# Generating Parameter Comments and Integrating with Method Summaries

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

***Abstract*— An important part of the leading comments for a method are the comments for the formal parameters of the method. According to the Java documentation writing guidelines, developers should write a summary of the method's actions followed by comments for each parameter. In this paper, we describe a novel technique to automatically generate descriptive comments for parameters of Java methods. Such generated comments can help alleviate the lack of developer-written parameter comments. In addition, they can help a programmer in ensuring that a parameter comment is current with the code. We present heuristics to generate comments that provide a high-level overview of the role of a parameter in a method. We ensure that sufficient context is provided such that a developer can understand the role of the parameter in achieving the computational intent of the method. In the opinion of nine experienced developers, the automatically generated parameter comments for methods are accurate and provide a quick synopsis of the role of the parameter in achieving the desired functionality of the method.**

***Keywords*-documentation, software maintenance**

## I. Introduction

Often developers would like a high-level overview of the computational intent of a particular method without having to read the entire method body. For instance, in response to a user query, a concern location tool [1] might return just the signature of each relevant method. When the signature does not provide enough details about a method to further distinguish relevance to the maintenance task, a high-level overview of the method's functionality can be obtained from the leading summary comments, either manually written by the developer or automatically generated from the method source [2].

For the method in Figure 1, the developer-written summary is *"create and start meta server"*. While this summary furnishes a succinct abstraction of the method's computational intent, it does not describe the role played by the parameter args. Typically, developers document the role of a parameter by comments (marking such comments in Java by the @param tag). For Figure 1, the developer wrote the parameter comment *''@param args: the command line"*. Unfortunately, this comment does not state how the parameter is used. Note that the parameter is used only in Line 4. Even if a comment phrase is generated based on this line, such as, *"port is obtained using the given args"*, the comment still does not link the usage of the parameter args to the abstraction of the method's computational intent, (captured by the developer's comment *"create and start meta server"*, extracted from Lines 14 and 19).

We believe that it would be more desirable for the parameter comment to suggest how it is used in the context of the computational intent of the method. In this example, the *linkage* to the computational intent can be captured by noting that port is used to create metaServer (Line 14).

```
1 public static void main(String[] args) {
2  int port = -1;
3  try {
4     port = Integer.parseInt(args[0]);
5   } catch (ArrayIndexOutOfBoundsException e) {
6     println("Usage: java MetaServer PORT_NUMBER");
7     System.exit(-1);
8   } catch (NumberFormatException e) {
9     println("Usage: java MetaServer PORT_NUMBER");
10     System.exit(-1);
11   }
12   MetaServer metaServer = null;
13   try {
14      metaServer = new MetaServer(port);
15   } catch (IOException e) {
16      logger.warning("Could not create MetaServer!");
17      System.exit(-1);
18   }
19   metaServer.start();
20 }
```

Figure 1. Running example Java Method : We use lines 4 and 14 to generate parameter comments and link to the method's intent.

In this paper, we present what we believe is the first known technique to automatically generate comments for parameters of a method. For the parameter in Figure 1, our system can automatically generate the comment:

> *@param args: create meta server using the port obtained from the parameter*

Our system is customizable and is also capable of producing a more concise comment:

> *@param args: create meta server using the param*

or a more detailed comment:

> *@param args: parse integer and get port using the parameter. create meta server using the port*

We believe that the automatically generated comments provide a reasonably accurate high-level overview of the role of the parameter args, linking it to the computational intent of the method. Furthermore, a summary with added

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

information about the parameter role provides more insight into the method's functionality than the original developer's comment in this example.

For many formal parameters, there exist no developer-written comments. For instance, in 18 open-source projects including Azureus (Vuze) and JHotDraw, the percent of methods that have one or more parameters but no @param comments ranges from 31% to 97%. Cumulatively, across all 18 projects, only 19% of the 99,316 formal parameters have an @param comment. This suggests that a tool that can automatically generate parameter comments based on the current method code could significantly enhance the documentation of such legacy systems while alleviating the software maintainer from the tedious task. Beyond dealing with a lack of parameter comments, the automatic parameter comment generator could help a developer in keeping existing documentation current with the source code.

Our algorithm takes a Java method as input and analyzes the usage of each formal parameter to synthesize their main role in the method's functionality and their relevance to the method's intent. The output can be tailored to generate individual @param comments and/or augmented method summaries with varying levels of detail involving the parameter usage within the context of the summary. The next section describes the unique challenges addressed, focusing on the significant challenges new to this problem, beyond previous work on summary comment generation [2]. We leverage both programming language syntax and semantic information as well as linguistic clues embedded in the developers' naming of entities. To synthesize succinct natural language descriptions to express the parameter role, we utilize advanced text generation techniques. Our automatic system involves analysis of source code only, requiring no execution information, and thus can be applied to incorrect, incomplete, and unexecutable legacy systems. Since the analysis is local to a method, this capability can be integrated easily into an IDE to provide current descriptions of parameters as a software developer completes editing a given method.

This paper presents the following main contributions beyond the state of the art:

- Heuristics to *automatically identify the main role of a formal parameter* within a method, and *to connect the formal parameter with the method's computational intent* such that a reader can gain a high-level perspective of a method's functionality and the purpose of the parameter in fulfilling that functionality.
- Heuristics to *automatically generate* succinct, accurate phrases for a formal parameter. The generated comments can be in the form of stand-alone parameter comments or can be integrated with the method summary.
- An evaluation by experienced developers of the accuracy, utility and necessity of the generated parameter comments and augmented method summaries.

In the opinion of nine experienced developers, the generated parameter comments and augmented summaries for methods are accurate and are useful in providing a quick overview of the role played by the parameter in achieving the desired functionality of the method.

## II. Challenges

Overall, documentation needs to be accurate with respect to the source code, and contain information that is useful for the intended purpose of the particular documentation. Users also prefer documentation that is complete and at an appropriate level of detail [3]. For parameter comments, our goal was to generate comments that provide an accurate and useful synopsis of the main role of the parameter within the method. We also had the goal of ensuring that the generated parameter comments were connected to the computational intent of the method.

Given a method $M$ and a representation of its computational intent, these goals pose two major challenges:

- Determining the main role of each parameter in the overall method's intent and which statements implement that role
- Identifying the *linking context* between a parameter used in its main role and the method's intent

**Identifying Parameter Role and Corresponding Uses.** A method typically uses each parameter in more than one statement. However, the @*param* comments written by developers do not necessarily describe every statement in which a parameter is used, but rather emphasize the usage within one or more statement(s) that represent the important role of the parameter. Thus, when a formal parameter is used in more than one statement, an important challenge is to automatically determine which among these statements represent the primary usage of the parameter.

On the surface, this challenge appears similar to selecting the key lines of code to be included in an automatically generated method summary comment from among all the method statements [2]. However, in summary generation, the selection is guided by patterns of characteristics such as location within the method body, data and control dependences, relation to other statements in the summary and role in programming, with the goal of finding characteristics similar to beacons [4] for summary comments. In contrast, to identify the major role of a given parameter, the challenge is to estimate the "closeness" of each of the uses of the parameter with the computational intent of the method, and then use the closeness as an indicator of how important the particular usage is to the parameter's major purpose. The open questions are: (1) How do we measure "closeness" of a parameter use to the computational intent of a method? (2) In order to measure closeness, how do we represent a method's computational intent? (3) How do we use closeness to identify the main role of a parameter in the method?

**Identifying Linking Context to Computational Intent.** According to the Java documentation writing guidelines for a method [5], developers should write a summary of the method's computational intent followed by comments for each parameter, followed by comments about the return values, exception handling and so on. Thus, in a good system, a parameter's comments are always associated with a summary. Therefore, an important challenge is *not only producing* parameter comments but *ensuring* that they are placed *in context* of the method's summary. We call the variables and statements that link a particular parameter's usage in its main role to the method summary the *linking context* for that parameter. Sometimes, the linking context is already identified as part of the statement containing the parameter's main role, while other times it involves additional statements within the method.

Thus, determining the linking context presents several challenges: (1) Which code artifacts, including individual variables and statements, provide the linking context for a particular parameter? (2) How do we automatically identify these linking artifacts? (3) How can we generate natural language phrases that link a parameter comment with a method summary using the linkage context information? (4) How do we ensure that the overall generated leading comment is not too verbose? The text generation, challenge (3), can leverage our previous work on automatic summary generation [2], with minor modifications; however, the remaining challenges are significantly different from automatic summary generation.

## III. Foundations

Our automatic parameter comment generator uses both structural and linguistic information extracted from a given method's signature and body. This information is obtained by leveraging existing techniques [6]–[8]. This section presents an overview of how that information is extracted and represented, as well as our representation of the computational intent.

### A. Structure and Linguistic Information

We use information available in several common program representations in combination with information from naming conventions and linguistic knowledge gained from observations of thousands of Java programs. In particular, we use information from the control flow graph, control and data dependences (def-use chains), along with textual clues which we obtain from the Software Word Usage Model (SWUM) [6] of the program.

Identifiers must be split into component words, before any word usage information can be extracted from names used in the program. We use camel case splitting, which splits words based on capital letters, underscores, and numbers (e.g., childXMLElement would be split into "child XML Element"), and aspects of more advanced splitting [7]. As in any system that uses linguistic information, our technique 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 [9].

Readability of the generated descriptions and the accuracy of our analysis can be reduced due to abbreviations in variable and type names (e.g., Button butSelectAll, MouseEvent evt). We use techniques from prior work [8] to automatically identify and expand abbreviations in code.

The Software Word Usage Model (SWUM) [6] provides us with the necessary linguistic information beyond individual word frequencies necessary to generate parameter comments. SWUM not only captures the occurrences of words in code, but also their linguistic and structural relationships. SWUM has been successfully used in concern location, summary comment generation and high-level action identification for Java methods [1], [2], [10].

Particularly, we use SWUM to obtain the action, theme, and optional secondary arguments of a method to generate succinct and smooth descriptions, and, in conjunction with program structure, we use this information to identify the statements that need to be used to describe a parameter, and, statements that need to be used to integrate the parameter comments with the computational intent of a 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". Further, in this example, the location of the theme is the given parameter while the location of the secondary argument is the receiver object. In addition to these locations, a theme or secondary argument can be the method name itself (e.g., buildMenu()).

### B. Automatically Determining the Computational Intent

To determine closeness of a given parameter use to the computational intent of the method, we need a representation of the method's intent. In particular, we need to identify the code statements that provide the content for a high-level overview of the method's intent. Developer-written comments could potentially be used to represent the method's intent. However, automatically determining that a leading comment is a summary comment, as opposed to other types of comments and mapping the developer-written summary back to individual statements within the method body is not a trivial task. Further, for many methods, there are no developer-written summaries.

Thus, to determine the intent of a method, we automatically generate a summary for the method using the summary comment generator defined previously by us in [2]. Using the structural and linguistic information from a given method, our summary generator selects the content that should be present in a summary and generates succinct

```
1 void buildResourceItem(ResourceType r, TreeNode parent) {
2  ImageIcon icon=library.getScaledBonusImageIcon(r,0.75f);
3  TreeNode n= new TreeNode(new TreeItem(r,r.name,icon));
4  parent.add(n);
5 }
```

Figure 2. Link To Summary Via Variable in Summary Phrase (Variable n) : We choose Line 3 and not Line 2 to describe the parameter r

natural language phrases to describe the selected content. We use the generated summary as a representation of the method's intent.

## IV. Generating Parameter Comments

To gain insight into closeness and linking context, consider the example method in Figure 2 which has two parameters parent and r. The computational intent is approximately captured by the generated summary *"add tree node to the given parent tree node"*, which is derived from Line 4.

The parameter parent already appears in the summary phrase and thus nothing needs to be done to determine the major role and integrate the parameter comments for parent with the intent of the method. The parameter r is used on both Lines 2 and 3. On Line 3, the parameter r is used to define the variable n, which appears in the summary. We generate the phrase "create tree node using the given resource type" for Line 3. The noun phrase "tree node" in this phrase overlaps with the noun phrase "tree node" in the method summary, thereby integrating closely with the generated summary already. For Line 2, we would generate a phrase such as "get image icon from library using the given resource type". This phrase has no common words with the summary. To link Line 2 with the summary, we would have to include Line 3 as its linking context. In contrast, Line 3 links to the summary without the need for another line. Thus, we consider the parameter usage on Line 3 closer to the computational intent of the method than Line 2.

**Approach.** The major steps of our approach are depicted in Figure 3. The input is a Java method along with the method's structural and linguistic representations, which are described in Section III. The input also includes a representation of the computational intent of the method. As described in Section III-B, we use the generated summary for the method [2] to represent the intent. Along with the actual phrase(s) constituting the summary, the input also contains the *sUnit(s)* from which the phrase(s) were generated. We use the notion of an sUnit defined previously in [2] as a Java *statement*, except when the statement is a control flow statement; then, the sUnit is the *control flow expression* with one of the *if, while, for* or *switch* keywords.

We first use def-use information to identify all uses of the parameter. To identify the *most important* uses, we first prune the uses that can be disqualified as irrelevant to the parameter's major role in the method (Step 2 in Figure 3). Second, we estimate closeness of the remaining uses to the computational intent along with identifying linking context information (variables and statements) (Step 3 in Figure 3). We select the closest sUnit and generate phrases for the parameter comment and integrated summary comment using the linking context information. At this point, rules for generating more concise comments can be applied as desired by the tool user. The following subsections describe the *major* steps in our approach - pruning, closeness and linking context identification, phrase generation and phrase transformation (i.e., steps 2, 3, 5 and 6).

**Input:**
1. Method M (signature + body)
2. Structural and Linguistic Representations of M
3. Computational Intent of M
   (Represented by the Generated Summary for M)

**Output:**
Comment phrases for each formal parameter of M which are connected to the computational intent of M.

**Process:**
For each formal paramter fp of M:

**Step 1:** Find AllUsesSet i.e. all sUnits in which fp is used.

**Step 2:** If needed, prune any unimportant sUnits from AllUsesSet such that at least one sUnit remains.

**Step 3:** For each remaining sUnit:

Estimate how close the sUnit is to M's computational intent.
Also identify linking context information
(i.e., additional sUnits and variables needed for linking the sUnit to the computational intent)

**Step 4:** Select sUnit (s) closest to M's computational intent

**Step 5:** Generate phrases for the selected sUnits and any additional sUnits in the linking context information.

**Step 6:** Apply transformations to create concise parameter comments, depending on user choice

**Step 7:** Depending on user preference, place the comments :
1. As an @param comment
2. As an addition to the existing summary for M

Figure 3. Overview of Approach: Generate Parameter Comments and Integrate with Method's Computational Intent

### A. Step 2: Prune Uses

This step performs the first part of finding the *most important* uses of a parameter. The main heuristic behind pruning is to prune uses of the formal parameter that are less likely to be executed from consideration as the major role in the method. We statically estimate the relative execution frequency by adapting some of the heuristics described in [11]. For example, the heuristic *references are typically not null* prunes based on known predicates, i.e., given an if statement of the form if (x != null) ... else ..., the then block is more likely to execute than the else block. Similarly, given if($x < 0$) ... else ..., the else block is more likely to execute. In Figure 4, the formal parameter guiName is used on Lines 2, 4, 6 and 8. The primary role of the parameter is in the definition of the mainGUI object, on Line 4 (Line 4 gets the GUI to be started). Line 6 will be executed *if and only if* mainGUI is null. Hence, according to [11], Line 6 is less likely to execute and will be pruned.

Other examples that indicate relative frequency of execution are an if or else block that (1) ultimately throws

```
1  void startGUI(String guiName, String[] args) {
2   assertTrue(guiName != null, "guiName must be non-null");
3   assertTrue(args != null, "args must be non-null");
4   IMegaMekGUI mainGui = getGui(guiName);
5   if (mainGui == null)
6    displayMessageAndExit(UNKNOWN_GUI_MESSAGE+guiName);
7   else {
8    StringBuffer message = new StringBuffer("Starting GUI"+guiName);
9    dumpArgs(message, args);
10   displayMessage(message.toString());
11   mainGui.start(args);
12  }
13 }
```

Figure 4. Eliminating a use of the formal parameter (Line 6) in an infrequently executed block (Line 5).

an exception or (2) returns the formal parameter *without* modifying it (i.e., there are no assignments to the parameter before it is returned).

Additionally, we use some heuristics similar to those used to remove unnecessary sUnits from a method summary [2]. We select those sUnits that call a method with the *same action* as the method under analysis and prune other sUnits. For example, in Figure 5, the parameter shell is used on Lines 2 and 3. However, the action for the call on Line 2 is the same as the action of the method (i.e., *remove*). Thus, we delete Line 3. We also prune sUnits that deal with logging, exception handling and sanity checks. After all these heuristics are applied, we prune sUnits containing an outer-most method call that is a *get, set or a constructor*. If at any time during pruning, a heuristic would prune the only remaining use, we keep the last use not pruned yet, so there is at least one use for generating a parameter comment.

### *B. Step 3: Estimate Closeness to Method's Intent and Identify Linking Context*

This step performs the second part of finding the *most important* uses of a parameter. In order to do so, we estimate the closeness of a given parameter use to the method's computational intent. This step also finds the linkage context. We describe the closeness estimation heuristics here. The cases are described from closest to computational intent to farthest, with increasingly more linking context to be generated.

**1: Parameter already appears in the summary text.** Here the chosen sUnit for parameter comment has also been selected for inclusion in the method summary by the heuristics in [2] and the generated phrase for the summary already mentions the parameter. Hence, the parameter is already integrated with the summary.

We can use the generated phrase for the sUnit *as is* for the parameter comment or transform the phrase into another form as described in Section IV-D. Figure 5 illustrates this case. The generated summary already has the phrase "remove the given shell from the list, shells" which contains the input parameter shell. We can use this phrase as the parameter comment or using the transformation templates defined in IV-D, we can generate "@param shell: which is removed from the list, shells". Note that the Java documentation writing guidelines [5] suggest that the @param comments should be written for a parameter even when its description is obvious.

```
1 void removeWindow(Shell shell) {
2    shells.remove(shell);
3    notifyRemoveListeners(shell);
4 }
```

Figure 5. Parameter already in summary phrase; we choose same-action use (Line 2) and not Line 3

```
1 ParserResult importToOpenBase(String argument) {
2  ParserResult result=importFile(argument);
3  if (result != null)
4     result.setToOpenTab(true);
5  return result;
6 }
```

Figure 6. Parameter in summary sUnit (Line 2) but not in summary phrase

**2: Parameter is used in an sUnit selected for the summary.** This case is similar to case 1, with the following difference: The parameter is used in an sUnit which has been selected for the method summary, but does not appear in the text phrases constituting the final summary.

Consider Figure 6. The underlined portions represent the content chosen for the summary. In the method call importFile on Line 2, the theme *File* of the action *import* is present in the method name itself. In such cases, a phrase such as "import file and get parser-result" is sufficient for a concise summary, though argument was omitted.

However, as far as the formal parameter argument is concerned, Line 2 represents an important use as the sUnit on Line 2 has been selected for the summary. We generate the @param comment "import file *using the given argument string* and get parser result". This provides a link to the summary and describes the parameter role in the method.

**3: A ubiquitous method.** Typically a summary comment for a method does not include ubiquitous operations such as a *setter* for brevity. However, the *most important* use of a parameter can be in such an ubiquitous method (e.g., Line 4 in Figure 7). In such cases, we generate a phrase for this important parameter use but do not integrate the generated parameter comment phrase with the summary. In Figure 7, the summary consists of the underlined sUnits. The role of the parameter c is not immediately obvious from the signature alone. From Line 4, where the parameter is used, we generate the stand-alone parameter comment "set mnemonic to the given character for menu item".

**4: Link to summary via variable in summary phrase.** In this case, the parameter is used in an sUnit to define another variable (i.e., the parameter is used on the Right Hand Side of the = operator). The defined variable is then used in another sUnit which has been selected for the summary, and summary phrase includes the defined variable. Figure 2 shows an example of this case. The linking context

```
1 void addMenuItem(String name, char c) {
2    JMenuItem mi = new JMenuItem();
3    mi.setText(name);
4    mi.setMnemonic(c);
5    mi.addActionListener(this);
6     _thisMenu.add(mi);
7 }
```

Figure 7. Important use of parameter (c) occurs in an ubiquitous method (line 4)

```
 1 void loadParseURL(String newURL,String cookie,
    CleanupHandler cl) {
 2  m_parser = new JHTMLParser(this);
 3  StringBuffer loadedPage;
 4  try {
 5   URLConnection uc = Http.getPage(newURL, cookie, null);
 6   loadedPage = Http.receivePage(uc);
 7   if(loadedPage != null) {
 8      if(cl != null)
 9        cl.cleanup(loadedPage);
10      m_parser.parse(loadedPage);
11      m_loaded = true;
12    }
13   } catch(IOException e) {
14    loadedPage = null;
15    ErrorManagement.handleException(e);
16  }
17   if(loadedPage == null) m_loaded = false;
18 }
```

Figure 8. Link to summary via variable (uc) in summary sUnit but not in Summary Phrase (Line 6). We use Line 5 to describe the parameter, newURL. We augment the summary phrase on Line 6 for linking.

includes the variable r. We generate the phrase, "create tree node" using the phrase generation templates in [2]. We then augment this phrase to include the linking context variable, "create tree node, using the given resource type".

**5: Link to summary via variable in summary sUnit but not in phrase.** The difference between this case and case 4 is analogous to the difference between the cases 1 and 2. Here, the variable defined using the formal parameter, is used later on in an sUnit selected for the summary, but does not appear in the summary phrases. We now generate a phrase for the sUnit in which the parameter is used and augment the summary to include the variable defined using the formal parameter.

Consider Figure 8. The underlined portions represent the content chosen for the summary. The parameter newURL is used on Line 5 to define the variable uc. Thus, we generate a phrase for Line 5, "get URL connection from Http using the given newURL". The linking context includes the variable uc which is used on Line 6. We augment the phrase generated for Line 6 to include uc, "receive loaded-page from Http *using the URL connection*".

**6: Link to summary via intermediate variables.** This case can be viewed as a generalization of case 4 except that one or more intermediate statements are in the linking context. For example, consider the Figure 1 in Section I. The generated summary is "start meta server", which comes from Line 19. On Line 4, the parameter args is used but the variable assigned on Line 4, port, is not directly used in Line 19. However, the variable metaServer, which is in the sUnit chosen for the summary, is defined on Line

```
1 void addGoods(Goods g,boolean enabled,boolean indent) {
2  String text = (indent? "  ":"") + g.toString();
3  JMenuItem mi = new JMenuItem(text);
4  if (indent)
5   mi.setFont(mi.getFont().deriveFont(ITALIC));
6  if (!enabled)
7   mi.setEnabled(false);
8  add(mi);
9  hasAnItem = true;
10 }
```

Figure 9. Use in a Conditional Expression not part of the Summary (line 6). We use Line 6 and 7 to describe the parameter, enabled

14 by using port. We call port an intermediate variable. Thus, we generate the following @param comments with the following phrases from Lines 4 and 14: "parse integer and get port using the given arguments. create meta server using the port". This provides the reader with a better idea of the role of the parameter args than the phrase produced only from the sUnit on which it is used (i.e., Line 4). Note that the rules for generating more concise parameter comments can be applied to this generated comment as desired by the tool user.

We use *def-use* chains to automatically establish the link between an input parameter and a variable that appears in the summary. Note that if there is a link to a variable which is in an sUnit selected for the summary, but does not appear in the summary phrase, then we augment the summary phrase to include the intermediate variable.

**7: Use in a conditional expression not part of the summary.** Consider Figure 9 for which the underlined sUnits represent the content for the summary. Often, conditional expressions such as the one on Line 6 in Figure 9 are not included in the summary, for conciseness. However, the important role of a parameter could be in such an expression. In this example, the important role of the parameter enabled is on line 6 and the role is not clear from the signature.

Generating a phrase utilizing *only* the sUnit using the parameter (Line 6 in Figure 9) is *not sufficient* to describe the role of the parameter enabled as it gives a reader *no idea* of what is done based on whether enabled is true or false. Thus, we *utilize sUnits within* the if block to describe the parameter. Currently, we select the last sUnit in the then block. We generate, "if enabled flag is false, set enabled flag to false for menu item", which is more meaningful than generating a phrase from Line 6 alone. We use a similar heuristic for a parameter that appears in a looping expression when the loop expression is not a part of the summary.

**Illustrative example.** We will now describe how the concept of closeness allows the selection of Line 3 rather than Line 2 for the parameter r in Figure 2. For the sUnit on Line 3, there is a link to the summary phrase via the variable n. In contrast, for the sUnit on Line 2 for parameter r, there is a no direct link to the summary, but from r, we define icon on Line 2 and from icon, we define n on Line 3. Thus, there is a link via the intermediate variable n (i.e., via the variable

```
fp = formal parameter; VP = Verb Phrase; SA = Secondary Argument;
t = transformed phrase; prep = preposition; rem = remaining;
pastPart = past participle

1 IF fp corresponds to theme in the generated VP AND VP has an SA THEN
    t= which is <pastPart(action)> <prep in SA> <rem SA>
2 ELSE IF fp corresponds to theme in the generated VP  THEN
    t= which is <pastPart(action)>
3 ELSE IF fp corresponds to SA in the generated VP
    t= <prep in SA> which <theme> is <pastPart(action)>
4 ELSE IF fp does not correspond to theme/SA  AND VP has an SA
    t= using which,  <theme> is <pastPart(action)> <prep in SA> <rem SA>
5 ELSE IF fp does not correspond to theme/SA
    t= using which,  <theme> is <pastPart(action)>
```

Figure 10. Relative Clause Transformation Templates

chain r, icon, n to the computational intent).

Our heuristics consider the parameter use on Line 3 to be closer to the computational intent of the method than Line 2. Thus, we select the sUnit on Line 3 ahead of the one on Line 2 to generate the parameter comments for r.

### C. Step 5: Generate Phrases

Step 4 selects the sUnit(s) closest to the method's intent as determined in Step 3. In this step 5, we generate a phrase for the selected sUnits and any additional sUnits required to link the parameter comments with the method intent. We utilize the phrase generation templates defined in [2] with some modifications to generate phrases. Phrase generation also involves variable lexicalization [2], in which descriptive noun phrases describing variables are generated with modifiers extracted through type information. For example, a variable current is transformed into the more descriptive noun phrase, *current document*, based on the type of current.

### D. Step 6: Apply Phrase Transformations

In a summary, a selected sUnit is described by a verb phrase (VP). However, according to convention [5], parameter comments are written such that the *emphasis* is on the parameter. Therefore, we apply transformations to the verb phrase that involves the usage of relative clauses to put the emphasis on the parameter. The current transformation templates are shown in Figure 10.

Consider the following sUnit mb.add(m) in which *m* is a formal parameter for the method under analysis. For this sUnit the verb phrase "add the given menu to menu bar" [2] is generated. Applying template 1 of Figure 10 transforms the phrase into "@param m: which is added to menu bar".

Consider Figure 1 in which we used the variable chain, args, port, metaServer to establish a link between the input parameter args and the method's intent. While we can generate verb phrases for Lines 4 and 14, depending on user preference, we can also merge the phrases into a single phrase for a concise comment.

Each phrase in the different verb phrases will have a verb; the challenge is to determine which verb among the multiple verbs should be used in the resulting single phrase. We use the leading verb in the verb phrase for the sUnit in which the last variable in the chain is assigned (Line 14 in Figure 1). Thus, for Figure 1, we generate "create meta server using the input arguments". This can be further transformed into "@param args: using which, meta server is created", by applying template 5 of Figure 10.

### E. Additional Parameter Comments

In addition to descriptive comments for parameters, we are able to extract and add the following information to @param comments. Often, the actual parameter value across different call sites can follow a pattern. For example, the actual values might always be null or the same string value. Consider the method addToolTo from the open-source project JHotDraw. Across 53 call sites, the actual parameter value for the fourth parameter, label, always begins with the word *create*. Such a fact might be useful in alerting a caller of the method that there is some uniformity in the string parameter and he must adhere to the convention. We scan the call sites and check if the actual values are always null, the same numerical value (e.g., 0) or the same boolean value. For string parameters, we check if the actual call site values are always the same string literals or the leading words are the same, after splitting the string literals into their constituent words [7].

Often the formal parameters to a method are always used together in one or more sUnits. For example, in Figure 8, the parameters newURL and cookie are used together. We mention this fact in the generated comments.

## V. Evaluation

We implemented the heuristics described in Section IV as an Eclipse plug-in. We leveraged the Eclipse Java Development Tools to provide the program structural information, an implementation of SWUM [6] for linguistic information, as well as an abbreviation expander [8] and identifier splitter [7]. We focused on the following questions:

- What is the accuracy of the generated parameter comments?
- What is the utility of the parameter comments in aiding a programmer in obtaining a high-level overview a parameter's role in a method?
- In helping a developer gain a high-level overview of the computational intent of a method, what is the *utility* of the phrases that we add or modify to integrate the parameter comments with the method summary?
- Within the context of the summary, what is the *necessity* of the phrases that we add or modify to integrate the parameter comments with the summary?

Our goals are to generate accurate parameter comments, provide important information about the high-level role of a parameter, augment the summary with important phrases when integrating the parameter comments with the summary and avoid unnecessary additions/modifications to the summary during the integration.

**Procedure.** We asked nine human evaluators to judge the generated comments and answer the above questions. The

programming experience of this group ranges from 4 to 20 years, with a median of 12 years. All the evaluators considered themselves as expert or advanced programmers. Four evaluators have software industry experience ranging from 1 to 7 years.

We ran our prototype on methods from six open-source Java projects from across different domains. Table I shows some characteristics of these projects. We generated summary and parameter comments for all the methods in these projects, and randomly selected methods for human judgement. The evaluation task requires an evaluator to read the entire method to answer the evaluation questions. Thus, to avoid burdening the evaluators, we restricted the methods to have at most 25 sUnits. We also did not choose methods with less than ten sUnits as many such methods can be read quickly to understand the role of the parameters. We avoided methods in which all the formal parameters already appeared in the generated summary phrases. We also avoided methods in which the role of a parameter is obvious such as constructors, comparison methods (e.g., boolean equals(Object o1, Object o2), methods for which the underlying SWUM rules and summarization heuristics are being developed.

| Project | #Methods | KLOC | Project | #Methods | KLOC |
|---|---|---|---|---|---|
| Freecol | 5971 | 110 | JBidwatcher | 1877 | 30 |
| GanttProject | 4956 | 60 | JHotDraw | 4267 | 63 |
| Jajuk | 2139 | 44 | MegaMek | 9256 | 200 |

Table I
SUBJECT PROGRAMS IN STUDY. KLOC: 1000 LINES OF CODE.

18 methods were judged by the developers. The 18 methods had 33 parameters in all. In integrating the parameter comments with the method summaries, the automatic parameter comment generator added 22 new phrases to existing summaries. In addition, it augmented 12 existing summary phrases with additional information leading to a total of 34 modified or added phrases.

We gave each evaluator six methods to examine and to account for subjectivity in the evaluation, we gathered three independent opinions per method. Thus, we obtained 54 independent judgements on 18 methods by 9 developers evaluating independently in groups of 3. To control for any learning effects, the evaluators in a group did not see the methods in the same order.

We showed evaluators the body of each method assigned to them along with the generated summary and parameter comments. We highlighted the 34 phrases in the summary that were added/modified to integrate the parameter comments with the summary. To avoid bias, we deliberately did not provide an explicit definition or examples of a summary/parameter comment. The evaluators were allowed to use any resource to help in the evaluation. Table II shows the questions and answer choices shown to the evaluators. The questions on *Accuracy* and *Utility-Standalone* were repeated for each formal parameter of the method. *Necessity* was asked for each of the 34 phrases added to/modified in the summary.

**Threats to Validity.** Our results may not generalize to other Java programs. To mitigate this threat, we chose six large open source programs across different domains representative of typical Java programs. Our conclusions might not hold with other languages or longer methods. In the future, we will address portability to other languages and perform studies with longer methods. It is possible that our results may not be applicable to beginner Java programmers since we had no such evaluators in our study. We plan to conduct a study with such programmers. Finally, we did not compare with developer-written parameter comments as such comments can convey information that cannot be obtained from code (i.e., domain knowledge), and there is no obvious unbiased way of identifying methods with such comments.

## VI. RESULTS

***Accuracy.*** Table III shows the individual developer responses and the majority opinion for *Accuracy*. The results strongly suggest that the parameter comments that we generate are indeed accurate. In 89 of the 99 responses (for 33 parameters), developers said that the generated parameter comment was accurate. When we consider the majority opinion, for 32 of the 33 parameters, a majority of the developers said that the comments were accurate. In the remaining case, the "slight inaccuracy" was not due to the parameter per se, but due to the majority of the developers feeling that the comment had to include additional information given by another variable appearing in a method call in the selected sUnit.

| | Accurate | Slightly Inaccurate | Inaccurate |
|---|---|---|---|
| **Individual Responses** | 89 | 9 | 1 |
| **Majority Opinion** | 32 | 1 | 0 |

Table III
DEVELOPERS' OPINION : ACCURACY OF PARAMETER COMMENTS

***Utility-Standalone.*** The second column of Table IV shows the opinions of the developers on the utility of the generated parameter comments in providing a high-level overview of the role of the parameter in the method. 47 of the 99 responses indicate that the generated parameter comments provided critically important information, while 42 responses indicate that the comments provided important information. If we consider the majority opinion per parameter, then for *all* 33 parameters, the majority said that the generated comments provided important or critically important information. We believe these are promising results for automatic parameter comment generation.

| Criteria | Question | Answer Choices |
|---|---|---|
| *Accuracy* | What is your opinion on the accuracy of the parameter comments? | • Accurate<br>• Slightly inaccurate<br>• Inaccurate |
| *Utility-Standalone* | What is your opinion on the parameter comments in terms of helping you gain a high-level overview of the *role of the parameter in the method*? | • Critically important<br>• Important<br>• Neither important nor unimportant<br>• Not important<br>• Detrimental |
| *Utility-Integrated* | What is your opinion on the highlighted text in the summary in terms of helping you gain a high-level overview of the *computational intent of the method*? | Same as above |
| *Necessity* | What is your opinion on the necessity of each highlighted phrase in the summary? | • Necessary<br>• Probably unnecessary (but tolerable)<br>• Definitely unnecessary |

Table II
QUESTIONS AND ANSWER CHOICES SHOWN TO EVALUATORS

***Utility-Integrated*.** The third column of Table IV shows the opinions of the developers on the utility of the added/-modified phrases to the summary in providing a high-level overview of the method's intent. 41 of the 54 opinions suggest that the added comments to the summary provided important or critically important information towards gaining a high-level overview of the computational intent of the method. For 15 of the 18 methods, the majority felt that the added comments to the summary due to the parameters provided important or critically important information. For *no method*, did the majority feel that the added comments were "not important" or "detrimental". In 2 of the 18 methods, a majority felt that the added comments were "neither important nor unimportant". For one method, the three opinions ranged from "not important" to "important". We analyzed these three methods, and the reason for the evaluators' responses are as follows: In the method URLConnection makeRequest(URL source, String cookie), the parameter comment for source is "using which, url-connection is opened". However, the summary already contains the phrase "open url-connection". Once the summary is seen, it is obvious to an experienced developer that an URL connection will be opened using the parameter source and hence the evaluators felt that the modification to the summary phrase had no impact. There were three such methods where the role of the parameter was *obvious* from the summary/method signature and thus the comments added to the summary to depict the parameter role were not perceived by the majority to be "important" or "critically important" to understand the method's intent.

| Response | Parameter Role | Method Intent |
|---|---|---|
| 5: Critically Important | 47 | 16 |
| 4: Important | 42 | 25 |
| 3: Neither important nor unimportant | 4 | 10 |
| 2: Not important | 5 | 2 |
| 1: Detrimental | 1 | 1 |
| Total | 99 | 54 |

Table IV
DEVELOPERS' OPINION : UTILITY OF COMMENTS FOR : OVERVIEW OF PARAMETER'S ROLE, OVERVIEW OF METHOD'S INTENT

***Necessity*.** Table V shows the opinion of the programmers on whether the phrases that were added to or modified in the summary to integrate the parameter comments with the summary were necessary at the level of the summary. We had 34 phrases that were added or modified due to the integration feature, and of the 100 opinions[1], 69 said that the addition was necessary. Only one opinion suggested that the addition was definitely unnecessary. According to the majority, *no phrase added or modified was definitely unnecessary*; 25 of 34 phrases were definitely necessary. For 9 of the 34 phrases, a majority felt that the the added or modified phrases were probably unnecessary (but tolerable) given that the purpose of a summary is to provide a high-level overview. The reasons for evaluators feeling that the phrases were probably unnecessary were due to the same factors as explained in the paragraph for *Utility-Integrated*. We view these results as quite positive overall for automatic parameter comment generation.

| | Necessary | Probably Unnecessary but Tolerable | Definitely Unnecessary |
|---|---|---|---|
| **Individual Responses** | 69 | 30 | 1 |
| **Majority Opinion** | 25 | 9 | 0 |

Table V
DEVELOPERS' OPINION : NECESSITY OF ADDING THE PARAMETER COMMENT PHRASES TO THE EXISTING SUMMARY

**Summary of Results.** The results from our study strongly suggest that our system has met the stated goals of generating accurate parameter comments which help a developer gain a high-level perspective on the role of a parameter in a method. The system has also met the goal of integrating the parameter comments with the summary such that a developer gains a quick overview of the intent of the method.

## VII. RELATED WORK

Studies suggest that well written comments can help in program comprehension [12], [13]. Unfortunately, studies also suggest that few software projects adequately document

[1]One evaluator did not evaluate two phrases.

the code to reduce future comprehension and maintenance costs [14], [15].

Fowler advocates using extremely descriptive identifier names to obviate comments [16]. Unfortunately, studies suggest that precise identifiers that accurately describe an entity lead to very long identifier names which *hinder* code readability [9], [17].

There has been some limited work on comment generation. Semi-automated approaches either automatically determine uncommented code segments and prompt developers to enter comments [18], [19], or automatically generate comments from high level abstractions which the programmer provides during development [20]. Although useful for newly created systems, none of the semi-automatic techniques apply to existing legacy systems.

Previously, we presented heuristics to automatically generate comments summarizing a given Java method [2]. Our comment generation focused on identifying important content for the method summary and then generating text for each selected statement in isolation. The emphasis was on producing a succinct summary and hence the generated summary need not contain any reference to the formal parameters of the method.

We also presented a technique for identifying and describing high-level actions in statement sequences, conditionals and loops [10]. However, those heuristics are not applicable to this problem.

In addition, we are aware of other techniques that could be used towards generating comments for legacy code [21]–[24]. However, these approaches are limited to inferring documentation for exceptions [21], generating API function cross-references [22], synthesizing method stereotype information [23] or producing documentation for program changes between versions [24]. The possibility of using natural language text summarization techniques to summarize source code has been explored in [25]. We believe that none of these techniques are intended for generating descriptive parameter comments in relation to the method's intent.

Harman et al. defined *key statements* as statements through which the largest part of the program's dependence appears to flow [26]. The notion of key statements do not necessarily help in identifying the main role of a parameter.

## VIII. Conclusions and Future Work

To the best of our knowledge, we present the first technique to automatically generate comments for Java method parameters, such that the comments provide a high-level overview of the role of the parameter in facilitating the envisaged functionality of the method. According to nine experienced developers, the generated comments are accurate and are useful in providing a quick perspective on the parameter's purpose in accomplishing the method's intent.

We plan to continue to improve our system by analyzing additional methods with varying features. We will expand our evaluation to include longer methods and inexperienced programmers. We plan to explore porting our system to other languages such as C++.

## References


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

[2] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *IEEE/ACM Intl conf on Automated Software Engineering*, 2010.

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

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

[5] SUN, "How to Write Doc Comments for the Javadoc Tool," online, http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

[6] E. Hill, "Integrating Natural Language and Program Structure Onformation to improve Software Search and Exploration," Ph.D. Dissertation, University of Delaware, 2010.

[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] 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," in *5th Intl Working Conf on Mining Software Repositories*, 2008.

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

[10] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically Detecting and Describing High Level Actions within Methods," in *Intl Conf on Software Engineering (ICSE'11)*, 2011, to Appear.

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

[12] 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.*, vol. 4, no. 3, 1996.

[13] T. Tenny, "Program Readability: Procedures Versus Comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, 1988.

[14] M. Kajko-Mattsson, "A Survey of Documentation Practice within Corrective Maintenance," *Empirical Softw. Engg.*, vol. 10, no. 1, pp. 31–55, 2005.

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

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

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

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

[19] D. Roach, H. Berghel, and J. R. Talburt, "An Interactive Source Commenter for Prolog Programs," *SIGDOC Asterisk J. Comput. Doc.*, vol. 14, no. 4, 1990.

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

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

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

[23] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in *22nd Intl Conf on Software Maintenance (ICSM)*, 2006.

[24] R. P. Buse and W. R. Weimer, "Automatically Documenting Program Changes," in *IEEE/ACM Intl conf on Automated Software Engineering*, 2010.

[25] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of Automated Text Summarization Techniques for Summarizing Source Code," in *Working conf on Reverese Engineering (WCRE '10)*, 2010.

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