# Using Stereotypes in the Automatic Generation of Natural Language Summaries for C++ Methods

Nahla J. Abid[1], Natalia Dragan[2], Michael L. Collard[3], Jonathan I. Maletic[1]

[1]Department of Computer Science
Kent State University
Kent, Ohio 44242, USA
{nabid, jmaletic}@kent.edu

[2]Department of Mang. and Info Systems
Kent State University
Kent, Ohio, USA
ndragan@kent.edu

[3]Department of Computer Science
The University of Akron
Akron, Ohio
collard@uakron.edu

***Abstract*—An approach to automatically generate natural language documentation summaries for C++ methods is presented. The approach uses prior work by the authors on stereotyping methods along with the source code analysis framework srcML. First, each method is automatically assigned a stereotype(s) based on static analysis and a set of heuristics. Then, the approach uses the stereotype information, static analysis, and predefined templates to generate a natural-language summary for each method. This summary is automatically added to the code base as a comment for each method. The predefined templates are designed to produce a generic summary for specific method stereotypes. Static analysis is used to extract internal details about the method (e.g., parameters, local variables, calls, etc.). This information is used to specialize the generated summaries.**

***Index Terms*—source-code summarization, program comprehension, method stereotypes, static analysis.**

## I. INTRODUCTION

During the evolution of large software systems, developers spend a great deal of time trying to locate and understand the parts of the system that are related to their current task [1-3]. A typical practice for developers is to alternate between the source code and the associated internal documentation (aka comments) to understand the intent of a method or function. Furthermore, comments are found to be critical in assisting developers to understand code faster and more deeply. Unfortunately, comments are oftentimes outdated or incomplete [3] due to lack of either proper maintenance or adherence to proper documentation standards.

Recently, several techniques have been proposed to automatically generate summaries directly from source code methods [4-9] and classes [4, 7, 10]. Natural language processing (NLP) techniques have been used to generate English descriptions for Java methods [5, 6]. These approaches analyze the part-of-speech tag of identifiers (method or variable names) to capture the intent of methods. While these approaches produce some reasonable results, their performance depends upon high-quality identifiers and method names. For example, when grouping method calls, NLP requires that all method names starts with the same verb [6]. Furthermore, it is observed that the technique often selects lines of code that are considered essential for the method, but does not generate English descriptions for these lines [9]. Consequently, if identifiers and methods are poorly named, the approach may fail to generate accurate comments or any reasonable comments at all.

In the approach taken here we do not depend on NLP methods. Instead, the summaries (for methods) are generated using predefined fill-in-the-blank sentence phrases, which we refer to as templates. We constructed the templates specifically for different method stereotypes. Stereotypes reflect the basic meaning and behavior of a method and include concepts such as *predicate*, *set*, and *factory*. In previous work by the authors on method stereotypes, a fully automated approach to assign stereotypes to methods was developed and evaluated [11].

Here we leverage that stereotype information and construct custom template phrases for each stereotype to form the summaries. After the appropriate templates are matched to the method, they are filled with data to form the complete summary. To fill in the templates, static analysis and fact extraction on the method is done using the srcML [12] infrastructure (www.srcML.org).

The generation of the summaries is fully automated. The summaries start with a short and precise description of the main responsibility of the method. Following is additional information about external objects, properties modified, and a list of function calls along with their stereotypes.

The summaries can improve source-code comprehension and speed up maintenance in several ways. First, the short description assists developers in understanding the main behavior of the method and conclude its relevancy to their current task. Second, the documentation summary describes the main elements of a method (e.g., external objects, data members and parameters modified) in a structured format that can be easily mapped to the code. Third, because the main side effect of the method is often caused by one or multiple calls, including an abstract summary of calls (the stereotype of calls) provides additional information about their roles.

While the generated summary may not be a complete substitute for developer-written summaries, they do organize the method's main ideas in a structured manner and provide additional information about the responsibility of calls.

The main contributions of this work demonstrate that method stereotypes can directly support the process of automatically generating useful method summaries in an efficient manner. The approach also demonstrates that a template-based technique, specialized for each stereotype, combined with simple static analysis is a practical approach for automated documentation.

The remainder of this paper is organized as follows. The next section gives related work on source-code summarization. Then, the approach is given in section III. Sections IV and V describe the heuristics of method stereotype identification and documentation generation, respectively. Finally, conclusions and future work are presented in Section VI.

## II. Related Work

Automated summarization of source code has been a topic of recent interest for multiple researchers. Sridhara et al. proposed techniques to automatically generate natural language comments for Java methods [5], sequences of statements [6], and formal parameters [13] using NLP [14]. McBurney and McMillan [9] proposed generating documentation summaries for Java methods that include information about the context of methods using call graph information and NLP.

Moreno et al. [10] use method stereotypes and class stereotypes [15] to generate natural-language summaries for Java classes. The approach uses method and class stereotypes to filter out irrelevant methods to the class stereotype. While we also use method stereotypes as applied in [10], the main contribution of our paper is to generate documentation summaries for methods, not classes. Thus, we generate a much finer granularity of documentation then that for a class. Our technique uses method stereotypes to generate a standard summary for each method.

Haiduc et al. [4] investigated the suitability of several text summarization techniques (TR) to automatically generate term-based summaries for methods and classes. This was further extended by Eddy et al. [7] using a new TR technique named Hierarchical PAM. It was concluded that TR might provide better results if it was combined with static analysis.

Rodeghero et al. [16] conducted an eye-tracking study to determine the statements and terms that programmers view as important when they summarize a method.

Mapping existing human descriptions to source code segments was used to generate summaries for Java and Android systems [8, 17]. Another approach targets automatic source-to-source summarization by performing a user study on how humans summarize code-fragment examples [18]. We are also aware of other work on summarization generation for source code-related artifacts such as cross-cutting concerns [19], bug reports [20], and class diagrams [21].

## III. The Approach

Our approach takes a C++ system as input and for each method automatically generates a documentation summary that is placed as a comment block above the method. An example of the automatically generated documentation summary for one method is presented in Fig.1. The member function `calcWidthParm()` is in the class `BinAxsLinear` from the open-source system HippoDraw. The summary starts off with a short description, (e.g., the first three lines in Fig.1). This emphasizes the computed value along with the data members and parameters used in the computation. Following that is a list of calls made from this method along with the corresponding stereotypes of the calls.

```
/**
calcWidthParm is a property that
   returns a computed value new_width
   based on data member: m_range and parameter: number.
Calls to- high()     get
          low()      get
 */
double BinAxsLinear::calcWidthParm (int number) const {
  double new_width = (m_range.high() - m_range.low ())
                     / (static_cast < double > (number));
  return new_width;
}
```

Fig.1. An example of a *property* method and its generated documentation summary. The summary is inserted into the code as a method comment

To automatically construct a summarization, two issues must be addressed. The first is determining what information should be included in the summary. The next is to present this information efficiently. Previous studies on source-code summarization have investigated the former issue in depth and this information is used as a set of guidelines for building our automatic source-code summarization tool. First, the method summary should include at least a verb (i.e., describes the action performed by the method) and an object (i.e., describes on which the action is performed) [4, 7, 9]. Second, the name of the method is considered to be the most useful part of a method to use in the summary [16, 22]. Moreover, full identifiers are more preferable than split identifiers [4, 7] as it assists in mapping the summary to the source code. Calls within the method often produce side effects and the recommendation is to include them in the summary [16, 22]. Finally, local variables do not provide very useful information, and should not be included [22].

However, capturing the method responsibilities requires further analysis. The stereotype of a method can be used as a basis for selecting the content of the automated summary. In other words, the stereotype of the method is used to locate and determine the lines of code that reflect the main actions of a method. For example, the main action of a *property* method is to return a computed value whereas the main action for a *command* method is to modify more than one data member.

The second issue is how the information can be efficiently presented to support the goal of generating documentation summaries that are detailed enough to assist during maintenance tasks. To this end, summaries must be structured in a manner that is easily read and mapped to the source code. Therefore, the summary follows a predefined structure of four parts: 1) A short (mostly one or two line) description that expresses the main responsibilities of the method; 2) External objects used by the method; 3) Data members/parameters written; and 4) List of method calls and their stereotypes.

The documentation summaries are generated through two steps. Frist, the stereotype for all methods are identified and augmented to the source code. Next, our approach takes the stereotyped source code as input and uses the created database to generate method documentation summaries. The following section explains the main steps of our approach.

## IV. Method Stereotypes Identification

Method stereotypes are a concise abstraction of a method's role and responsibility in a class and system. A taxonomy of method stereotypes (see Table I) and techniques to automatically reverse-engineer stereotypes for existing methods was presented by Dragan et al. [11]. We refer the readers to that paper for complete details on computing method stereotypes; however, the main points are presented here.

TABLE I. TAXONOMY OF METHOD STEREOTYPES [15].

| Stereotype Category | Stereotype | Description |
|---|---|---|
| **Structural *Accessor*** | get | Returns a data member. |
| | predicate | Returns a Boolean value which is not a data member. |
| | property | Returns information about data members. |
| | void-accessor | Returns information through a parameter. |
| **Structural *Mutator*** | set | Sets a data member. |
| | command | Performs a complex change to the object's state (this). |
| | non-void-command | |
| **Creational** | factory | Creates and/or destroys objects. |
| **Collaborational** | collaborator | Works with objects (parameter, local variable, and return object). |
| | controller | Changes an external object's state (not *this*). |

The taxonomy was developed primarily for C++ but it can be easily applied to other OO programming languages. Based on this taxonomy, static-program analysis is used to determine the stereotype for each method in an existing system. The taxonomy is organized by the main role of a method while emphasizing the creational, structural, and collaborational aspects with respect to a class's design. Structural methods provide and support the structure of the class. For example, *accessors* read an object's state, while *mutators* change it. *Creational* methods create or destroy objects of the class. *Collaborational* methods characterize the communication between objects and how objects are controlled in the system.

Methods can be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have secondary stereotypes from the *collaborational*. For example, a *get collaborator* method returns a data member that is an object or uses an object as a parameter or a local variable.

A tool [11], *StereoCode*, was developed that analyzes and re-documents C++ source code with the stereotype information for each method. Re-documenting the source code was done using srcML [12]. Additionally we build a list of methods and their stereotypes. The list includes information about each class to speed up the stereotype extraction of calls during summary generation. For each class, data members and method names along with their stereotype are extracted. The list of data members is important in order to locate a method in the database. If a method is invoked on the data member (i.e., the data member is an object), then the type of the data member is used to identify the class owner of that method.

During summary generation, the stereotypes of calls are looked up, which provides additional information about the actions taken upon the call. Although the method name is an important way to indicate the role of a method, names can also be misleading [1]. For example, in the HippoDraw system the name of a call to the method `setRange()` indicates that the method modifies one variable/data member (`Range`). However, in this case the method actually modifies `Range` and two other data members, named m_num_bins and m_width. Including the stereotype of the call (in this case, *command*) alerts the developers of the possibility of changing other data members. Another example is a non-void method call. If the call is stereotyped as a *property* method, programmers can conclude that the main action of the call is to return a computed value. On the other hand, stereotyping the call as a *non-void-command* indicates that the call performs two actions: modifying multiple data members and returning a value. Therefore, including the stereotype of calls can reduce the need to examine the definition of these calls, which can be time-consuming, especially for calls to methods in different classes. Over a large system, this can improve the overall maintenance process. The heuristics and templates used to generate the automated summaries are explained in the following section.

## V. AUTOMATIC GENERATION OF SUMMARIES

The template for the four sections of the documentation summary is shown in Fig. 2. If a section is not applicable, it is omitted. For example, the automated summary of the *property* method in Fig.1 omits lines 2 and 3. The individual templates for each section of the summarization are explained below.

The first section of the documentation summary (line 1 in Fig. 2) is an English description that explains the main responsibility of the method in terms of the stereotype(s). The template for the short description is composed of two phrases. The first phrase is standardized for all stereotypes while the second phrase is customized for each specific stereotype. The first phrase of the template for the short description presents the method stereotype in the following form:

```
<method> is a <stereotype> [that collaborates with <obj>]
```

The second part in the above template is optional and is used if the secondary stereotype of the method is *collaborator* or *controller*. For example, the generated summary of the method `validate()` that works with the object `Range` gives:

```
validate is a void-accessor that collaborates with Range
```

We avoid long lists of objects. Therefore, if the method is collaborating with more than one object, then the number of objects is included in the short description and the list of names is moved to the second part of the summary (line 2 Fig. 2). The second phrase of the short description explains the main actions of the method. The particular template used depends on the stereotype of the method.

```
  /**
1 <Short Description>
2 Collaborates with- <obj_1>, ... <obj_m>
3 Data members or parameters modified- <name_1>, ... <name_p>
4 Calls to-  <call_1>        <stereotype_1>
                      …
           <call_n>        <stereotype_n>
  */
```

Fig. 2. The template of the four sections of the documentation summary.

***Templates for Accessors***. *Accessors* are const methods that do not change the object state. The main responsibility of an *accessor* is to return: 1) a data member (*get*), 2) information about data members (*property* and *predicate*), or 3) information through parameters (*void-accessor*). Therefore, data members or local variables used in the return expression or as reference parameters are included in the short description. To cover all the variations, we have developed 27 template phrases for the *accessors*' short descriptions. The main cases are now described. A *get* method returns a data member only and gives:

```
that returns one data member: <data-member>
```

A *property* returns information about data members. Thus, the return expression is the essential line that presents the main role of the method. The return expression can be a single variable, a computation, or a method call, each of which is handled differently. If the return expression is a call, then the call and its stereotype is included in the short summary as:

```
that delegates to <stereotype>: <call>
```

If the return expression is a single variable (Fig.1) or a value, the following template is used:

```
that returns a computed value <value>
```

If the return expression is a computation (e.g., return x + y), the above is used and "<value>" is omitted. Then, for the three cases explained above, data members, parameters, and calls used to compute the final value are listed using the "based on" template as given below:

```
based on data member(s): <data-mem1>, …, <data-memn>
and parameter(s): <parameter1>, …,  <parameterm>
and value(s) computed from <call1>, .., <callk>
```

Finally, if the method has more than one return expression, the list of the returned values is displayed and separated with "or". Typically, some logic controls which return statement is executed. So the phrase "depending on" is then used to specify the properties controlling the returned values as shown below:

```
returns <value1>, …, or <valuen>
depending on data member(s): <data-mem1>, …, <data-memn>
and parameter(s): <parameter1>, …,  <parameterm>
and value(s) computed from <call1>, .., <callk>
```

A *predicate* returns a Boolean value by an expression that includes a comparison to a data member. Therefore, the main responsibility of a *predicate* method is also located in its return expression. We apply *property*'s templates to *predicates*.

A *void-accessor* returns information through a parameter. Therefore, the key responsibility of a *void-accessor* method is to modify one or more parameters. To conclude that a parameter $p$ is modified by a method $M$, two conditions are evaluated. First, the parameter $p$ is passed as a non-const reference or as a pointer. Second, the parameter $p$ is modified directly using an assignment statement or indirectly using a method call within the method $M$. A parameter that is modified by a call is presented using the following template:

```
that modifies parameter: <parameter>
                                 via <stereotype>: <call>
```

This analysis is done statically on the method using srcML and the database of method stereotypes for the system. To determine whether a parameter is changed by a call, the stereotype of the call is used. The stereotype of calls within a method are extracted from the method/stereotype database explained in section IV. When a *mutator* is called on the parameter, or it is passed by reference to a *void-accessor*, we can conclude that the parameter is modified.

Consider the *void-accessor* method in Fig. 3. The method is from the class `XyPlotter` from the system HippoDraw and has three formal parameters: `mx`, `my`, and `picked`. As the parameter `picked` is by reference, it may be modified. The parameter `picked` is also a parameter of the call `fillPickedPoint()` which is a *void-accessor collaborator* method. Thus, it is determined that the parameter `picked` is modified by the call to `fillPickedPoint()`. Here, there is only one call so it will be included in the short summary, and the list of calls section is omitted.

```
/**
fillPickedPointFrom is a void-accessor that
   modifies parameter: picked
   via void-accessor collaborator: fillPickedPoint().
Data member read- m_plotter
*/
void XyPlotter::fillPickedPointFrom ( double mx, double my,
std::vector < double > & picked ) const {
  m_plotter -> fillPickedPoint ( mx, my, picked );
}
```

Fig. 3. An example of a void-accessor method and its generated summary.

If more than one parameter is modified by multiple calls, the number of modified parameters is included and the list is moved to the third section. Additionally, if more than five parameters are modified by assignment, the number is included the short description and the list is moved in the third section.

***Templates for Mutators****.* *Mutators* are designed to change the object state by modifying one data member (*set*) or multiple data members (*command* or *non-void-command*). Therefore, the data members changed within a method are included in the short description. In total, there are 21 templates for the short descriptions of *mutators*. The following describes the main cases. A *set* method modifies one data member only and we use the following template:

```
that modifies one data member: <data-member>
```

A *command* method executes a complex change of the object's state. The change involves more than one data member either directly or indirectly using another *mutator*. To identify whether a data member is modified by a call, the same analysis rules as described for *void-accessor* are used. In addition, the template of *command* is similar to the template of *void-accessor*. However, the modification of the data members are reported instead.

A *non-void-command* is a special form of a *command* method where the return type is not void or Boolean. This method has two responsibilities: changing data members and returning a value. Therefore, in addition to the modified data members, information about the returned value is also added to the short description. The modified data members are handled using *command* templates while the returned value information is expressed using *property* templates. If a data member is modified and returned, the following is used:

```
that returns a modified data member: <data-member>
```

***Templates for Creational Methods****.* A *factory* method is a *creational* method that returns an object created within the method. Typically, the created object is initialized through a constructor, copy constructor, *mutator*, or *void-accessor*. The summarization template for these is:

```
that returns an object: <object-name> of type <object>
```

If the object is modified by a call, this information also is included in the short description:

```
<object-name> is modified by <stereotype> : <call>
```

Summarization is restricted to the primary action of the method and omits details of the conditional logic. This decision was based on findings from prior work [16] where it was observed that when programmers write their own summary for a method, control statements (if and loops) receive lower attention compared to other parts of the method. Finally, if the approach fails to generate a short description for a method (usually when a method is just a list of calls), the numbers of method calls from each stereotype category (*accessors*, *mutator*s, *collaborational*, and *creational*) are included in the short description.

The second section of the documentation summary (line 2 in Fig. 2) includes a list of objects that the method collaborates with. The list helps to identify the external objects, which is particularly important for large methods.

This part of the summary is used to prevent the short description from being too long when more than one data member or parameter is modified via calls. Additionally, if the number of parameters or data members modified by assignments is greater than five, the list is included here.

The last section of the documentation summary of a method is a table that includes all method calls along with their corresponding stereotypes as shown in Fig. 2 line 4. Calls are ordered as they appear in the method. To reduce the complexity, the parameters of each call are removed [18]. Since the list is structured as a table, even a long list will not reduce the readability of the documentation summary.

The final documentation summary uses the template shown in Fig. 2 to group the four sections just described. A complete list of templates for each stereotype is provided online at (www.sdml.info/method_summarization).

## VI. Conclusion and Future Work

This paper proposes an automatic approach to generate documentation summaries for C++ methods. Using method stereotypes, a summary template was created for each specific method stereotype. Then, static analysis is used to extract the main components of the method. Finally, each method is re-documented with the generated documentation summaries. The approach is highly scalable and can be generalized to other OO programming languages. We believe that the summaries can support comprehension during maintenance.

We conducted an initial evaluation using undergraduate students as participants. The results indicate that the automated summaries accurately express what the method does and include necessary information. The results also imply that our approach to this problem, creating separate templates for each stereotype, is a promising and efficient solution. While the majority of the participants agreed about the automated summaries, some improvements are needed especially for *controller* and *collaborator* as they are somewhat complex and difficult to summarize accurately. We are currently extending this evaluation to assess the quality and effectiveness of the summaries for maintenance tasks.

## References

[1] Ko, A.J., Myers, B.A., Coblenz, M.J., and Aung, H.H. "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks", *IEEE TSE*, vol. 32, no. 12, 2006, pp. 971-987.

[2] LaToza, T.D., Venolia, G., DeLine, R. "Maintaining mental models: a study of developer work habits", ICSE, 2006, pp. 492-501.

[3] Fluri, B., Wursch, M., Gall, H.C. "Do code and comments co-evolve? on the relation between source code and comment changes", WCRE, 2007, pp. 70-79.

[4] Haiduc, S., Aponte, J., Moreno, L., Marcus, A. "On the Use of Automated Text Summarization Techniques for Summarizing Source Code", WCRE, 2010, pp. 35-44.

[5] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K. "Towards Automatically Generating Summary Comments for Java Methods", ASE, 2010, pp. 43-52.

[6] Sridhara, G., Pollock, L., Vijay-Shanker, K. "Automatically Detecting and Describing High Level Actions within Methods", ICSE, 2011, pp. 101-110.

[7] Eddy, B.P., Robinson, J.A., Kraft, N.A., Carver, J.C. "Evaluating Source code Summarization Techniques: Replication and Expansion", ICPC, 2013, pp. 13-22.

[8] Wong, E., Yang, J., Tan, L. "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation", ASE, 2013, pp. 577-582.

[9] McBurney, P., McMillan, C. "Automatic Documentation Generation via Source code Summarization of Method Context", ICPC, 2014, pp. 279-290.

[10] Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K. "Automatic generation of natural language summaries for Java classes", ICPC, 2013, pp. 23-32.

[11] Dragan, N., Collard, M.L., Maletic, J.I. "Reverse Engineering Method Stereotypes", ICSM, 2006, pp. 24 - 34.

[12 Collard, M.L., Decker, M.J., Maletic, J.I. "Lightweight Transformation and Fact Extraction with the srcML Toolkit", SCAM, 2011, pp. 173-184.

[13] Sridhara, G., Pollock, L., Vijay-Shanker, K. "Generating Parameter Comments and Integrating with Method Summaries", ICPC, 2011, pp. 71 - 80.

[14] Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K. "Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse", ICSE, 2009, pp. 232 - 242.

[15] Dragan, N., Collard, M.L., Maletic, J.I. "Automatic Identification of Class Stereotypes", ICSM, 2010, pp. 1 -10.

[16] Rodeghero, P., McMillan, C., McBurney, P.W., Bosch, N., and D'Mello, S. "Improving Automated Source code Summarization via an Eye-Tracking Study of Programmers", ICSE, 2014, pp. 460-471.

[17] Vassallo, C., Panichella, S., Di Penta, M., Canfora, G. "CODES: mining source code Descriptions from developers discussions", ICPC, 2014, pp. 106-109.

[18] Ying, A.T., Robillard, M. "Selection and Presentation Practices for Code Example Summarization", SIGSOFT 2014, pp. 460-471.

[19] Rastkar, S., Murphy, G.C., Bradley, A.W.J. "Generating Natural Language Summaries for Crosscutting Source Code Concerns", ICSM, 2011, pp. 103-112.

[20] Rastkar, S., Murphy, G.C., Murray, G. "Summarizing Software Artifacts: A Case Study of Bug Reports", ICSE, 2010, pp. 505-514.

[21] Burden, H., Heldal, R. "Natural Language Generation from Class Diagrams", MoDeVVa, 2011.

[22] Moreno, L., Aponte, J. "On the Analysis of Human and Automatic Summaries of Source Code", *CLEI Electronic Journal*, vol. 15, no. 2, 2012