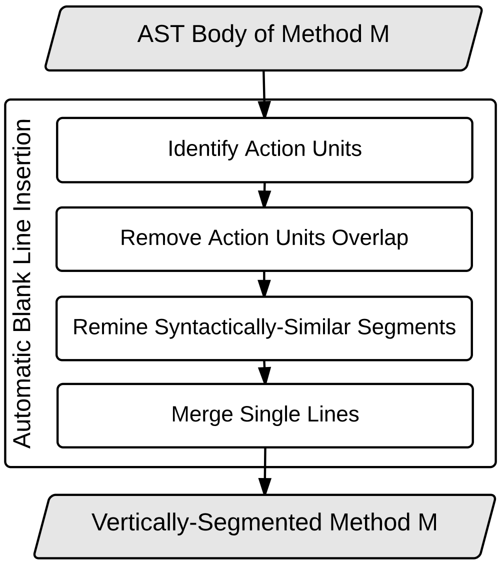

AST Body of Method M
Automatic Blank Line Insertion
Identify Action Units
Remove Action Units Overlap
Remine Syntactically-Similar Segments
Merge Single Lines
Vertically-Segmented Method M

Figure 6.1: *Automatic Method Segmentation Process*

Our approach to automatically inserting blank lines into code to segment a Java method follows the process shown in Figure 6.1. The input is the method body statements in the form of an abstract syntax tree (AST), with their associated variable definitions and uses. The first phase identifies action units. Because this phase is focused on identifying action units without concern for overlap, it sometimes results in statements at the separation points being included in two action units. The second phase, 'Remove action units overlap', decides where to place each statement involved in overlapping action units. Sometimes, the output of the first phase results in some very large syntactically-similar segments. The third phase, 'Refine syntactically-similar segments' further segments these large blocks into smaller blocks. In the final phase, we consider merging small blocks with their neighbouring blocks.

In the remainder of this section, each subsection describes the individual phases of our approach. The algorithm is applied first on the overall method, and then is applied recursively on the bodies of compound statements including conditionals, loops, try, and synchronized, when the bodies consist of four or more statements. The threshold is set to four based on our observations of developers' blank line insertions.

6.3.1 Phase I: Identify Action Units

The first phase identifies action units. In addition to the two major kinds of action units: *object-related* and *E-SWIFT*, we also group statements with similar syntax together, which we call Syntactically-Similar segments (SynS). Recall from Chapter 3, a SynS segment is a consecutive sequence of statements in which every statement belongs to the same *syntactic category*, thus related through syntax. A statement's *syntactic category* is its class of programming language syntax, which we enumerate as one of {init, declare, assign, method call, object method call, others}, as exemplified in Table 6.1. Notice that SynS segments are not action units, but we identify them in this step for blank line insertion. Furthermore, each return statement is separately treated as a block.

Table 6.1: Syntactic Categories with Examples

Syntactic Category	**Example Format**
init	type name = ...;

declare	type name;
	type name-1, name-2, ..., name-n;

assign	name = ...;
	name.property = ...;

method call	methodcall(...);
object method call	object.methodcall(...);

Others:
super method call
postfix expression
prefix expression
infix expression
throw
return
	
	super(. . .);
	i–;
	++i;
	a+b;
	throw . . . ;
	return ...;

6.3.2 Phase II: Remove Action Unit Overlap

Sometimes, two consecutive action units have an overlapping statement. The second phase focuses on making decisions on where to insert blank lines for the case of overlapping statements. Consider Listing 6.1, where lines 1 and 2 form a SynS block, and lines 2 and 3 form an object-related action unit. While the object-related action unit typically has a strong relation between its statements, we want to keep lines 1 and 2 together because they are not only both initializations (i.e., syntactically-similar), but the words in their names indicate they are also very similar semantically. This situation motivates a measure of different levels of similarity between consecutive statements (statement pairs) in a SynS block.

```
1 ActionMap am = tlb.getActionMap();
2 InputMap im = tlb.getInputMap();
3 im.put(prefs.getKey(), "close");
4 am.put("close", closeAction);
```

Listing 6.1: Overlapping Statement between SynS and DFC Blocks

6.3.2.1 Statement-pair Similarity

We define a statement-pair similarity level for consecutive statements of the same syntactic category based on word usage and naming conventions. The statement-pair similarity level is computed differently for each syntactic category due to the available information in that kind of statement. The similarity level is defined as 1, 2, or 3, with 3 being the most similar. Similarity level is defined as 0 if there is no added similarity beyond syntactic category. Table 6.2 shows the similarity level table. For example, the similarity level between a pair of object method call statements is defined as 1 if either the object or method names are similar to each other, while it is defined as level 2 if both object and method names are similar, respectively.

The statement-pair similarity level computation utilizes notions of id (name) similarity and right-hand side (RHS) similarity. We now describe our *ID-Similarity* function and *RHS-Similarity* function and then finally explain how we make decisions on which block to include the overlapping statements, based on the similarity levels in neighboring SynS blocks.

6.3.2.2 ID-Similarity Function

The ID-Similarity function returns true or false based on the similarity in appearance of two variable names, type names or method names given as input. It is called to implement the statement-pair similarity table, whenever similarity between two types, variable names, object names, or method names is needed. For the purpose of blank-line insertion, similarity of two identifiers is defined in terms of appearance, not based on meaning.

The first step is to split the input strings by camel case letters (i.e., capitalized substrings), special characters such as underscore, and numbers. We call each of the component substrings of each identifier, words, while there may be abbreviations or non-dictionary words. There are several cases to consider in computing the similarity of the identifiers. If both identifiers are single words, ID-Similarity returns true. For example, ID-Similarity is true for names `x` and `y`. If the identifiers are multi-word with the same number of words, and there is at least one exact matching word in the same position of the identifiers, ID-Similarity returns true. For example, ID-Similarity is true for `strTmp` and `strMsg`. If the identifiers are multi-word with one identifier having one more word than the other identifier and either the first or last words match, ID-Similarity returns true. For example, ID-Similarity is true for `srcPixels` and `dstInPixels`. ID-Similarity returns false for all other situations.

6.3.2.3 RHS-Similarity Function

The RHS-Similarity function takes two RHS's as input and returns true or false based on a simple similarity measure. The RHS of an assignment or initialization statement can be any of: constant, single variable name, class instance creation, infix expression, prefix expression, postfix expression, cast expression, method call, or object method call.

In general, if both RHSs are the same syntactic category (e.g., both class instance creations), then RHS-Similarity returns true. If both are either method calls or object method calls, then ID-Similarity is called on RHSs. If ID-Similarity returns true, then RHS-Similarity returns true, else false. This is illustrated below:

Input: *getPrintAction();* and *getCloseAction();*

RHS-Similarity returns: *true*

Table 6.2: Statement-pair Similarity Levels in a SynS Block

Syntactic type	**1**	**2**	**3**
init	type or RHS	type + name or type + RHS	type + name + RHS
declare	type	-	-
assign	name or RHS	name + RHS	
object-method call	object or methodname	object + methodname	
method call	methodname	-	-

6.3.2.4 Placement of Overlapping Statements

The placement of an overlapping statement among two consecutive action units depends on the kind of each consecutive action unit, among SynS, object-related, and E-SWIFT. Here, we describe the heuristics for placing the overlapping statement into the appropriate action unit of a consecutive action unit pair.

1. (object-related, object-related) Pair. If the overlapping statement S is an *init* statement, then S is placed as the first statement of the second object-related action unit. Otherwise, if S is any other kind of statement, the two object-related action units are merged into a single object-related action unit. This heuristic is based on our observation that *init* statements often start an object-related action unit, while *non-init* statements do not. For example, in Listing 6.2, Line 3 is the overlapping statement. Lines 1-3 form an object-related action unit and Lines 3-4 form another object-related action unit. We determine Line 3 to be included in the second object-related action unit because of the initialization of `r`. We would segment before line 3, and place the line in the second object-related action unit as shown in Listing 6.3.

```
1 Tree t = new Tree();
2 t.getNodeTable().addColumns(LABEL_SCHEMA);
3 Node r = t.addRoot();
4 r.setString(LABEL, "0,0");
```

Listing 6.2: (object-related object-related) Example; Before Insertion

```
1 Tree t = new Tree();
2 t.getNodeTable().addColumns(LABEL_SCHEMA);

3 Node r = t.addRoot();
4 r.setString(LABEL, "0,0");
```

Listing 6.3: (object-related object-related) Example; After Insertion

2. (SynS, object-related) / (SynS, E-SWIFT) . Normally, the object-related and E-SWIFT relations between statements is stronger than SynS relations, and the overlapping statement would be placed in the object-related or E-SWIFT action unit. For example, in Listing 6.4, Line 2 forms a SynS with Line 1 and forms an object-related action unit with Lines 3 and 4. The blank line is placed before Line 2 as shown in

Listing 6.5, because object-related relation is stronger than SynS.

```
1 Image image = ii.getImage();
2 MediaTracker mt = new MediaTracker();
3 mt.addImage(image, 0);
4 mt.waitForID(0);
```

Listing 6.4: (SynS object-related) Example; Before Insertion

```
1 Image image = ii.getImage();

2 MediaTracker mt = new MediaTracker();
3 mt.addImage(image, 0);
4 mt.waitForID(0);
```

Listing 6.5: (SynS object-related) Example; After Insertion

3. (object-related, SynS) Pairs. To determine the exceptional situations when the overlapping statement should be placed with the neighboring SynS block instead of the object-related or E-SWIFT action unit, we use the statement-pair similarity level. We say that two consecutive statements are *highly similar* when the statement-pair similarity level is 2 or 3, for those syntactic categories that have defined levels at least 2.

For statements in the method call or declare syntactic categories, we designate level 1 similarity as *highly similar*. In Listing 6.6, Lines 1-3 form an object-related action unit, but Lines 3 and 4 form a *highly similar* SynS block. Thus, Line 3 is placed in the SynS block as shown in Listing 6.7.

```
1 icon_url = getResource();
2 icon = new ImageIcon(icon_url);
3 o1.putValue(Action.SMALL_ICON, icon);
4 o2.putValue(Action.SHORT_DESCRIPTION, description);
```

Listing 6.6: (object-related SynS) Example; Before Insertion

```
1 icon_url = getResource();
2 icon = new ImageIcon(icon_url);

3 o1.putValue(Action.SMALL_ICON, icon);
4 o2.putValue(Action.SHORT_DESCRIPTION, description);
```

Listing 6.7: (object-related SynS) Example; After Insertion

4. (object-related, E-SWIFT) Pair. E-SWIFT's preamble statement may form an object-related action units with its other statements. Based on our manual analysis of blank line usage, we consider the object-related relation to be stronger than E-SWIFT, thus the overlapping statement is placed in the object-related action unit. In Listing 6.8, blank line is inserted after Line 3, despite Line 3 initializes a variable that is used in the *if* condition.

```
1 int imageWidth = image.getWidth(null);
2 int imageHeight = image.getHeight(null);
3 int imageRatio = imageWidth / imageHeight;

4 if (thumbRatio < imageRatio) {
5     thumbHeight = (thumbWidth / imageRatio);
6 } else {
7     thumbWidth = (thumbHeight * imageRatio);
8 }
```

Listing 6.8: (object-related E-SWIFT) Example; Line 3 Overlap

6.3.3 Phase III: Refine Syntactically-Similar Segments

In the first phase, Identify Action Units, SynS segments are separated based only on the syntax type of each statement. The SynS segment could be very large. This phase insert blank lines to the large SynS segments that sometimes result from the first step.

The decision to further segment a SynS segment depends on (a) the length of the block and (b) whether there exist consecutive subsequences with different similarity levels. If a block only contains 2 or 3 statements, there is no need to segment the block further. However, if the block contains more statements, it is reasonable to segment the block. The basic idea of our approach is that if a sequence contains statements that are all the same similarity to each other, there is no need to break the sequence. However, readability may be improved by segmenting into groupings that have different similarity levels.

```
in.start();
out.start();
error.start();
out.join();
error.join();
```

Listing 6.9: SynS Clustering Example; Before Insertion

```
1 in.start();
2 out.start();
3 error.start();

4 out.join();
5 error.join();
```

Listing 6.10: SynS Clustering Example; After Insertion

We designed an algorithm to cluster syntactically-similar statements based on a statement-pair similarity level. The clustering algorithm can be described as follows. The input of the algorithm is the SynS segment which contains more than 3 statements. The clustering algorithm performs a sequential scan through the SynS block with a sliding window of three statements, a, b, c. At each point in the scan, the similarity levels of each of the two pairs is computed, and the difference between the similarity levels of the two statement pairs involved in the three statements (a, b) and (b, c) is computed. If this difference in levels is nonzero, then we insert a blank line between the pair with the lowest similarity level. At the end of the algorithm execution, there will be a blank line between groupings of consecutive statements with the same similarity level.

Consider the example shown in Listing 6.9. The algorithm will produce similarity levels of {1, 0} in the second sliding window which contains Lines 2-4. Lines 2 and 3 have similarity level 1, and Lines 3 and 4 have similarity level 0. The difference is 1 which is greater than 0. Therefore, a blank line is inserted between Lines 3 and 4, as Lines 3 and 4 have lower similarity level. Listing 6.10 shows the code after insertion.

6.3.4 Phase IV: Merge Single Lines

Sometimes because we examine statement pairs individually, we create very small blocks of 1-3 statements, which may harm readability. This last phase focuses on merging very small blocks.

We currently target single-line blocks, with the goal of merging them with one of their neighboring blocks. Listing 6.11 shows an example of three statements that were single-statement blocks after the first three phases, but can be merged into a single block because they have similar RHSs. Our strategy is to look for similar features

```
JLabel label1 = new JLabel("Search␣Name:");
searchNameField = new JTextField(12);
JLabel label2 = new JLabel("Search␣Type:");
```

Listing 6.11: Example Single-line Blocks Being Merged

between the single-line block and its neighboring blocks using features summarized in Table 6.3.

6.4 Evaluation

Table 6.3: Similar Features between different kinds of statements

Syntactic Categories	**Example Format**	**Condition**
init & declare	type name = ...; type name;	type similar
init & assign	type name = ...; name = ...;	RHS or name similar

methodcall
& object-methodcall
	methodcall(parameter);
	object.methodcall(parameter);
		method call or
		param similar

We implemented the automatic blank line insertion algorithm described in Section 6.3 as a prototype tool called SEGMENT, for Java methods. Our evaluation focused on the following questions:

- How does SEGMENTcompare with developers' insertions?
- How does SEGMENTcompare with readers' notion of where blank lines should be inserted?

We designed two studies to investigate these questions. The first study compares SEGMENT-generated blank lines with original developers' usage of blank lines. Developers' usage should reflect how developers conceptualize code as different blocks. Their insertions inform us on how code is segmented into logical blocks from a developer's (writer's) perspective. However, since we randomly selected our evaluation data set from open source projects, the data set could include methods where developers may not have paid sufficient attention to blocks and blank line insertions. For this reason, we developed our second study in which blank lines are inserted by people (newcomers to the code) reading the method code. In contrast to the first study, the blocks are segmented from a reader's perspective.

Both studies involve human judges. We conducted a preliminary study to better understand the subjectivity of the blank line insertion task and how humans may differ in their opinions of where blank lines should be used. We gave 3 methods with no blank lines to 12 evaluators and asked them to insert blank lines at appropriate places. We found that at least 2 out of 3 agreed on nearly every blank line location. Based on these results, we developed our studies to have each method to be randomly assigned to 3 human judges and take the majority opinions when there was no unanimous opinion.

We engaged 15 human evaluators as our subjects, including 6 software engineers, 1 faculty member, 6 graduate students, and 2 undergraduates. Evaluators have programming experience with C++/Java ranging from 4 to 20 years, with a median of 7 years. Nine of the evaluators considered themselves as expert or advanced programmers. Six evaluators have software industry experience ranging from 1 to 10 years.

For the two studies, we chose different sets of 50 methods randomly from 7 projects (from 18186 methods). Table 6.4 provides information about the non-trivial-sized, open-source projects we used from sourceforge [100]. Although we are studying blank lines, evaluators need to read the entire method to judge blank line insertion. Thus, to avoid making the task too tedious, we selected methods ranging in size from 10 to 40 lines, neither too short for blank line segmentation nor too long.

Table 6.4: Subject projects in study. kloc: 1000 lines of code

Project	**#methods**	**kloc**
GanttProject	4956	60
Jajuk	2139	44
PlanetaMessenger	1142	22
Dguitar	1211	20
DrawSWF	2747	41
JRobin	1913	30
SweetHome3D	4078	73

The same evaluators participated in both studies. In the first study, the human evaluators examined the output of SEGMENT and the developers' usage of blank lines. Thus, to reduce the potential bias that the first study may have on human opinions in the second study, evaluators completed the second study before they started on the first study.

6.4.1 Developer-written vs. SEGMENT-inserted

In the first study, humans were presented with two copies of the 50 methods, one with SEGMENT-inserted blank lines and one with developer-written blank lines. To let evaluators easily compare SEGMENT and developer-inserted blank lines, a web-based evaluation system was developed to show each pair of method copies, left and right on the screen with highlighting by using SyntaxHighlighter [1]. To reduce possible bias, our web system randomly places the SEGMENT-inserted and developer-inserted copies on the screen, and randomly orders each evaluator's 10 methods to avoid learning effect.

To enable detailed analysis of situations, we manually categorized the differences between the blank line insertion into 4 types:

- Type 1: SEGMENT inserts a blank line; developer does not.
- Type 2: Developer inserts a blank line; SEGMENT does not.
- Type 3: Developer and SEGMENT insert a blank line at different locations, i.e., for the same group of statements, the developer inserted at some location, but SEGMENT inserts at another nearby location. For example, in Listing 6.12, if SEGMENT inserts between lines 2 and 3, while the developer inserts between lines 3 and 4, then this is called a Type 3 difference.
- Type 4: Developer and SEGMENT insert a blank line at the same location.

When there was a difference between the system and developers as in types 1-3, we asked evaluators which blank line insertion is better or whether it matters for the readability of the code. We also asked overall which method copy with blank lines was

```
Vector chars = new Vector();
byte[] aChar = new byte[1];
int num = 0;
while( ( num = in.read( aChar )) == 1 )
```

Listing 6.12: Type 3 Difference

better. When a blank line was inserted by both SEGMENT and developer in the same location (type 4), we asked if they agreed with the blank line.

The SEGMENT system inserted blank lines in the same locations as the original developers in 128 of 247 locations analyzed (i.e., 52% of the locations were Type 4). The human evaluators agreed with 127 of the common 128 placements, 99.2% of the common occurrences. In all the 119 cases where SEGMENT differed in some way from the original developer in blank line insertion (Types 1-3), the human evaluators preferred the SEGMENT system in 91 situations, 79.1% of the time. We believe these results are very encouraging. Overall in 128+91=219 cases out of 247(i.e. 88.7%), the system-generated blank lines are as good as or better than those inserted by developers. Recall these methods all had some blank lines and also do not represent cases where developers were oblivious to vertical spacing considerations.

Next, when we asked the human evaluators to judge the blank line insertion as a whole for entire methods, they reported 33 of 50 instances where SEGMENT was preferred versus 10 of 50 instances where the developer's was preferred overall, and 4 of 50 equally well between the developer and system. Thus, SEGMENT output for entire method segmentation was at least as good or better in 70% of the methods examined.

Analysis. We analyzed 24 locations where the majority of human evaluation judged developers' insertions to be better than SEGMENT. We found nearly all of these cases fall into two general cases.

Nine of twelve Type 1 cases (where SEGMENT insertion was judged to be extraneous) occur in the preamble of a method. The human judges (as well as the developers) expect to see lines of variable declarations and initializations kept together, and apparently do not believe that it is necessary to insert blank lines in this part (as long as it is not long). However, the insertion of SEGMENT does not distinguish between different general regions of a method. For example, SEGMENT separates the following two statements because there is neither similarity between them nor will they be a SynS action unit (because the first statement is init and second statement is a declare.) If we don't distinguish between init & declare statements and allow for such blocks in the preamble, nearly all these cases will be handled similar to developers.

```
boolean  bSocketOk = true;

java.io.IOException e;
```

We also examined six of the ten cases for the Type 2 differences (developer has inserted blanks but SEGMENT does not). These cases were at the beginning of the method. SEGMENT does not insert blank lines after a method's signature; more specifically, after '{' and before the method body. However, in these six cases, because of the number of parameters in the signature and the substantial preamble block, the developers inserted blank lines to separate the method signature from the body (see example below) and the evaluators agreed with this vertical spacing.

```
public void onRegisterNewUser( java.lang.String strNickName,
   java.lang.StringBuffer strNewUserId, java.lang.String
   strPasswd )  {

userDetails.setNick( strNickName );
userDetails.setPassword( strPasswd );
session.setUser( userDetails );
```

Among the remaining few cases, the issue was mostly that our current implementation does not cover certain cases, although SEGMENT's underlying approach could have covered these cases.

6.4.2 Gold Standard vs. SEGMENT-inserted

To obtain a gold standard for blank line insertions, we gave evaluators a set of Java methods without any blank lines and asked them to insert blank lines at appropriate locations. To account for variation in human opinion, we gathered 3 separate judgments for each method. Thus, we obtained 150 independent method annotations totally. To control for any learning effects, each annotator's 10 methods were shuffled and shown in different orders. Also, the 15 evaluators were randomly put into 5 groups.

For each method, we manually compared all 3 human-inserted blank line copies of a given method line by line. We defined the gold standard to be the locations where at least 2 out of 3 evaluators inserted a blank line.

The results of comparing SEGMENT's blank line insertion against this gold standard are shown in Table 6.5. The results quite strongly suggest that SEGMENT's automatic insertion agrees with the gold standard, 161/179=89.9% of the gold standard insertions are also locations where SEGMENT inserts blank lines.

Table 6.5: Automatic Blank Line Insertion Versus Gold Standard

	Human Majority	**SEGMENT-inserted**	**Percentage**
3/3 Agree	93	92	98.2%
2/3 Agree	86	69	80.2%
Total	179	161	89.9%

SEGMENT inserted 234 blank lines in total. Thus, there are 73 blank lines where the majority of human judges did not insert any blank line. However, in 44 among those positions, one of the human judges had inserted a blank line where SEGMENT did. So, there are 29 positions where SEGMENT inserted a blank line where none of the human judges did.

We analyzed these 29 cases and again observed two major kinds of situations. 13 of these cases indicate that the merging in Phase IV is not aggressive enough. Recall Phase IV merges any two or more neighboring single-line blocks where there is at least some similarity between the statements. The example below has two statements that have no similarity at all, but evaluators put them together.

```
flags |= 0x2005;

startTag( TAG_DEFINETEXTFIELD , fieldId , true );
```

The second major type of error cases is due to the fact that SEGMENT makes its decisions to insert blank lines oblivious of location in the code. Six of the 29 locations were inserted blank lines after '}' where it appears there is already sufficient vertical spacing and an additional blank line was not necessary. For example below, consider the blank line before `dispose` statement in the code. As noted before, SEGMENT inserts a blank line after the try block. However, because the `dispose` statement is a single-line block followed by just a '}' in a line by itself and another blank line, none of the human judges inserted a blank line above the `dispose` statement. We believe that Phase IV should consider merging of such single-line blocks.

```
  if (prnJob.printDialog(attr)) {
    try {
        ...
    } catch (Exception PrintException) {
        ...
    }

    dispose();
  }

  setCursor(Cursor.getPredefinedCursor());
}
```

6.5 Summary and Conclusion

To our knowledge, this is the first automatic system to insert blank lines into source code towards improving code readability and locating points for internal documentation. According to programmers who judged our generated blank lines, the automatically generated blank lines are accurate, and separate different logically-related blocks. Our first evaluation study shows that 88.7% of the time, the system-generated blank lines are as good as original developer-written ones or even better. The second evaluation study suggests that blank lines generated by the automated system match the majority of human opinions of where vertical spacing should be used.

Chapter 7
A STUDY OF SOFTWARE READERS' INTERPRETATION OF ACTION UNITS

In previous chapters, we presented our notion of action units and developed the techniques to identify actions for loop-if and object-related action units, and used action units to insert blank lines automatically into methods toward improving readability. This chapter presents our study of software readers' interpretation understanding of the concept of action units.

7.1 Research Questions

We designed our study to answer the research question: *how does our notion of action units compare with software readers' interpretation?* More specifically, we investigated several smaller questions. Do readers partition the method into non-overlapping action units or do they select action units with some statements not included and some action units overlapping? Which action units fit into the types that we identified and do any not fit into the defined types of action units? These questions enable us to understand the essential interpretation of action units by software readers and how they actually annotate action units. Therefore, we learn the essential differences between our analysis of developers' code and the general software readers' interpretation.

7.2 Methodology

We used developers in our study to find out what they consider to be action units. They were told to consider action units as "an algorithmic step, typically more than one statement in length, within a method body". We also provided them with a method and our identification of action units within them. Participants were then provided with multiple methods and asked to identify action units within these methods.

To conduct this study, we randomly selected 2000 open source projects from a repository [116] that contains 39,000 open source Java projects. Among 2,614,381 methods, we randomly selected 100 methods that each have 10-20 statements. Because we focus on the readers' perspectives of action units, blank lines were removed from each of the 100 methods. We asked 10 developers to participate in this study. Their programming experience ranges from 5 to 15 years, with a median of 9 years. Six developers have software industry experience. We randomly assigned 10 methods to each of the 10 developers and asked them to mark action units in these methods. The complete survey we used is in Appendix A.

7.3 Results and Implications

The readers indicated action units by either highlighting different portions of the method or using special marks to separate the methods. Among 100 methods, there are 382 segments annotated by our readers. We manually examined each of the 382 segments and map them to the different types of action units based on our conceptualization of these types (and not just the syntactic definition used in their identification).

The first observation is that all readers annotate action units by partitioning the whole method with every statement part of some action unit. Thus, there is no overlap of any action units and every statement of a method belongs to some action unit. Table 7.1 shows the types of action units identified by the readers along with their frequencies.

The most predominant type is object-related which occurs 144 times in the readers' identified action units. 96 out of the 144 object-related action units can be identified by our technique. The remaining 48 action units in this category contain either a short loop or a conditional statement. However, the loop or the conditional still assign or use the object as required in the definition of object-related action unit given in the template shown in Figure 3.1. Thus, although these examples will not be recognized by our method because of the presence of loop or conditional, we believe they

Table 7.1: Frequency of each type of action unit identified by readers

Action Unit Type	**Count**
object-related	144
loop	12
conditional	65
exception-handling	2
single statement	97
syntacically-similar	51
Not an AU	11
Total	**382**

still fit our concept of object-related action units. Hence, in the future, our object-related action unit templates should be extended to include structured statements controlling them.

12 selected action units fit into our definition of loop action unit. Note that the 12 loops do not include all loops found in the 100 methods used in the study. As mentioned above, some of them were included inside object-related action units. However, if we do not include these cases, all the remaining loops found in the 100 methods were selected as loop action units and our technique would be able to capture all cases of the readers highlighting. Among the 12 loops, there are 2 loop-ifs. While this proportion is different from the prevalence reported in Chapter 4, it can be noted that the numbers are few in this case.

Next, 65 selected action units fit into our definition of conditional action unit. As with loops, all other conditionals that appeared in these 100 methods were considered to be among the object related action units or loop-ifs. Our technique would be able to capture all cases.

As mentioned in Chapter 3.4 that there could be individual statements that perform a single action by themselves. Readers indicated 97 such cases. There is not clear relation between the single line and its previous or next line. Each of these single statements performs an action by themselves.Among the 97 cases, 36 are return statements and 34 are method calls with a leading verb in the method names. The others are assignment, declaration or initialization statements. Our technique would recognize those single statements as action units when there is no relation between the current statement and its previous or next statement. Also, as discussed in Chapter 3.5 that statements with similar syntax are frequently grouped by developers. We found 51 such cases from the 382 reader-annotated segments. These segments are sequences of declarations or initializations of some variables.

Thus,371 out of 382 cases fit our concept of action units and there are only 11 segments that do not. Listing 7.1, 7.2 and 7.3 show three cases where the statements in each segment do not have obvious relations with each other. Therefore, our technique would not recognize them as action units. However, if we try to consider the function of the code, we can see that the statements in each of the three segments perform a resource releasing, clean-up and initialization action respectively. We did not develop any method to identify them, but they fit into our definition of action units.

```
endDrag();
mFakeDragging = false;
```

Listing 7.1: None Action Unit 1

```
infoTA.clear();
stnTable.setBeans(new ArrayList());
stationMap.setStations(new ArrayList());
obsTable.clear();
selectedCollection = null;
```

Listing 7.2: None Action Unit 2

```
__equalsCalc = obj;
boolean _equals;
_equals = true &&
         ((this.objectId==null && other.getObjectId()==null) ||
              (this.objectId!=null &&
               this.objectId.equals(other.getObjectId()))) &&
             ((this.extension==null && other.getExtension()==null)
                 ||
              (this.extension!=null &&
               this.extension.equals(other.getExtension())));
         __equalsCalc = null;
```

Listing 7.3: None Action Unit 11

Listing 7.4, 7.5 and 7.6 show three cases where the statements in each segment are related with an object on the right-hand side of the first declaration/assignment, or an object that is the parameter of a method call. On the other hand in our templates for object-related action units, the object in question is either on the left-hand side of an assignment or the object that a method is invoked on. Our motivation for defining such a template was so that all the statements together accomplish a single high-level action. When statements are related each other by an object in their parameter, the sequence tends to have multiple actions in general. However, for the three cases here, the sequences implement a single high level action and requires more advanced technique to identify the action. In the future, we can consider them as a new type of object-related action unit and perform a deeper analysis of such cases.

```
CheckInResponse other = (CheckInResponse) obj;
if (obj == null) return false;
if (this == obj) return true;
if (__equalsCalc != null) {
     return (__equalsCalc == obj);
}
```

Listing 7.4: None Action Unit 3

```
toggle.put(sp, tog);
if (!silent) {
    sp.sendNotification("Exception␣Type␣Changed", ChatColor.GREEN +
          tog.toString(), Material.CACTUS);
}
updateExceptionPages(sp, currentPage.get(sp), sh, r);
```

Listing 7.5: None Action Unit 4

```
proxyClient = connection;
ProxyProtocolDriver driver = new ProxyProtocolDriver();
driver.connection = connection;
```

Listing 7.6: None Action Unit 8

Listing 7.7 shows a case with three statements related to each other by the object `jpdl`. In this case, `jpdl` is defined in the first statement and used in both the second and the third statement. In our object-related action unit template, we only allow the defined object to be used only in one statement. The reason is that when multiple statements use the object, the actions of those statements could be different, and therefore they perform more than one action.

```
1 final File jpdl = new File(getJpdlFile());
2 assertThat("Indicated␣input␣file␣missing.", jpdl.exists(), is(true)
      );
3 if (validateJpdl) {
4     assertThat("Not␣a␣valid␣jPDL␣definition.", JbpmMigration.
          validateJpdl(FileUtils.readFileToString(jpdl)), is(true));
5 }
```

Listing 7.7: None Action Unit 5

Listings 7.8, 7.9, 7.10 and 7.11 show the other four reader-annotated units that our approach does not recognize. Unless, there are hidden semantics based on readers' deeper understanding of the code, we believe those four cases do not form action units, since we believe they achieve multiple rather than single action. In Listing 7.8, the two statements do not have any obvious clues that would be link them together. It is unclear they should be consider as a single unit rather than been broken up further as two action units. In Listing 7.9, Lines 2-4 are related by the object `bpmn` (even not an object-related action unit by our definition). However, the first statement does not appear have any relation with the next three statements. In Listing 7.10, Lines 1,2 and 5 are related with each other by the variables `tf` and `duration`, but Lines 3 and 4 have no relation with the other three lines. In Listing 7.11, Lines 1-3 are related by the object `fcBeans`, but the last line obvious relation with the first three lines. In Listing 7.11, Lines 1-3 fits the template for an object-related action unit like Listing 7.6, but the last line does not seem to have any obvious relation with the first three lines. We cannot relate the last line with the first three line with our own understanding of the code.

```
1 actionCtx.setForwardConfig(forwardConfig);
2 return CONTINUE_PROCESSING;
```

Listing 7.8: None Action Unit 6

```
1 JbpmMigration.main(new String[] { jpdl.getPath(), XSLT_SHEET,
      getResultsFile() });
2 bpmn = new File(getResultsFile());
3 assertThat("Expected␣output␣file␣missing.", bpmn.exists(), is(true)
      );
4 assertThat("Not␣a␣valid␣BPMN␣definition.", JbpmMigration.
      validateBpmn(FileUtils.readFileToString(bpmn)), is(true));
```

Listing 7.9: None Action Unit 7

```
long tf = System.currentTimeMillis();
duration = tf - t0;
Service travelService = chor.getDeployedServiceByName(TRAVEL_AGENCY
    );
travelWSDL = travelService.getUri() + "?wsdl";
System.out.println(Utils.getTimeStamp() + "Choreography␣#" + idx +
    "␣enacted␣in␣" + duration + "␣miliseconds");
```

Listing 7.10: None Action Unit 9

```
if (fcBeans.size() == 0)
    JOptionPane.showMessageDialog(null, "No␣PointFeatureCollections␣
        found␣that␣could␣be␣displayed");
fcTable.setBeans(fcBeans);
infoTA.clear();
```

Listing 7.11: None Action Unit 10

In summary, in our study, we found that software readers' interpretation of action units is essentially in line with ours. The study also indicates that our current techniques will need to be generalized for the object-related action units.

Chapter 8
SUMMARY OF CONTRIBUTIONS AND FUTURE WORK

Nothing in the world can take the place of Persistence. Talent will not; nothing is more common than unsuccessful men with talent. Genius will not; unrewarded genius is almost a proverb. Education will not; the world is full of educated derelicts. Persistence and determination alone are omnipotent. - Calvin Coolidge (1872 - 1933)

In programming, each method typically consists of multiple high-level algorithmic steps, where an algorithmic step is too small to be a single method, but requires more than one statement to implement. Information at the level of abstraction between the individual statement and the whole method is not leveraged by current source code analyses. This dissertation focused on exploring action units as a unit of granularity between statement and method for effectively improving software maintenance tools. More specifically, we introduced the notion of action units and identified the kinds of action units based on a preliminary study. We developed techniques to automatically identify and describe the actions for loop-if and object-related action units. The action identification techniques developed for each type of action unit were evaluated through studies with human judges. Based on the results from the evaluation studies, our techniques to identify actions for loop-if and object-related action units are effective. Human opinions of our automatically generated descriptions showed that they agree that the automatically generated descriptions are an adequate and concise abstraction of the code blocks high level action. We also developed and evaluated SEGMENT, a tool that inserts blank lines into Java methods to improve code readability based on the notion of action units. According to programmers who judged our generated blank lines, the automatically generated blank lines are as good as original developer-written ones or even better, and they match the majority of human opinions of where vertical spacing should be used. Finally, we conducted a study to explore software readers' interpretation of action units. The study results indicate that our notion of action unit is in line with readers' perspective.

Overall, we have made a big step toward using the level of abstraction between individual statements and whole methods. The remainder of this chapter summarizes the dissertation and describes opportunities for future work.

8.1 Summary of Contributions

Chapter 1 motivated the need for source code analysis between the level of individual statement and the whole method. Chapter 2 presented background on text analysis for software maintenance tools and surveyed the state of the art in related domains.

Chapter 3 presented our notion of action units. We first motivated the notion of action units by using two examples. The examples show that each of the steps is implemented as a sequence of statements in the method. We then presented the definition of action units. We conducted a preliminary study in which we use blank lines and internal comments to learn the types of action units, and described the major types. Chapter 3 also describes the similar notions to action units.

Chapter 4 described our model of loop actions developed by mining loop characteristics from a large code corpus. We leveraged the available, large source of high-quality open source projects to mine loop characteristics and develop an action identification model. The loop characteristics are extracted based on the program structure and naming information of the code. We used the model and feature vectors extracted from loop code to automatically identify the high level actions implemented by loops. We evaluated the accuracy of the loop action identification and coverage of the model over 7159 open source programs. Based on 15 experienced developers' opinions in which 93.9% of responses indicate that they strongly agree or agree that the identified actions represent the high-level actions of the loops, we conclude that characterizing loops in terms of their features learned from a large corpus of open source code is enough to accurately identify high-level actions. The results also show that the technique needs only a few comment-loop associations to exist in a large corpus to support the approach.

Chapter 5 presented our approach to automatically generate natural language descriptions of object-related action units within methods. Based on a preliminary study of object-related action units from the available, large source of high-quality open source projects, we identified the statement that can represent the main action for the object-related action units, and generated natural language descriptions for these actions. We conducted two studies to evaluate our technique for action and argument selection, and natural language text generation. The comparison between action and argument selections by system and gold standard shows that our approach identifies the action, theme, and the secondary argument highly accurately. Based on the 10 experienced developers opinions in which 79% of responses indicate that they agree that the system-generated description represents the high level actions of the code fragments, we have demonstrated the feasibility of generating natural language descriptions for object-related action units.

Chapter 6 described our technique to segment a Java method into action units to improve code readability. Our study of Java methods with the original developers' blank lines showed that action units are often the blank line separated segments. Developers use blank lines to segment the different sub-action or steps of a method. However, we also realized that some blank line segments are not action units, and a statement can form an action unit with either its previous or next line(s). So we developed heuristics to handle the overlapping statements and the syntactically-similar statement sequences. Our tool, SEGMENT, takes as input a Java method, and outputs a segmented version that separates the method by vertical spacing. We reported on an evaluation of the effectiveness of SEGMENT based on developers' opinions. According to programmers who judged our generated blank lines, the automatically generated blank lines are accurate, and separate different logically-related blocks. Our first evaluation study shows that 88.7% of the time, the system-generated blank lines are as good as original developer-written ones or even better. The second evaluation study suggests that blank lines generated by the automated system match the majority of human opinions of where vertical spacing should be used.

Chapter 7 presented a human study in which we explored software readers' interpretation of action units. We asked 10 experienced developers to annotate the action units for 100 Java methods and then manually analyzed their annotations. The results show that the types of action units that readers annotated are almost the same as what we defined. In addition to the major types of action units we defined, we also found 11 out of 382 segments do not fit into our definition of any type of action unit. Those cases are mainly statements that do not have any obvious relation between each other. We listed those cases as a future work to investigate.

8.2 Future Work

The level of abstraction between the individual statements and the whole method provide a new level of information to use for software analysis. In addition to the work that we have presented, our study opens up several opportunities for future work.

8.2.1 Highlighting Consecutive Similar Action Units

From our analysis of internal comments, we observed that internal comments often highlight the differences between consecutive similar action units. Instead of using a full verb phrase to describe an action unit, developers often highlight the differences using a noun or noun phrase. In Listing 8.1, there are two *for* loops that differ from each other in several locations. One difference is in the collection's name in the iteration condition. The other differences are in the type and variable names in the statements inside of each loop. The comments highlight the difference between the two blocks, focusing on the iteration collection.

```java
// Rules
for (int i = 0; i < this.eRules.size(); i++) {
  EdRule r = this.eRules.elementAt(i);
  r.setEditable(b);
}
// Atomic
for (int i = 0; i < this.eAtomics.size(); i++) {
  EdAtomic a = this.eAtomics.elementAt(i);
  a.setEditable(b);
}
```

Listing 8.1: Consecutive Similar For Loops

In general, we have noticed that the similar action units associated with this type of comments are usually an *if* or *for*. The differences being highlighted typically come from *for* or *if* conditions and statements inside of the block. From those cases, very few times comments do actually say what exactly an action unit does. Instead, they describe the difference in a few words. Frequently, the differences are stated by using noun phrases, without mentioning the actions. From 4000 randomly selected Java projects from GitHub, we found 36,815 pairs of *for*'s are syntactically similar. Out of those similar code blocks, only 5.9% are commented. Therefore, highlighting the differences between consecutive similar action units can potentially improve internal comments for a large number of methods.

8.2.2 Internal Comment Generation

Previous research has shown that 20% of developers' comments are descriptive [101]. While our technique is a step forward towards generating natural language descriptions for source code, internal comments and our natural language descriptions are not exactly the same. In addition to the comments that highlight differences between similar consecutive action units, we have noticed that there are many other kinds of internal comments that need to be addressed in order to generate internal comments for Java methods.

8.2.3 Method Summary Generation

To automatically generate summary comment for methods, Sridhara et. al. [103] presented a generator called *genSumm*. The major idea is to select important statements and translate them to natural text. Important statements that should be included in the summary are defined as S_units. This approach is based on individual statements. However, a method often consists of multiple steps, describing individual statements other than those steps either miss important key steps or cause redundancy. By identifying and describing action units, method summary could be improved by using the new level of abstraction.

8.2.4 Exploration of the Applicability to Other Programming Languages

We conducted our research based on Java. Java is reported to be the most popular programming language in recent years, but its share is only 23.4% [87]. There are many other high-level programming languages designed for various purposes, such as C++, Python, PHP, C#, Javascript, Objective-C, R Swift, Ruby, etc. Some languages are similar to Java, and some are not. Future work includes investigating other programming languages and apply our action unit identification, description generation and blank line insertion techniques to programs written in other high-level programming languages.

8.2.5 API Code Examples

Using the application programming interfaces (API) of large software systems requires developers to understand details about the interfaces that are often not explicitly defined. Programmers often learn how to use an API by studying its usage examples. However, documentation with proper examples about the API is often incomplete or out of date. We have observed that many API examples fit into our definition of action units. We could use our action unit identification techniques to identify the related action units for a given API. In addition, our technique is able to generate natural language descriptions for action units. The identified API examples could be described in natural language, which can help developers better understand the function of the examples.

BIBLIOGRAPHY

[1] Syntaxhighlighter, 2010. [Online; accessed 1-April-2016].

[2] Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, pages 156–159. IEEE, 2010.

[3] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pages 235 –241, 2002.

[4] Alfred V Aho et al. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007.

[5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49. ACM, 2015.

[6] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013.

[7] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 472–483. ACM, 2014.

[8] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, 28(10):970–983, 2002.

[9] Anya Helene Bagge and Tero Hasu. A pretty good formatting pipeline. In *International Conference on Software Language Engineering*, pages 177–196. Springer, 2013.

[10] Feras Batarseh. Java nano patterns: a set of reusable objects. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 60:1–60:4, New York, NY, USA, 2010. ACM.

[11] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 203–206. ACM, 2011.

[12] J. Borstler, M. Caspersen, and M. Nordstrom. Toward a Measurement Framework for Example Program Quality. In *Department of Computing Science, Umea University*, 2008.

[13] R Brooks. Towards a theory of the comprehension of computer programs. *Int. J. Man-Mach. Stud.*, 18:543–554, 1983.

[14] Kim Bruce, Andrea Danyluk, and Thomas Murtagh. *Java: An Eventful Approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[15] Alexander Budanitsky and Graeme Hirst. Evaluating wordnet-based measures of lexical semantic relatedness. *Computational Linguistics*, 32(1):13–47, 2006.

[16] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE Press, 2012.

[17] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

[18] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42. ACM, 2010.

[19] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36:546–558, 2010.

[20] Richard Catrambone. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General*, 127(4):355, 1998.

[21] comet. https://github.com/alevy/comet.

[22] Anna Corazza, Sergio Di Martino, and Valerio Maggio. Linsen: An efficient approach to split identifiers and expand abbreviations. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 233–242. IEEE, 2012.

[23] Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of the 14th Annual Psychology of Programming Workshop*, London, United Kingdom, June 2002. Psychology of Programming Interest Group.

[24] Lionel E. Deimel, Jr. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, June 1985.

[25] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25:512–521, August 1982.

[26] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 71–80. IEEE, 2009.

[27] Len Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, 2000.

[28] Letha Etzkorn, Carl Davis., and Lisa Bowen. The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics*, 33:1731–1756(26), November 2001.

[29] J-R Falleri, Marianne Huchard, Mathieu Lafourcade, Clementine Nebut, Violaine Prince, and Michel Dao. Automatic extraction of a wordnet-like identifier network from software. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, pages 4–13. IEEE, 2010.

[30] Joel Galenson, Philip Reames, Rastislav Bodik, Bjorn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 653–663. ACM, 2014.

[31] Marek Gibiec, Adam Czauderna, and Jane Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[32] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–116, New York, NY, USA, 2005. ACM.

[33] A. Goldberg. Programmer as Reader. *IEEE Softw.*, 4(5):62–70, 1987.

[34] grep. http://www.gnu.org/software/grep/manual/grep.html.

[35] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE, 2013.

[36] Paul Haahr. A programming style for java, October 1999.

[37] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 35–44, 2010.

[38] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 223–226, New York, NY, USA, 2010. ACM.

[39] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[40] D. Haouari, H. Sahraoui, and P. Langlais. How good is your comment? a study of comments in java programs. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 137–146, 2011.

[41] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration.* PhD thesis, University of Delaware, 2010.

[42] Emily Hill, Zachary P Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K Vijay-Shanker. Amap: automatically mining abbreviation

expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 79–88. ACM, 2008.

[43] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242. IEEE Computer Society, 2009.

[44] Emily Hill, Lori Pollock, and K Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527. IEEE Computer Society, 2011.

[45] Rebecca Hiscott. 10 programming languages you should learn in 2014, January 2014.

[46] Matthew J Howard, Samir Gupta, Lori Pollock, and K Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386. IEEE Press, 2013.

[47] Watts Humphrey. *Introduction to the personal software process(sm)*. Addison-Wesley Professional, first edition, 1996.

[48] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE)*, pages 621–631, 2014.

[49] Chen Huo and James Clause. Interpreting coverage information using direct and indirect coverage. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–244, 2016.

[50] Stoney Jackson, Premkumar Devanbu, and Kwan-Liu Ma. Stable, flexible, peephole pretty-printing. *Science of Computer Programming*, 72(1-2):40 – 51, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[51] JetBrains. https://www.jetbrains.com/idea/help/live-templates.html, 2015. [Online; accessed 1-April-2016].

[52] Mira Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *ICSM '01: IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society.

[53] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, 2007.

[54] Dawn Lawrie and Dave Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 113–122. IEEE, 2011.

[55] Doug Lea. Draft java coding standard, February 2000.

[56] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A Kraft. Automatically documenting unit test cases. In *Proceedings of 9th IEEE International Conference on Software Testing, Verification and Validation*, April 2016.

[57] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Citeseer, 2006.

[58] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[59] Yoëlle S Maarek, Daniel M Berry, and Gail E Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering*, 17(8):800–813, 1991.

[60] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 68–77. IEEE, 2010.

[61] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[62] Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[63] Andrian Marcus and Jonathan I Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[64] Andrian Marcus and Vaclav Rajlich. Panel: Identifications of concepts, features, and concerns in source code. In *the Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM2005), Budapest, Hungary*, page 718, 2005.

[65] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[66] Lauren E Margulieux, Mark Guzdial, and Richard Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research*, pages 71–78. ACM, 2012.

[67] James H Martin and Daniel Jurafsky. Speech and language processing. *International Edition*, 710, 2000.

[68] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India Software Engineering Conference (ISEC)*, pages 113–120. ACM, 2008.

[69] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, Feb 2016.

[70] Paul W McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016.

[71] Steve McConnell. *Code Complete, Second Edition.* Microsoft Press, 2004.

[72] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32, 2013.

[73] Laura Moreno and Andrian Marcus. Jstereocode: automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 358–361. ACM, 2012.

[74] Gail Murphy, Mik Kersten, Martin Robillard, and Davor Cubranic. The emergent structure of development tasks. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conf*, Glasgow, Scotland, July 27–29, 2005.

[75] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In

Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[76] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, Nov 2015.

[77] Masaru Ohba and Katsuhiko Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, pages 1–5. ACM, 2005.

[78] Oracle. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html, 1999. [Online; accessed 1-January-2015].

[79] orbisgis. http://orbisgis.org/.

[80] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers - Taxonomies and characteristics of comments in operating system code. In *31st Intl Conf on Software Engineering (ICSE09)*, May 2009.

[81] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 63–72, 2012.

[82] Ted Pedersen, Satanjeev Banerjee, and Siddharth Patwardhan. Maximizing semantic relatedness to perform word sense disambiguation. *University of Minnesota supercomputing institute research report UMSI*, 25:2005, 2005.

[83] Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, and Wenyun Zhao. Improving feature location using structural similarity and iterative graph mapping. *Journal of Systems and Software*, 86(3):664–676, 2013.

[84] Maksym Petrenko and VáClav Rajlich. Concept location using program dependencies and information retrieval (depir). *Information and Software Technology*, 55(4):651–659, 2013.

[85] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23:1–23:34, February 2013.

[86] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 73–82, New York, NY, USA, 2011. ACM.

[87] PYPL. Popularity of programming language, Visited November 2016.

[88] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending insightful comments for source code using crowdsourced knowledge. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 81–90, Sept 2015.

[89] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Softw.*, 21(4):62–69, July 2004.

[90] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, April 2014.

[91] Darrell R. Raymond. Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '91, pages 3–16. IBM Press, 1991.

[92] P.-N. Robillard. Automating comments. *SIGPLAN Not.*, 24(5):66–70, 1989.

[93] Lior Rokach and Oded Maimon. Clustering methods. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 321–352. Springer US, 2005.

[94] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-2):143–192, 2000.

[95] Stephen R Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2004.

[96] Helmut Schmid. Probabilistic part-ofispeech tagging using decision trees. In *New methods in language processing*, page 154. Routledge, 2013.

[97] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-oriented Concerns. In *AOSD '07: 6th Intl Conf on Aspect-oriented Software Development*, New York, NY, USA, 2007. ACM Press.

[98] Bunyamin Sisman and Avinash C Kak. Assisting code search with automatic query reformulation for bug localization. In *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 309–318. IEEE, 2013.

[99] SnipMatch. http://marketplace.eclipse.org/content/snipmatch, 2015. [Online; accessed 1-April-2016].

[100] SourceForge. Sourceforge, 2011. [Online; accessed 1-April-2016].

[101] Giriprasad Sridhara. *Automatic Generation of Descriptive Comments for Java Methods*. PhD thesis, University of Delaware, 2011.

[102] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[103] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[104] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *16th IEEE Intl Conf on Program Comprehension, 2008*, 2008.

[105] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[106] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 71–80. IEEE, 2011.

[107] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. *13th Intl Workshop on Program Comprehension, 2005.*

[108] SUN. Code conventions for the java programming language, 1999. [Online; accessed 1-April-2016].

[109] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series).* Addison-Wesley Professional, 2004.

[110] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Program Languages*, 4(3):143–167, 1996.

[111] T. Tenny. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.

[112] Yuan Tian, David Lo, and Julia Lawall. Automated construction of a software-specific word similarity database. In *Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 44–53. IEEE, 2014.

[113] Scott Tilley. 15 years of program comprehension. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 279–280, Washington, DC, USA, 2007. IEEE Computer Society.

[114] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[115] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 762–771. IEEE Press, 2013.

[116] Xiaoran Wang. https://www.eecis.udel.edu/˜xiwang/open-source-projects-data.html, 2015. [Online; accessed 1-April-2016].

[117] Xiaoran Wang and Coskun Bayrak. Injecting a permission-based delegation model to secure web-based workflow systems. In *Proceedings of the International Conference on Intelligence and Security Informatics*, pages 101–106, June 2009.

[118] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 35–44. IEEE, 2011.

[119] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process*, 2013.

[120] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Developing a model of loop actions by mining loop characteristics from a large code corpus. In *Proceedings of 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–44. IEEE, 2015.

[121] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatically generating natural language descriptions for object-related statement sequences. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 205–216, Feb 2017.

[122] Andrzej Wasylkowski, Andreas Zeller, and Christian 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.

[123] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567, Nov 2013.

[124] Wordnet. https://wordnet.princeton.edu/wordnet/, March 2015. [Online; accessed 1-April-2016].

[125] Lei Wu, Linjun Yang, Nenghai Yu, and Xian-Sheng Hua. Learning to tag. In *Proceedings of the 18th international conference on World wide web*, pages 361–370. ACM, 2009.

[126] Jinqiu Yang and Lin Tan. Inferring semantically related words from software context. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 161–170, Piscataway, NJ, USA, 2012. IEEE Press.

[127] Annie T. T. Ying and Martin P. Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 655–658, New York, NY, USA, 2013. ACM.

[128] N. Yoshida, M. Kinoshita, and H. Iida. A cohesion metric approach to dividing source code into functional segments to improve maintainability. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 365–370, 2012.

[129] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 506–511, Nov 2015.

[130] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 625–636, Sept 2016.

[131] Benwen Zhang and James Clause. Lightweight automated detection of unsafe information leakage via exceptions. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 327–338, New York, NY, USA, 2014. ACM.

[132] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.

[133] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM*

International Conference on Automated Software Engineering, ASE '09, pages 307–318, Washington, DC, USA, 2009. IEEE Computer Society.

[134] Bohdan Zograf. Java programming style guidelines, January 2011.

Appendix

ACTION UNIT ANNOTATION INSTRUCTIONS FOR SOFTWARE READER STUDY

In this study, we are trying to develop the notion of an action unit. We have provided an informal definition of what it is. Please use this definition to annotate the action units in methods as instructed below.

We define an action unit as an algorithmic step, typically more than one statement in length, within a method body.

For example, consider the following method. We consider this method to be comprised of three high-level algorithmic steps. The first step is checking if the given bitstream exists in bitstreams, the second step is adding bitstream to bitstreams, and the third step is adding the newly created mapping row to database. Notice that by itself, any subpart of any of these steps does not look like a high-level algorithmic step.

```
public void addBitstream(Bitstream b) throws SQLException, AuthorizeException {
  for (int i = 0; i < bitstreams.size(); i++) {
    Bitstream existing = (Bitstream) bitstreams.get(i);
    if (b.getID() == existing.getID()) {
         return;
    }
  }


  bitstreams.add(b);

  TableRow mappingRow = DatabaseManager.create(ourContext, "bundle2bitstream");
  mappingRow.setColumn("bundle_id", getID());
  mappingRow.setColumn("bitstream_id", b.getID());
  database.add(mappingRow);
}
```

Listing A.1: Action Unit Study Example

You will be given 10 Java methods, each with blank lines removed. Please indicate the action units in each method based on your understanding of the definition of action unit.