Towards Automatically Generating Summary Comments for Java Methods*

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock and K. Vijay-Shanker
Department of Computer and Information Sciences, University of Delaware
Newark, DE 19716 USA
{gsridhar, hill, muppanen, pollock, vijay}@cis.udel.edu

ABSTRACT

Studies have shown that good comments can help programmers quickly understand what a method does, aiding program comprehension and software maintenance. Unfortunately, few software projects adequately comment the code. One way to overcome the lack of human-written summary comments, and guard against obsolete comments, is to automatically generate them. In this paper, we present a novel technique to automatically generate descriptive summary comments for Java methods. Given the signature and body of a method, our automatic comment generator identifies the content for the summary and generates natural language text that summarizes the method's overall actions. According to programmers who judged our generated comments, the summaries are accurate, do not miss important content, and are reasonably concise.

Categories and Subject Descriptors:
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Documentation*

General Terms: Algorithms, Documentation

1. INTRODUCTION

Software maintenance demands as much as 90% of software engineering resources [24], and much of this time is spent understanding the maintenance task and any related software or documentation [25]. In spite of numerous studies demonstrating the utility of comments for understanding software [30–32], few software projects adequately document the code to reduce future maintenance costs [6, 14].

There are a number of ways to overcome insufficient comments. One approach is to obviate comments by using extremely descriptive identifier names [9]. Unfortunately, precise identifiers that accurately describe an entity lead to very long identifier names [2, 16]. Longer names can actually *reduce* code readability, rather than increase it [2, 16].

Another way is to encourage the developer to write comments (1) by automatically prompting the developer to enter them [8,22], or, (2) by using a top-down design paradigm and generating comments directly from the specification [23], or, (3) by using a documentation-first approach to development [15]. Although these solutions can be used to comment newly created systems, they are not suitable for existing legacy systems.

An alternative to developer written comments is to automatically generate comments directly from the source code. In addition to providing comments for existing code, generated comments encourage developers to comment newly written code [19]. In recent work [3], Buse and Weimer automatically generate comments about the exceptions thrown by a given Java method. Specifically, they use symbolic execution to identify under what conditions the exception is thrown, and then use a set of templates to generate the textual comment for these conditions. However, the templates they developed are specific to exception handling, and are inadequate for generating comments in general. We are unaware of any system that is capable of automatically generating comments in general, for arbitrary methods.

This paper describes a novel technique to automatically generate descriptive summary comments for Java methods. The system takes a method signature and body as input and outputs a natural language summary comment for the method. Our key insight into automatic summary comment generation is to model the process after natural language generation, dividing the problem into subproblems of content selection and text generation [21]. In our context, *content selection* involves choosing the important or central code statements to be included in the summary comment. For a selected code statement, *text generation* determines how to express the content in natural language phrases and how to smooth between the phrases to mitigate redundancy.

Comments can be used to communicate a variety of information. In this work, we focus on comments that describe a method's intent, which we call *descriptive comments*. Specifically, we are interested in descriptive comments that summarize the major algorithmic actions of the method. Similar to how an abstract provides a summary for a natural language document [18], a descriptive summary comment for a method can help programmers quickly understand what a method does, aiding program comprehension and software maintenance. We target *leading* summary comments occurring before a method, in contrast to other places comments

*This material is based upon work supported by the National Science Foundation Grant No. CCF-0702401 and CCF-0915803.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

may occur. Leading summary comments are not only useful for a developer while modifying code, but also useful for skimming a set of methods to decide which methods need to be examined in more detail.

Our automatic summary comment generation involves analysis of source code, requiring no execution information, and thus can be applied to incorrect, incomplete, and unexecutable legacy programs. The comment generation process, currently implemented for Java, requires less than 5 minutes for 200 KLOC.

The main contributions of this paper are:

- An algorithm to automatically extract important code statements for a method's summary comment,
- A text generation technique that takes a Java code statement as input and outputs a natural language phrase which represents the code, and
- A human evaluation of the accuracy, content adequacy, and conciseness of the automatically generated leading descriptive summary comments.

Our results show that the generated summary and individual phrases are accurate, do not miss important summary information and are reasonably concise.

2. PROBLEM: GENERATING COMMENTS AUTOMATICALLY

Users prefer documentation that is at an appropriate level of detail, "complete", and "correct" [20]. From this high-level goal, we derive our problem statement:

> Given a method signature and body statements for a method M, generate natural language text that summarizes the overall actions of M accurately, adequately, and concisely.

The most important goal is to generate comments which are accurate and contain adequate content. That is, a summary comment should not miss important information for understanding, should be specific enough to be practically useful to the developer, and not overly general. A concise comment contains little to no redundancy and has no extraneous information, thereby not wasting the developer's valuable reading time.

There are a number of challenges in developing a system to automatically generate comments. First, a system will be hindered if the source code does not include at least some meaningful variable, method, and type names. We believe this requirement is reasonable, given that developers tend to choose long and descriptive names for highly visible program entities such as methods and types [16]. Assuming the developer used meaningful identifier names, the problem of automatically generating descriptive summary comments from the source code has three main challenges:

Method names are inadequate summaries. In prior work [13], Hill, et al. presented a technique capable of generating natural language phrases for arbitrary Java method signatures. One way to generate a comment would be to simply generate a phrase for the method based on its signature. At first glance, this might seem an adequate summary of a method's actions. For the method compareTo(Object object) in the class PlayListFile, the phrase "compare play list file to object" is an accurate summary of the method's actions. Unfortunately, there are many well-named methods for which this summarization technique will fall short.

When using the method signature to summarize a method's actions, we are making the assumption that the method's name accurately captures the method's main action. In compareTo, the method name contains the main action, "compare". However, some methods are not named based on the action being executed, but *under what conditions* the action takes place. For example, consider the methods mousePressed, actionPerformed, and afterSave. We call these methods *reactive*, since their names are based on the events the methods are reacting to, rather than the actions being performed. Reactive naming is very common for methods conforming to an API or overriding existing methods.

Because it is impossible to rely solely on a method's name for an accurate summary comment, a general-purpose comment generation technique must use information from the method's body. That leads us to our next two challenges:

Not all method body statements belong in a summary. Another way to generate a leading descriptive comment would be to simply generate a phrase for every statement in the method. However, such a comment would cease to provide a summary of the method's actions, and instead uselessly repeat the entire method body. For a comment to qualify as a summary, it must contain less information and be faster and easier to read than the method's source code. Although every statement in a method is necessary for proper execution, just a few statements may be important for the summary.

Consider the method processSource in the popular open-source project, Rhino, which contains 35 statements. In addition to the main actions, which "read each input file, compile the file and write the results to an output file," the other statements in the method handle exceptions, resource cleanup and object creation. Although all 35 statements are necessary for execution, just 4 are needed for the summary. Thus, one of the major challenges in summary comment generation is to precisely identify these necessary statements.

Using names in the summary loses contextual information from the source code. Assuming we can automatically select the important statements containing content that should be included in the summary, the next challenge is how to present that content. The simplest approach would be to generate a phrase based only on the statement. For example, consider the statement print(current); from which one could generate the phrase "print current". The problem with this approach is that the name of the variable current alone is insufficient; we are left with no concept of what is being printed. The missing contextual information is current's type, which is Document. However, for any given statement there may be a wealth of contextual information—the types of variables, the return type or declaring classes of methods, or the names and types of the formal and actual parameters—and including all of this contextual information would lead to a verbose or redundant comment. Thus, when generating a summary from a method's body statements, we must carefully select additional contextual information for the summary that the developer would have had when reading the code directly.

3. AUTOMATIC SUMMARY GENERATION

There are three main components to our approach to automatically generating leading summary comments: (1) selecting the content, or *s_units*, to be included in the sum-

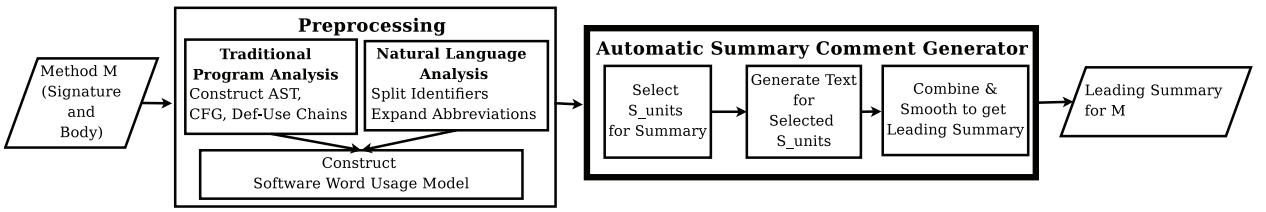

Figure 1: The Summary Comment Generation Process

mary comment, (2) lexicalizing and generating the natural language text to express the content, and, (3) combining and smoothing the generated text. Our design is driven by our goals: (1) to accurately represent the method's main actions, (2) to include precise contextual information needed for understanding, and (3) to be concise and avoid unnecessary words.

Figure 1 illustrates our summary comment generation process. The input to the generator is a set of structural and linguistic program representations for a method. We use information from programming semantics, naming conventions and linguistic knowledge gained from observations of thousands of Java programs. In this section, we describe our approach to: preprocess the source code, convert code statements into a meaningful semantic representation, select content, and generate text for a summary comment.

3.1 Preprocessing

Before any names can be analyzed for text generation, identifiers must be split into component words. We use camel case splitting, which splits words based on capital letters, underscores, and numbers (e.g., XYLine3DRenderer would be split into "xy line 3 d renderer"). We will adopt more advanced splitting techniques after they are refined [7].

Abbreviations in variable and type names can reduce the readability of generated comments and accuracy of our analysis (e.g., Button butSelectAll, MouseEvent evt). We use techniques from prior work [12] to automatically identify and expand abbreviations in code before generating text.

During summary content selection, we use information from the control flow graph, data and control dependences, along with textual clues which we obtain from the Software Word Usage Model (SWUM) of the program.

3.2 Software Word Usage Model (SWUM)

Automatically generating comments requires the ability to identify linguistic elements of an arbitrary method. Specifically, it is critical to identify the *action*, *theme*, and any *secondary arguments* for a given method. Consider the example method signature list.add(Item i), which can be captured by the phrase, "add item to list." In this example, the action is "add", the theme is "item" and the secondary argument is "(to) list". The action, theme, and optional secondary arguments can be used to generate intelligible text for the summary and, in conjunction with program structure, can be used to select what content to summarize.

To capture the action, theme, and secondary arguments for a given method along with its program structure, we use a novel Software Word Usage Model (SWUM). Based on the prior work of Hill, et al. [13], we are developing SWUM to capture the conceptual knowledge of the programmer as expressed in both linguistic information and programming language structure and semantics [11]. Thus, SWUM not only captures the occurrences of words in code, but also their linguistic and structural relationships.

Although a formal definition of SWUM and its construction is outside the scope of this work, we can provide some intuition behind SWUM's extraction rules, which are used to construct the model. Based on common Java naming conventions [16, 28], we assume that most method names start with a verb. SWUM assigns this verb as the action and looks for the theme in the rest of the name, the formal parameters, and then the class. However, some methods do not begin with a verb, such as str.length() and obj.toString(). In these situations, SWUM infers an action for the method, such as "get" or "convert", respectively. In general, the position of a word within a name (e.g., beginning, end) or its structural location (e.g., formal name, formal type, class) provide clues as to the word's semantics. For example, the action in the method void saveImage() is "save", whereas in Image savedImage(), the action is "get", and in void imageSaved() the action is "handling" or "reacting to" the event of the image being saved.

For content selection, we exploit the structural relationships based on the linguistic information of the theme and secondary arguments. For text generation, we use the semantics captured by action-theme relationships, in conjunction with the actual words, to generate intelligible phrases that accurately represent the code.

3.3 S_unit Selection

We focus on the content that would be in the first few sentences of a descriptive summary comment, which provides the reader with the overall gist of a method's actions. Our primary goal in *s_unit selection* is to choose the important or central lines of code to be included in the summary.

Because a Java statement can span multiple lines, we define an *s_unit* to select individual lines. An *s_unit* is a Java *statement*, except when the statement is a control flow statement; then, the s_unit is the *control flow expression* with one of the *if, while, for* or *switch* keywords.

Our heuristics for selecting s_units for a summary comment are based on a wealth of observations. Specifically, we have identified characteristics similar to beacons [5] for summary comments, where a beacon is a surface feature which facilitates comprehension. We have studied comments from popular open source Java programs, investigated the summary needs of methods with a variety of structural characteristics, and surveyed experienced Java programmers as to which statements they personally felt were necessary for summary comments. We analyzed this information and classified statements by their importance to a summary comment. We then looked for patterns of characteristics, such as location within the method body, data and control depen-

dences, relation to other statements in the summary, and role in programming.

3.3.1 Identifying Major S_unit Candidates

Program comprehension studies have shown that each line does not make an equal contribution to comprehension for expert programmers [27]. This section describes a set of characteristics, or clues, that suggest when an s_unit is a good candidate for a method's summary comment.

Ending S_units. An ending s_unit lies at the control exit of a method. To identify the ending s_units of a method *M*, we build a single-entry, single-exit Control Flow Graph (CFG) for *M*, with all *return* nodes leading to the *exit*. An ending s_unit is a predecessor of the CFG's *exit* node.

In the *void* return example below, the s_unit on line 6 is an *ending s_unit*. Note that the *reactive* method name returnPressed does not provide a summary of the method's actions and hence we need to use additional heuristics to select s_units for the summary.

```
1 void returnPressed() {
2    Shell s = getShell();
3    String input = s.getEnteredText();
4    history.addElement(input);
5    String result = evaluate(input);
6    s.append(result);
7 }
```

We observed that methods often perform a set of actions to accomplish a final action, which is the main purpose of the method. This observation across many methods led to the selection of *ending s_units* for the summary.

Void-Return S_units. An s_unit which has a method call that does not return a value or whose return value is not assigned to a variable is a *void-return s_unit*. In returnPressed, line 4 is a *void-return s_unit*. We have observed that method calls that do not return a value often supply useful content for a summary. The intuition is that when a method call does not return a value, it must be invoked purely for side effects. In contrast, method calls returning a value will likely serve as facilitators to a major action by their return value being used to build toward the main action. We use a method's AST to identify void-return s_units.

Same-Action S_units. In a method *M*, if an s_unit has a method call *c* such that *M* and *c* have the same action, then the s_unit is called a *same-action* s_unit. On line 4 below, consider the compileRE method call, which is neither an *ending* s_unit nor a *void-return* s_unit.

```
1 Scriptable compile(Object[] args) {
2  String s =  ScriptRuntime.toString(args[0]);
3  String glob = ScriptRuntime.toString(args[1]);
4  re = (RECompiled)compileRE(s, glob, false);
5  lastIndex = 0;
6  return this;
7 }
```

Intuitively, the call on line 4 is very important towards achieving the intended functionality of compile. Thus, in a summary of a method, *M*, we include the *same-action s_units*. We use SWUM to identify such same-action s_units. In the compile method, the call on line 4, compileRE, has the same action as the method, compile.

Data-Facilitating S_units. Data-facilitating s_units assign data to variables used in s_units identified by the previous three heuristics. For example, consider the s_unit s.append(result) on line 6 in returnPressed. Although this s_unit is selected by the ending s_unit heuristic, little information is known about the theme variable, result. Thus, we select its *data facilitator* on line 5, String result = evaluate(input). Although an s_unit may contain many variables, because our text generation technique only uses variables that represent the theme or secondary arguments in a method call, we only identify data-facilitating s_units for these variables. For s_units without a method call, we currently find data facilitators for all variables used. We use SWUM to identify the variables corresponding to the theme and secondary argument, and then we find data facilitating s_units for these variables using the *def-use* chains for the method. We go back one level from the *use* along the *def-use* chain.

Controlling S_units. A controlling s_unit (i.e., an s_unit with one of the *if, while, for* or *switch* keywords) controls the execution of an s_unit previously selected by one of the earlier heuristics. For example, consider the method actionPerformed below. Line 5 contains an *ending* s_unit that is control dependent on line 4, making line 4 a *controlling* s_unit.

```
1 void actionPerformed(ActionEvent e) {
2   String cmd = e.getActionCommand();
3   if (cmd != null) {
4      if (cmd.equals("interrupt"))
5         exportFile.delete();
6   }
7 }
```

The summary content would be inadequate without the controlling s_unit. Contrast the following two plausible summaries for this method: /*delete export file*/ and /*if command equals interrupt, delete export file.*/ The latter summary conveys more information, specifically about when the major action occurs. Once a controlling s_unit is included in a summary, we recursively include its controlling s_unit, if any, so the reader is given the conditions under which a major action is performed.

3.3.2 Filtering out Ubiquitous Operations

Some operations are less specific to a method's computational intent than to the overall program behavior. Such s_units would add unnecessary information to a summary comment. For instance, exception handling statements are typically not needed for a summary. Similarly, programmers would not expect information describing a method's resource cleanup operations, logging or other diagnostic operations in a method's summary comment. The exception to this rule is when a method's sole intent is exception handling, resource cleanup, or logging. We identify such s_units by using the AST and checking if an s_unit is within the *catch or finally* block of a method. We also check the action and theme identified by SWUM for the words 'log', 'error', 'debug', 'trace', 'exception', or 'close'. Similarly, a code pattern of the form if (X == null), with only a return null statement in its body, represents an exception handling block that is typically used to check for null input parameters.

Some *ending* s_units can also be omitted from a summary. Many methods return a boolean value to indicate success or failure. For example, a method named saveData() might return false if it fails to save the data. We believe these failure paths to be unnecessary and thus do not add return false s_units in such methods to the summary.

In addition, certain *void-return* and *data-facilitating* s_units can be elided from a summary. Specifically, *get, set,* and *object creation* operations are ubiquitous in an object-oriented system and do not provide important distinguishing information for a method summary's actions. Hence, we exclude s_units in which a *get/set/object creation* method is the outermost call. The exception is when the computational intent of a method is to get/set a value or create an object. Similarly, a statement that initializes a variable to a value *0* or *null* is too trivial to include in a summary comment.

Also, not all controlling s_units are needed for a summary. An *if* expression of the form if (X != null) without an accompanying else is often used to guard against null pointer exceptions. These sanity checks are not important enough for a concise summary comment.

3.3.3 S_unit Selection Process

We have a three phase process to select the summary s_units for a method *M*. In the first phase, we identify the *same-action*, *ending*, and *void-return* s_units, and add them to the *summary set*. For each s_unit in the summary set, we add its *data-facilitating* s_units to the summary set. We then augment the summary set with any controlling s_units. In each phase, we apply the filter to exclude ubiquitous operations along with s_unit identification. Finally, the *summary set* is sorted in ascending order based on the line number within the method. In the previously shown returnPressed example, the s_units on lines 4 and 6 are selected in the first phase. The s_unit on line 5 is added during the second phase. No s_units are added during the third phase.

3.4 Text Generation

After identifying the set of s_units for the content of the summary comment, the next challenge is to convert each s_unit into a natural language phrase that can be understood independent of the s_unit's context within the method body.

Given: f.getContentPane().add(view.getComponent(), CENTER)
We generate:
/* Add component of drawing view to content pane of frame*/

The text generator first constructs subphrases for the arguments in an s_unit and then concatenates subphrases for the entire s_unit. In generating phrases for the arguments (e.g., component, of view, to pane, of frame), our system strives to produce more descriptive phrases than contained within individual s_units. For instance, in the above example, we generate "drawing view" rather than "view". This addition of the modifier "drawing" to produce the more specific "drawing view" is accomplished via *variable lexicalization*.

Note that we did not include the parameter CENTER in the generated text as it does not correspond to the *theme* or *secondary argument* of add. We do not claim that CENTER is an unimportant parameter of add, but it is not necessary to include in a summary.

Although the example shown has 3 method calls with associated verbs, the generated text includes only one verb. The text generator combines phrases when possible and removes words that are not needed for the summary. For example, we dropped the 'get' in the generated text for getContentPane() and getComponent(). Although not shown in our example, s_units that are conditional or loop expressions also have templates for generation.

The basic s_unit is one with a single method call. The text generation strategy for a single method call is then used in *nested method calls, composed method calls, assignments, returns, conditional* and *loop expressions*. Before describing our approach to generate text for these s_units, we first describe how we lexicalize the variables used in those s_units.

Lexicalization of Variables. One challenge in text generation is to construct noun phrases that represent variables. In English noun phrases, more specific noun modifiers appear on the left. Since a variable's type name typically provides general information about the variable while the name provides specific information, when converting a variable into an English noun phrase we generally place the variable name to the left of the type name. For example, consider the variables Document current and CallFrame parentFrame, which would be lexicalized as "current document" and "parent call frame", respectively. Notice that the phrase for the latter example does not repeat the word "frame". However, when the type name is an adjective (e.g., Selectable item), we place the variable name to the right ("selectable item").

Single Method Call. Consider a method call M(...). In Java, a method implements an operation and typically begins with a verb phrase [29]. Thus, we generate a verb phrase for M. The template for the verb phrase is:

action theme secondary-args
and get return-type [if M returns a value]

where *action, theme* and *secondary arguments* of M are identified by SWUM and correspond to the verb, noun phrase and prepositional phrases of the verb phrase.

Example s_unit: os.print(msg)
We generate: /* Print message to output stream */

Text generation can be improved by identifying *theme equivalences*, which occur when the theme in a method name overlaps some of its parameters. Consider the call removeWall(oldWall), for which we can generate the phrases: "remove wall", "remove wall, given old wall", "remove old wall". The last phrase is the most concise and descriptive. We generate this phrase by determining theme equivalence between the theme 'Wall' in the method name and parameter name. However, not all overlap leads to theme equivalence. Consider the call addItem (itemURL), which adds an item after retrieving the item from its URL. Although the theme, "item", overlaps the parameter, "item url", these phrases do not have equivalent meanings. Thus, when the overlap only *begins* the lexicalized parameter name, and does not occur at the end, we distinctly incorporate both the theme from the method name as well as the lexicalized parameter name. Table 1 shows some examples of theme equivalence.

Method Signature and Call		Generated Text
sig:	removeWall (Wall w)	"Remove old wall"
call:	removeWall(oldWall)	
sig:	addImage (ThumbImage ti)	"Add new thumb image"
call:	addImage(newThumbImg)	
sig:	addItem (Item itemURL)	"Add item, given item URL"
call:	addItem (itemURL)	

Table 1: Theme equivalence for text generation

Return. A *return* s_unit is a pseudo-method call where the action is 'return' and the theme is the method's *return type*.

Nested and Composed Method Calls. An s_unit may contain nested calls like $M_2(M_1(...),...)$ or composed calls such as $M_1.M_2(\dots)$. When SWUM detects that $theme_2$ = return-$type_1$, the template for text generation in general is:

> *action*$_1$ *theme*$_1$ *secondary-args*$_1$
> *and get return-type*$_1$, [if M_1 returns a value]
> *action*$_2$ *theme*$_2$ *secondary-args*$_2$
> *and get return-type*$_2$ [if M_2 returns a value]

Example s_unit: print(sendRequest())
We generate: /*Send request and get response, print response*/

When *theme*$_1$ = *return-type*$_1$, we can drop the second phrase from our general template, "*and get return-type*$_1$", without losing information.

Example s_unit: menu.add(makeMenuItem());,
We generate: /* Make menu item, add menu item to menu*/

When *theme*$_1$ = *return-type*$_1$ *AND action*$_1$ = *'get'*, we drop the first two phrases from our general template without loss of information. We use the theme and secondary arguments of M_1 as the theme of M_2. The "Add component of drawing view ..." example at the beginning of this section illustrates this case. The same example also illustrates composition along with combination of nesting and composition.

Assignment. Consider the general form of an assignment s_unit, lhs = M(...). The general template is:

> *action theme secondary-args and get return-type*
> *assign to lhs*

Since assignment involves "assigning" the *return-type* to the *lhs*, we can avoid redundancy by combining the second and third phrases in our template as *"get lhs"*.

Example s_unit: fileName = showFileDialog(...);
We generate: /* Show file dialog and get file name */

When *lhs overlaps the theme*, we generate a more concise phrase by dropping the redundant *"get lhs"*, and use either the *lhs* or *theme* in the text, whichever is more specific.

Example s_unit: boldFont = deriveFont();
We generate: /*Derive bold font*/

We are still developing templates for assignments where the right hand side is not a method call or variable but an infix expression (e.g., value = y+x*width).

Conditional Expressions. For an *if* s_unit, we generate text for the boolean expression of the *if* which controls execution of another identified s_unit. When an *if* expression is composed of multiple boolean expressions combined via operators, we generate text for the individual expressions and combine them.

In contrast to method calls, which are typically realized as verb phrases, boolean expressions are realized as sentences, since sentences can convey a true or false value. We developed SWUM rules for methods returning a boolean value and boolean fields to obtain the *subject* of a sentence.

When the expression is a boolean variable, we simply use the variable name in the generated text. In the case when the variable is a field and an adjective, SWUM provides the class name as the noun:

Example s_unit: if (visible)
We generate: /* If window is visible */

When the boolean expression is a method call, we identify the subject and place the verb after the subject as in an English sentence. When the method name begins with a third person singular verb (e.g., equals), we identify the subject as the receiving object of the method call (first parameter in case of static methods).

Example s_unit: if (amb.getStyles().contains(t.getStyle()))
We generate: /* If styles of ambience contains style of track*/

When the method call begins with an *auxiliary verb* (e.g., is), the subject is in the method name or is a receiver object or parameter of the method call.

Example s_unit: if (AbstractDoc.hasOnlySpaces(prefix))
We generate: /* If prefix has only spaces */

When the call begins with a base verb, the return value indicates success or failure of the operation. Thus, we transform the verb phrase into a propositional sentence by appending "succeeds" to the end of the phrase.

Example s_unit: if(saveAuctions())
We generate: /* If save auctions succeeds */

When the expression involves a comparison of two operands using an equality or a relational operator, the operator provides the verb, while the arguments of the verb are given by the phrases for the operands.

Loop Expressions. For *while* s_units, in general, text for the boolean loop expression is generated like an *if* s_unit. However, *while* loops frequently iterate over a collection using the well-known *iterator* pattern:

```
1 Iterator it = homeFolder.iterator();
2 while (it.hasNext()) {
3  File f = (File) it.next();
4  ...
5 }
```

Although we can generate, /* While iterator has next */ for the selected s_unit on line 2, we recognize the *iterator* pattern and generate the phrase /* For each file in home folder */ by using information from lines 1, 2 and 3.

We identify iterators by checking if the method call in the loop condition expression has a receiver object with a type name that contains either "Iterator" or "Enumeration". We use the template "For each *item* in *collection*". To get *item*, we use *def-use* chains and look for a use of the iterator as the receiver object of a call that returns a value within the body of the loop. This return value is used as the *item*. To get *collection*, we again use *def-use* chains to look for the

receiver object of a method call that provides a definition for the iterator.

Another frequently occurring loop expression iterates with an index value to access all elements of a collection:

```
1 for (int i=0;i<tree.getFunctionCount();++i){
2  FunctionNode fn=tree.getFunctionNode(i);
3  ...
4 }
```

Rather than generating /* For i < function count of tree */ for the s_unit on line 1, we produce /* For each function node in tree */ by using information from lines 1 and 2. As with iterators, we use *def-use* chains and linguistic clues (theme contains "count", "length" or "size" in the call on line 1) to identify the *item* and *collection* for the generation template.

Prototype Implementation. Our prototype, *genSumm*, is implemented as an Eclipse Plug-in. It implements the s_unit selection process and the text generation templates. *genSumm* utilizes SWUM, also an Eclipse plug-in.

4. EVALUATION

Our primary goal for this first automatic summary comment generator is to generate comments that are highly accurate, tolerating some unnecessary information in the summary, and allowing some content to be missed as long as the missing information does not hinder understanding the method. To examine how well we attain these goals, we designed a study to examine the following research questions:

Accuracy: How accurately does our generated text represent the code's actions?

Content Adequacy and *Conciseness*: How effectively can we automatically identify the s_units for the summary?

With no existing automatic comment generators for comparison, we asked humans to judge the generated comments, targeted at evaluating accuracy, content-adequacy, and conciseness. The study included 13 graduate students and postdocs in computer science, with advanced-to-intermediate Java programming experience. Six have experience in the software development industry.

Table 2 presents the multiple-choice questions that we asked the human evaluators and the possible answers. To account for variation in human opinion, we obtained and analyzed three separate judgements for each generated text.

Although the measures of *conciseness* and *content adequacy* appear similar to *precision* and *recall*, respectively, they are difficult to measure for this study. Measuring precision and recall requires mapping the evaluators' summaries to exact s_units, and the summaries may seamlessly merge information across different s_units.

We selected the methods in our study from four open-source programs that span different domains: **Megamek**, an unofficial, online version of the Classic BattleTech board game with 9300 methods and 200 KLOC; **SweetHome3D**, an interior design application with 4000 methods and 73 KLOC; **JHotDraw** a Java GUI framework for technical and structured graphics with 4300 methods and 63 KLOC; and **Jajuk**, an application that organizes and plays music with 2100 methods and 44 KLOC.

Since a summary comment consists of text generated for selected individual s_units, we first evaluate text generated for individual s_units and then for whole summaries. We did not evaluate s_unit selection in isolation from comment generation because we have found it very difficult for humans to judge s_unit selection independent of the generated text and to map their summaries back to specific s_units.

4.1 Evaluating S_unit Text Generation

Procedure. Our goal was to gain feedback on text generation templates for as many of the different types of s_units as possible without demanding a lot of time from our human subjects. Thus, we randomly selected s_units such that each newly selected method covered a new s_unit type, until we selected a total of 48 s_units. Each of 12 human evaluators examined a total of 12 s_units, 6 from two of the four programs, leaving us with 144 judgements of s_unit text.

We provided each evaluator with the body of the method in which each s_unit occurred and the entire program source. We instructed the subjects to read and understand what the method does and then write their own summarizing text for the s_unit. Since a summarizing text for an s_unit is very subjective, we did not want to bias the evaluators and *deliberately* did not provide exact instructions on what text to write. Once the subjects wrote text for an s_unit, they examined the text automatically generated by our system and answered the three questions similar to those in Table 2.

Results and Discussion. Figure 2 reports the majority opinions (where 2/3 evaluators agreed) and the number of individual responses in each category of responses for *Accuracy*, *Content Adequacy* and *Conciseness* of s_unit phrases. The overall results are quite positive and suggest that in general, we have met our stated goals of generating accurate text, including all the information relevant to understanding the method and avoiding unnecessary text.

Accuracy. Of the 144 accuracy judgements over all 48 s_units, 127 rated the generated comment *accurate* with only 1 judgement giving a rating of *very inaccurate*. Furthermore, no generated comments were ranked *very inaccurate* by majority opinion, and in fact, when majority opinion is taken into account, only 4 out of 48 generated comments were ranked as even *slightly inaccurate*. We analyzed the few comments judged as slightly to very inaccurate based on the text written by the evaluator. Two cases require SWUM to more precisely identify the action and theme, one requires domain knowledge, and in the final case, the generated text was so specific that it created the illusion of inaccuracy. For the s_unit parent.addChild (createPolyline( )), we generated /*Create polyline and get xml element. Add child, given child xml element to parent xml element*/. Evaluators who did not consider the return type of createPolyline said the generated text was slightly inaccurate.

Accuracy			Content Adequacy			Conciseness		
Response Category	**S**	**R**	**Response Category**	**S**	**R**	**Response Category**	**S**	**R**
Accurate	44	127	Adequate	41	110	Concise	46	128
Slightly Inaccurate	4	16	Misses Some	7	28	Slightly Verbose	2	15
Very Inaccurate	0	1	Misses Important	0	6	Very Verbose	0	1

Figure 2: Human judgements of individual phrases. S = Majority opinion (# s_units). R = # Responses.

Criteria	Question	Answer Choices
Accuracy	Independent of *Content Adequacy* and *Conciseness*, do you think that the comment is	• accurate • a little inaccurate • very inaccurate
Content Adequacy	Looking at only the content of the generated comment and not the actual text or the way the text is presented, do you think that the comment:	• is missing some very important information that can hinder the understanding of the method • is missing some information but the missing information is not necessary to understand the method • is not missing any information
Conciseness	Looking at only the content of the generated comment and not the actual text or the way the text is presented, do you think that the summary:	• has a lot of unnecessary information • has some unnecessary information • has no unnecessary information

Table 2: Questions and answer choices designed to elicit human judgements.

Content Adequacy. For 138 of 144 judgements the comments were not missing any important information for understanding, with 110 of those indicating no missing information. More significantly, there were no comments for which there was a majority opinion that important information was missing for understanding. Further, 41 of the 48 comments had a majority opinion of no missing information. In the 7 comments where a majority said there was some information missing, but not needed for understanding, the missing information was due primarily to how we generated text for nested method calls and loop conditions. Evaluators wanted more information about parameters beyond the theme and secondary arguments, particularly literals and constants, in the nested method calls. For the loops, the evaluators wanted more information from the loop body and the collection on which the loop iterates.

Conciseness. There was no s_unit where a majority of evaluators said the comment had too much unnecessary information. Instead, 46 of the comments were ranked by majority opinion as having no unnecessary information with only 2 of the 48 comments with the majority of evaluators saying it had some unnecessary information. When we examined the comments where there were 15 individual judgements of slightly verbose, we found that we could further avoid redundant information in assignments and nested calls by recognizing additional opportunities for simplification.

4.2 Evaluating Whole Summary Comments

Procedure. Judging a summary comment adequately to carefully answer our questions takes considerable time because the humans need to read and understand each method's body. Thus, we provided six of our human evaluators with four methods each, one from each of the four programs. A total of eight summary comments, two per program, were judged independently by 3 human evaluators. To control for learning effects, the humans did not see the methods in the same order.

To select methods for evaluation, we first eliminated methods with characteristics that make them quickly understandable without a summary comment or contain features that we do not yet attempt to address, such as methods with a `switch` statement where each case has a similar action. Thus, for this study, we chose methods that have between 10 to 40 s_units (about 95-99% of the methods in our subject programs had s_units ≤ 40). We did not select constructors or methods beginning with `get` or `set` since the method name itself can serve as a summary comment. From the remaining methods, we randomly selected two methods from each of the four programs.

Accuracy			Content Adequacy			Conciseness		
Response Category	M	R	Response Category	M	R	Response Category	M	R
Accurate	7	18	Adequate	4	11	Concise	3	13
Slightly Inaccurate	1	6	Misses Some	2	6	Slightly Verbose	4	6
Very Inaccurate	0	0	Misses Important	2	7	Very Verbose	1	5

Figure 3: Human judgements of method summaries. M = Majority opinion (# methods). R = # Responses.

As with the previous evaluation, we asked the evaluators to read, understand and write their own summary of each of the four methods assigned to them. To avoid biasing the evaluators, we did not give an explicit definition of a summary but only mentioned that the goal of a summary is to provide a gist of what the method does. Once the evaluators wrote a summary, they examined the summary generated by our system and answered the questions in Table 2.

Results and Discussion. Figure 3 reports the majority opinions and the number of individual responses in each category for *Accuracy*, *Content Adequacy* and *Conciseness*. In general, we believe the results are positive, especially when considering that our stated goal is to generate accurate summary text while tolerating slight verbosity and missing some information as long as it does not hinder understanding of the method.

Figure 4 shows an example generated summary, evaluators' summaries and the existing summary written by the developer for the method `exportToSVG`. Observe that the method name does not suffice as a summary. The majority opinion about the generated summary was: 'accurate', 'adequate' and 'concise'. We believe that the *similarity* of the generated summary with three of the four human written summaries attests to the accuracy of our text generation; it also testifies to the success of our s_unit selection strategy in selecting only the one relevant s_unit among 18 s_units.

Accuracy. Of the 8 methods, 7 had accurate summaries according to the majority of evaluators per method. Not one among the 24 individual responses suggested that we generated "very inaccurate" text. Only in 1 of the 8 methods did a majority believe the summary text was a "little inaccurate." Further analysis of the evaluators' summaries

Writer	Summary Comments
Developer	Exports the plan objects to a given SVG file
Our System	Export plan component to svg
Evaluator 1	export plan component to svg file
Evaluator 2	Sets up plan component and calls exportToSVG() on it
Evaluator 3	Get the actual planview. If planview is an instance of Plan Component, planview is copied to plan Component. else planComponent is a new instance of Plan Component. Try to get a new instance of BufferedOutputStream associated to the svgFile, and attempt to export it to SVG using the planComponent. If not possible address the potential cases for exceptions

```
1 public void exportToSVG(String svgFile) throws ...
2   View planView = controller.getPlanController()...
3   PlanComponent planComponent;
4   if (planView instanceof PlanComponent) {
5     planComponent = (PlanComponent)planView;
6   } else {
7     planComponent = new PlanComponent(this.home,...
8   }
9
10  OutputStream outputStream = null;
11  boolean exportInterrupted = false;
12  try {
13    outputStream = new BufferedOutputStream(new...
14    planComponent.exportToSVG(outputStream);
15  } catch (InterruptedIOException ex) {
16    exportInterrupted = true;
17    throw new InterruptedRecorderException("Expor...
18  } catch (IOException ex) {
19    throw new RecorderException("Couldn't export ...
20  } finally {
21    if (outputStream != null) {
22      try {
23            outputStream.close();
24            // Delete the file if exporting is...
25            if (exportInterrupted) {
26                  new File(svgFile).delete();
27            }
28      } catch (IOException ex) {
29            throw new RecorderException("Couldn't...
...
33 }
```

Figure 4: Contrasting generated summary with the developer's and evaluators' summaries.

suggests that for the s_unit beginEdit(textHolder), we generated "begin edit," whereas the evaluators wrote "edit text holder." More precise theme and action identification would address this problem.

Content Adequacy. 6 of the 8 summaries were judged to not have missed important information by the majority. In fact, 17 of the 24 individual responses suggest that the generated summaries did not miss important information. In the two methods where a majority felt that some important information was missing, we failed to identify important s_units for three reasons. We missed an s_unit because we do not use semantic similarity word relations between a method's action and a callee's action. Automatically detecting such semantic similarity in programs is not a trivial task [26]. We missed an s_unit because we do not include the data facilitating s_units for method parameters that are not a theme or secondary argument. Another s_unit was missed because the negation operator (!) was used to implement the concept of "flipping".

Conciseness. There was only 1/8 methods for which a majority of the evaluators felt that the generated summary was very verbose. 13 of the 24 individual responses suggest that the generated summary was concise. In the only method where a majority said that the generated summary was very verbose, the *void-return* and *same-action* s_unit selection rules were applied once too often. In the 4 methods where the summaries were slightly verbose, we included too many *if* blocks for special cases and error handling return paths.

4.3 Threats to Validity

Considering our stated problem is the lack of developer written comments, we unsurprisingly found few comments in application code with which to compare our automatically generated comments. Our results may not generalize to other Java programs or other languages. To mitigate this threat, we chose four large open source programs across different domains representative of typical Java programs.

4.4 Summary of Results

The results of our evaluation suggest that our summary comment generation technique has met its stated goals of accurate text, while tolerating some extra and missing relatively unimportant information. The summary generation would benefit from improved action and theme identification along with usage of other parameter information.

Based on evaluator feedback (as seen in the evaluators' summaries in Figure 4), summary generation should be tailored to the developers' experience (novice versus expert) and their personal preferences. Even among experts, the granularity of required detail varies. For example, 2/6 evaluators included exception handling logic in their summary. Thus, we plan to allow the developer to control the amount of detail in a summary.

5. RELATED WORK

There has been some limited work on comment generation. Semi-automated approaches either automatically determine uncommented code segments and prompt developers to enter comments [8,22], or automatically generate comments from high level abstractions which the programmer provides during development [23]. Although useful for newly created systems, none of the semi-automatic techniques apply to existing legacy systems. In contrast, we are aware of some techniques that could be used towards generating comments for legacy code [3,17]. However, these approaches are limited to inferring documentation for exceptions [3] and generating API function cross-references [17], and are not intended for generating descriptive summary comments.

In [10], Harman et al. defined *Key Statements* as the core of a program's computation, statements through which the largest part of the program's dependence appears to flow. From our observation, key statements do not necessarily provide the content for a summary, which is composed of major algorithmic actions which need not correspond to a computation.

Another way of measuring the importance of a statement is through frequency of execution. While useful in targeting optimization [4], these statements do not represent content for a summary.

6. CONCLUSIONS AND FUTURE WORK

To our knowledge, we presented the first technique to automatically generate leading comments that summarize the main actions of an arbitrary Java method by exploiting both structural and linguistic clues in the method. According to programmers who judged our generated comments, the automatically generated summaries are accurate, do not miss important content, and are reasonably concise.

In the future, we will improve content selection by excluding *if* blocks handling special cases and error return paths, by possibly leveraging branch execution frequency estimation [1]. For improved text generation, we are developing more precise action and theme identification. We will endeavor to make the summaries look more like human-written comments by applying smoothing techniques [21]. We will investigate the feasibility of utilizing the generated leading comments at call sites. Lastly, we will extend the evaluation to longer methods, a larger number of methods and include novices in the evaluation.

7. REFERENCES

[1] T. Ball and J. R. Larus. Branch Prediction for Free. *Conf on Programming Language Design and Implementation (PLDI)*, 1993.

[2] D. Binkley, D. Lawrie, S. Maex, and C. Morrell. Impact of Limited Memory Resources. *Intl Conf on Program Comprehension (ICPC), 2008.*

[3] R. P. Buse and W. R. Weimer. Automatic Documentation Inference for Exceptions. *Intl Symp on Software Testing and Analysis (ISSTA), 2008.*

[4] P. P. Chang, S. A. Mahlke, and W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Softw. Practice and Experience*, 21(12):1301–1321, 1991.

[5] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. *Workshop of the Psychology of Programming Interest Group (PPIG)*, 2002.

[6] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. *Intl Conf on Design of Communication (SIGDOC), 2005.*

[7] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. *Intl Working Conf on Mining Software Repositories (MSR), 2009.*

[8] T. E. Erickson. An Automated FORTRAN Documenter. *Intl Conf on Systems Documentation (SIGDOC), 1982.*

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[10] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code Extraction Algorithms which Unify Slicing and Concept Assignment. *Working Conf on Reverse Engineering (WCRE)*, 2002.

[11] E. Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration.* PhD thesis, University of Delaware, 2010.

[12] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. *Intl Working Conf on Mining Software Repositories (MSR), 2008.*

[13] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. *Intl Conf on Software Engineering (ICSE), 2009.*

[14] M. Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Softw. Eng.*, 10(1):31–55, 2005.

[15] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2), 1984.

[16] B. Liblit, A. Begel, and E. Sweetser. Cognitive Perspectives on the Role of Naming in Computer Programs. *Psychology of Programming Workshop (PPIG)*, 2006.

[17] F. Long, X. Wang, and Y. Cai. API Hyperlinking via Structural Overlap. *Foundations of Software Engineering (FSE), 2009.*

[18] I. Mani. *Automatic Summarization.* John Benjamins, 2001.

[19] D. P. Marin. What Motivates Programmers to Comment? Master's thesis, University of California, Berkeley, 2005.

[20] D. G. Novick and K. Ward. What Users Say They Want in Documentation. *Intl Conf on Design of Communication (SIGDOC), 2006.*

[21] E. Reiter and R. Dale. *Building Natural Language Generation Systems.* Cambridge Univ. Press, 2000.

[22] D. Roach, H. Berghel, and J. R. Talburt. An Interactive Source Commenter for Prolog Programs. *SIGDOC Asterisk J. Comput. Doc.*, 14(4), 1990.

[23] P. N. Robillard. Schematic Pseudocode for Program Constructs and its Computer Automation by SCHEMACODE. *Commun. ACM*, 29(11), 1986.

[24] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] M. J. Sousa. A Survey on the Software Maintenance Process. *International Conference on Software Maintenance (ICSM), 1998.*

[26] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. *Intl Conf on Program Comprehension (ICPC), 2008.*

[27] M.-A. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Intl Workshop on Program Comprehension (IWPC), 2005.*

[28] SUN. Code Conventions for the Java Programming Language - Naming Conventions. online. `http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html`.

[29] SUN. How to Write Doc Comments for the Javadoc Tool. online. `http://java.sun.com/j2se/javadoc/writingdoccomments/`.

[30] A. A. Takang, P. A. Grubb, and R. D. Macredie. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. *J. Prog. Lang.*, 4(3), 1996.

[31] T. Tenny. Program Readability: Procedures Versus Comments. *Trans. Softw. Eng.*, 14(9), 1988.

[32] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The Effect of Modularization and Comments on Program Comprehension. *Intl Conf on Software Engineering (ICSE), 1981.*