Automatic Generation of Natural Language Summaries for Java Classes

Laura Moreno[1], Jairo Aponte[2], Giriprasad Sridhara[3], Andrian Marcus[1], Lori Pollock[4], K. Vijay-Shanker[4]

[1]Wayne State University
Detroit, MI, USA
{lmorenoc, amarcus}@wayne.edu

[2]Universidad Nacional de Colombia
Bogotá, Colombia
jhapontem@unal.edu.co

[3]IBM Research India
Bangalore, India
gisridha@in.ibm.com

[4]University of Delaware
Newark, DE, USA
{pollock, vijay}@cis.udel.edu

***Abstract*—Most software engineering tasks require developers to understand parts of the source code. When faced with unfamiliar code, developers often rely on (internal or external) documentation to gain an overall understanding of the code and determine whether it is relevant for the current task. Unfortunately, the documentation is often absent or outdated.**

This paper presents a technique to automatically generate human readable summaries for Java classes, assuming no documentation exists. The summaries allow developers to understand the main goal and structure of the class. The focus of the summaries is on the content and responsibilities of the classes, rather than their relationships with other classes. The summarization tool determines the class and method stereotypes and uses them, in conjunction with heuristics, to select the information to be included in the summaries. Then it generates the summaries using existing lexicalization tools.

A group of programmers judged a set of generated summaries for Java classes and determined that they are readable and understandable, they do not include extraneous information, and, in most cases, they are not missing essential information.

***Index Terms*—Source code summarization, program comprehension, documentation generation.**

I. Introduction

Existing studies [1] revealed that developers often spend more time searching, browsing, and reading the code than editing it. Searching, browsing, and reading are essential activities needed to understand software, which in turn is needed for everyday software maintenance tasks. While browsing the source code, developers sometimes just glance at it to get a quick understanding and sometimes spend more time reading it in detail [1-3]. Skimming the code is performed in order to determine whether a specific part of it is relevant to the task at hand or not. When the code has good leading comments, developers can acquire a quick understanding of the code artifact. Unfortunately, more often than not, good comments are missing or outdated, and therefore, developers must spend much more time reading the code in detail, in order to gain even a superficial understanding.

One approach to overcome this problem is to automatically generate descriptive comments directly from the source code. While successfully applied for Java methods [4], generating comments for more complex code artifacts, e.g., classes, is significantly more difficult [5, 6]. Our focus here is on classes, as they are the primary decomposition unit in Object-Oriented (OO) programming languages, such as Java. In addition, the OO paradigm supports reasoning at the object level and, consequently, code understanding and (re)use at the class level.

Unfortunately, we cannot use existing comment generation tools for methods (e.g., [4]) and simply merge them to create a class summary. The reasons vary: (i) classes bundle together more than just methods – they also include data that the methods presumably operate on; (ii) adding together all method descriptions would result in very large summaries, which defeats their goal; (iii) not all methods are the same – some may be relevant to describe the behavior of the class instances, while some may not.

We propose in premiere a technique to automatically generate structured natural-language descriptions for Java classes, independent of their context and assuming that no documentation exists (i.e., if it exists, the comments are not currently used). The system takes a Java project as input, and for each class, it outputs a natural-language summary. The goal of the generated summaries is to support the quick understanding of a class by describing its intent and leaving aside its context and any algorithmic details. In this sense, the summaries are *indicative* (i.e., provide a brief description of the class content), *abstractive* (i.e., include information that is not explicit in the class), and *generic* (i.e., attempt to cover only the important information of the class).

The intended audience is any developer, especially a novice, who is unfamiliar with the code and needs to quickly get the gist of the class to decide whether to peruse the source code or not. For example, the developer may be deciding whether to (re)use a class *X* and wondering whether it would serve her needs; or, while reading the code of another class, she encounters an attribute of type *X* and wonders what it means. Developers sometimes write comments that describe the main responsibility of a class, to help other developers, regardless of their task. Our automatic summaries have the same goal. Although different maintenance tasks require different kinds of information from classes, our approach can serve as an initial step in the generation of specific-purpose summaries, which is outside the scope of this paper.

Our conjecture is that the type of methods and their distribution in a class is not accidental and denotes some design intent, which reflects the main goal of the class. Thus, our summarization technique first determines the stereotypes of the class [7] and each one of its methods [8]. The stereotype information is used in conjunction with predefined heuristics, to select the information that will be included in the summary.

The summary of the class is then generated by combining the selected information following a set of predefined rules, and using techniques developed to generate natural language phrases for variables and program statements [4]. The technique is completely automated and very fast.

As mentioned, the summarization technique assumes that no comments are present (i.e., a worse-case scenario). The generated summaries include identifiers associated with data attributes and methods. Since no domain information is used (e.g., domain vocabularies or ontologies), the quality of the summaries depends on the quality of the identifiers. The summaries focus majorly on class responsibilities rather than on class relationships. Clearly, to fully understand a class, its context should not be disregarded. Future work will investigate augmenting the summaries with class relationship information.

The paper presents the details of our technique and its implementation, concluding with an empirical evaluation where we asked 22 programmers to evaluate three aspects of summaries generated for 40 Java classes from two systems. They found that the summaries are expressive (i.e., readable and understandable) and concise (i.e., do not contain extraneous information), and that their content is adequate (i.e., important information is not missing) in most cases.

II. Automatic Summarization of Classes

While automated techniques to summarize source code blocks within methods [9] and entire methods [4] have been proposed, they cannot be used to generate class summaries, as discussed in Section I. Class summaries (as any other code summaries, for that matter) should be *concise* (i.e., include only necessary information), *adequate* (i.e., include all necessary information), *readable*, and *understandable*. In order for our technique to generate summaries with the above characteristics, we need to establish:

- What information to include in the summaries?
- How much information to include in the summaries?
- How to generate and present the summaries?

First, we identified certain class elements that help to describe the intent of a class, such as, the names of its parent classes, interfaces, and inner classes (if any); and its attributes and methods. As mentioned above, we decided not to include information on relationships between classes.

We divided the selected elements into two sets. The first set consists of elements that are included in the summaries directly (i.e., do not require any analysis): the names of interfaces, superclass, and inner classes. Class attributes and methods require further analysis. For example, many private attributes are used for algorithmic reasons, whereas others may be exposed by public methods to the users of the class. Likewise, some methods are used simply to access attributes, whereas others are used to send more complex messages between objects. We distinguish between the different types of attributes and methods, in order to decide which ones should be reflected in the summaries.

Classes in OO systems have *generic* (i.e., domain-independent) responsibilities and *specific* responsibilities (i.e., domain-dependent). *Class stereotypes* are high level abstractions that describe the role or responsibility of classes in a system [7]. We decided to extract and use class stereotype information to reflect the generic responsibility of a class. For example, an *Entity* class encapsulates data and behavior, and it is usually the keeper of the data model and business logic. This is domain independent information, which can be automatically extracted from the source code. The domain specific information is given by the identifiers of methods and fields. Note that we do not employ any domain analysis (e.g., word relations) or domain artifacts (e.g., domain vocabulary or ontology) to determine the quality of the identifiers, hence the quality of the summaries is implicitly dependent on the identifiers we include.

Class stereotypes determine which kind of methods (i.e., based on their stereotypes as well) to include in the summaries. We also developed a set of heuristics, based on the access level (e.g., public, private, etc.) to the data members and methods (see Section II.B for details).

Once the information is selected, the summaries are constructed using techniques developed to generate natural language phrases for variables and program statements [4] (see Section II.C). The summary consists of four parts:

1. A general description based on the interfaces, superclass and/or the stereotypes of the class.
2. The description of the structure based on its stereotype.
3. The description of the behavior based on enumerating its most relevant methods, grouped in blocks.
4. A list of the inner classes, if they exist.

The remainder of this section describes in details the main steps in the summary generation: *stereotype identification*, *content selection*, and *text generation*.

A. Stereotype Identification

We identify the stereotype of a class by adapting the rules proposed in [7, 8]. These rules consider the distribution of the methods and their stereotypes in the class.

1) Method Stereotypes. Method stereotypes describe the responsibility of methods within a class. For example, a method that returns an attribute directly is commonly known as a *get* method, whereas a method that modifies one attribute is called a *set* method. Recent work [8, 10] defined 15 stereotypes for C++ methods.

Method stereotypes are classified as: *accessors*, if the method returns information about the object's state directly, or through the parameters of the method; *mutators*, when the method changes the object's state; *creational methods*, if the method creates or destroys objects; *collaborational methods*, when the method defines the communication between objects or how the objects are controlled in the system; and *degenerate methods*, in any other case. Each category includes several specific stereotypes. We implemented *JStereoCode*, an Eclipse plug-in for automatically identifying Java code stereotypes [11]. The tool determines the method stereotypes based on the AST and some additional information, such as, read and modified local fields, method calls, and external and internal classes used. Some of these elements are used in the text generation step to enrich the summary of the class.

2) Class Stereotypes. Class stereotypes are categories that represent the intent of classes in the system's design. Previous work [7] defined a list of 13 class stereotypes (see Table I). The stereotypes are stated as rules based on the distribution of the class's method stereotypes. For example, to determine whether a class is *Boundary*, it has to contain more *collaborators* than non-collaborator methods, some *factory* methods, and a low number of *controller* methods. In quantitative terms, these conditions are translated to:

$$|collaborator| > |methods| - |collaborator| \text{ and}$$

$$|factory| < \tfrac{1}{2}|methods| \text{ and } |controller| < \tfrac{1}{3}|methods|$$

where $|methods|$ is the total number of methods, and $|factory|$, $|controller|$, and $|collaborator|$ represent the number of *factory*, *controller*, and *collaborator* methods in the class, respectively.

We modified several of the rules used to automatically identify the stereotypes in order to eliminate the overlap between some of the rules, which allowed classes to be classified as having too many stereotypes. We also relaxed some of the rules, which resulted in no classes getting certain classifications. We further extended the taxonomy by adding the stereotype *Pool*, which includes classes declaring several static final attributes and few or no methods.

The resulting class stereotypes are presented in Table I. It is important to highlight that a *Boundary* class can have a secondary stereotype if its main purpose is to provide access to its attributes or allowing their modification. Consequently, *JStereoCode* assigns each class to one of 16 categories, including the 13 described in Table I, plus *Boundary+Data Provider*, *Boundary+Commander*, and the *no-stereotype* category (i.e., classes that do not match the conditions of any stereotype). We tested *JStereoCode* on many Java systems and very few classes were categorized in the *no-stereotype* group.

B. Heuristics for Content Selection

Once the class stereotypes are determined, we need to identify which methods are to be included in the summary. These methods are determined through a filtering process, which starts with a set containing all the methods in the target class, except the ones overriding `Object`'s methods. Two filters are applied to this set:

1) Stereotype-Based Filter. The first filter removes the methods whose stereotypes are not relevant to the class stereotype according to its definition. In consequence, we apply one heuristic for each class stereotype. For example, *accessors* are the main methods in a *Data Provider* class, so every *non-accessor* method (i.e., every method that is not *accessor*) is filtered out when processing this kind of classes. For classes with two stereotypes we combine the two heuristics such that we remove the methods that are not relevant to *any* (not both) of the definitions. As an illustration, every method that is not *accessor* or *collaborator* is removed when processing *Boundary+Data Provider* classes. See [8, 9, 12] for more details in the stereotype definitions.

2) Access-Level Filter. The second filter is based on the access level permitted by the modifiers of the methods. Java provides four levels of access to method members (relevant in our context): *private*, *package-private*, *protected*, and *public*. Since we focus primarily on the most visible (i.e., system level) responsibility of a class, we remove private, package-protected, and protected methods (one category at a time). We remove these methods in order, from the least visible to the more visible, observing the stop rules defined below.

The filtering process ends when one of the following situations occurs: (i) the set has three or less methods; (ii) the set has been reduced to 50% or less than its initial size; or (iii) all the filters have been applied (occurs mostly for *Large Classes*).

TABLE I. CLASS STEREOTYPE TAXONOMY, ADAPTED FROM [7]

Stereotype	Description
Entity	Encapsulates data and behavior. Keeper of data model and business logic.
Minimal entity	Trivial Entity that consists entirely of accessor and mutator methods.
Data provider	Entity that consists mostly of accessor methods.
Commander	Entity that consists mostly of mutator methods
Boundary	Communicator that has a large percentage of collaboration methods, a low percentage of controller, and not many factory methods.
Factory	Consists mostly of factory methods.
Controller	Controls external objects - the majority of its methods are controllers and factories.
Pure controller	Consists entirely of controller and factory methods.
Large class	Combines multiple roles, i.e., it consists of accessors, mutators, collaborational, and factory methods.
Lazy class	Its functionality cannot be easily determined. It consists mostly of incidental, and get or set methods.
Degenerate	Very trivial class that does very little - it consists mostly of empty, and get or set methods.
Data class	Degenerate behavior - it has only get and set methods.
Pool	Consists mostly of class constants and a few or no methods.

C. Text Generation

The next challenge is generating a readable text description that expresses the relevant information of the class resulting from the content selection above. Based on Sridhara's work for method summary comment generation [4], we defined text templates for generating each of the four parts of class summaries (described at the beginning of Section II).

1) General Description. The first sentence of the summary provides a generic idea of the type of objects represented by the class. This is achieved by using the names of the super classes and interfaces of classes (if any), as qualifiers of the represented object. For example:

Class declaration:	`public class AudioFile extends File`
Generated text:	A file extension for audio files.

If both a superclass and interfaces exist, then they are included in the description using the template:

A <interface$_1$>,…,<interface$_i$> implementation, and <superclass> extension for <represented object>.

When classes do not extend or implement any type, the stereotype of the class is used as the qualifier. For example:

Class declaration:	`public class CdRipper`
Stereotype:	Boundary
Generated text:	A boundary class for cd rippers.

In the worst case, the general description only mentions the object represented by the class. The text generation component also considers whether the target class is an abstract declaration. For example:

Class declaration:	`abstract class Context`
Generated text:	An abstract class for contexts.

Considering the possible variations in class declarations and the class stereotype taxonomy, we defined 22 different templates to provide the general description.

2) Stereotype Description. The generic responsibility of the class is described using the stereotype definitions, presented in Table I. When it is possible, the definitions are enriched with specific information (i.e., some domain information), such as, the represented object, the classes used within the target class, or the existence of certain kind of methods. For example, when the class to summarize is a *Data Provider*, the generated text follows the template:

```
This entity class consists mostly of accessors to
the <represented object>'s state.
```

As mentioned before, the focus of the summaries is not on class relationships. The only (partial) exception is for *Boundary* classes, where it is important to mention (at least some of) the classes it communicates with. These classes can be selected in several ways, but we decided to include the most frequently used ones within the class. For example, the stereotype description for a *Boundary+Commander* class is:

```
This boundary class communicates with
<class1>, …, <class3>, and other <remaining number
of classes used> classes, and consists mostly of
mutators to the <represented object>'s state.
```

Some of the class stereotypes do not depend on the presence of all the method stereotypes (see the description presented in Table I). For example, a *Data Class* can consist entirely of get methods. In such a case, the fragment that refers to the existence of set methods is omitted from the description:

```
This data class consists only of getter methods.
```

When generating this part of the summary, we must consider whether the first part of the summary already contains the stereotype name or the word "class". If this is the case, then the text generation component uses alternative starting words in order to reduce redundancy and gain readability. For example:

Class declaration:	`Public class CdRipper`
Stereotype:	Entity
Generated text:	It includes accessors and mutators to the cd ripper's state, and some business logic.

Taking into account the class stereotype taxonomy, the stereotypes of the methods in the class, and the content of the first part of the summary, we defined 40 different templates for the second part of the summaries.

3) Behavior Description. For describing the behavior provided by the class, we use the relevant methods selected when applying the filtering process described in Section II.B. Note that we could simply enumerate all of those methods. However, this would reduce the readability of the summary, especially in cases when a class has methods of many different stereotypes.

Consequently, we divide the behavior description into three blocks. The first two blocks represent the *accessor* and *mutator* methods, respectively, while the third block represents the rest of methods. Then, to generate the behavior description, each relevant method is assigned to a block according to its stereotype. If one of the blocks is empty, it is ignored when generating the description. In general, the resulting text follows the template:

```
It provides access to:
 -   <phrase for accessor method1>;
 -   …; and
 -   <phrase for accessor methodx>.
It allows managing:
 -   <phrase for mutator method1>;
 -   …; and
 -   <phrase for mutator methody>.
It also allows:
 -   <phrase for method1>;
 -   …; and
 -   <phrase for methodz>.
```

To create a readable description of the behavior, we need to express the methods' action in natural-language form. We assume that the method signature provides enough information to describe the general functionality of a method in a readable form. This part of the summary is especially sensitive to the identifier quality. We used existing phrase generation tools [4] to obtain natural-language fragments from method signatures.

a) Lexicalization of Fields and Phrase Generation for Methods. The lexicalization of fields follows the same process described in [4] to lexicalize variables. The tool identifies nouns and adjectives in the name of the field and its type, and then it forms an English noun phrase that represents the field. For example, the field `Document current` is transformed into "current document." Also, the field `AudioFile currentFile` is transformed into "current audio file." There is no redundancy in this fragment, even when the word "file" is repeated in the declaration.

We adapted existing phrase generation templates for different programming constructs [4] and used them to generate phrases for method signatures. In these techniques, linguistic aspects of a method (e.g., its action, theme and secondary arguments) are identified [12, 13]. For example, the generated phrase for the method `void add(Item i)` is "add item". The parameter "item" is included as the theme in the text.

b) Final Lexicalization Details. We perform final adjustments on the phrases generated for each method. These adaptations vary from one block to the next. In the case of *accessor* methods (first block), the fragment to be added to the description depends on the stereotype of the method. If the method is a *get*, the fragment to add is the lexicalized form of the field returned by the method. For example:

Method signature:	`List getAudioFiles()`
Stereotype:	Get
Field accessed:	`List audioFiles`

Generated text:	It provides access to: - audio files list.

Otherwise, we remove the verb and secondary arguments from the phrase generated for the method to get the property that is being accessed; then, it is concatenated to the lexicalized form of the field that provides such property. For example:

Method signature:	`int getTrackNumber()`
Stereotype:	Property
Field accessed:	`Tag tag`
Generated text:	It provides access to: - Track number from tag.

On the other hand, the only modification to the *mutator* methods is the removal of the verb from the phrase generated from its signature. In that way, we add only the fragment that indicates the property which is being modified by the method. For example:

Method signature:	`void setContext(Context c)`
Stereotype	Set
Generated text:	It allows managing: - context.

One exception to this rule occurs when the method name consists of one verb only. Since the signature of the method does not provide the properties that are being modified, we move such methods in the third block and use the name of the class as the theme in the generated phrase. The final adjustment in this block is the transformation of the action of the methods into its gerund form. For example:

Class declaration:	`public class CdRipper`
Method signature:	`void stop()`
Stereotype	Command (mutator method)
Generated text:	It also allows: - stopping cd ripper.
Class:	`public class RepositoryHandler…`
Method signature:	`void refreshRepository()`
Stereotype	Collaborator
Generated text:	It also allows: - refreshing repository.

Note that we do not use the fields that are being modified by the *mutator* methods in the summary. After analyzing several methods in different systems, we found that modifications of the fields are better reflected by the signature of the methods.

4) Inner Classes Enumeration

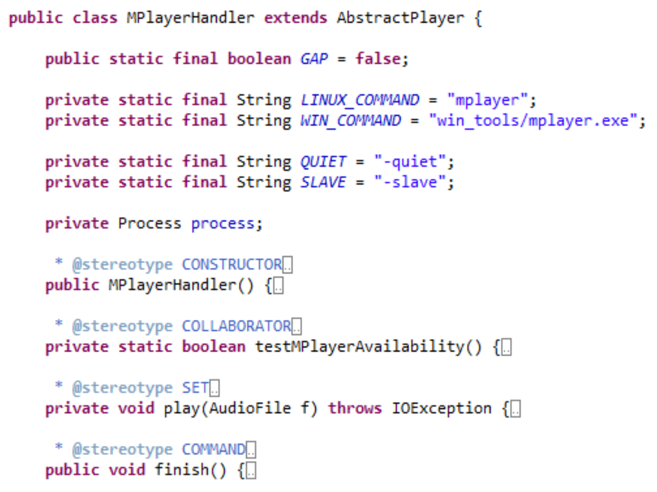
The last part of the summary is optional. It is only used when the class declares inner classes. In such cases, the following template is used:

`It declares the helper classes <inner class`$_1$`>, …, and <inner class`$_n$`>.`

The final summary is created by concatenating the four parts described above. Fig. 1. shows a complete summary generated for the `MPlayerHandler` class from aTunes, which consists of six fields and 23 methods. Nine of these methods are classified as *mutators* and one as *accessor*. The rest of them are spread in the creational, collaborational, and degenerate categories. According to the stereotype identification rules, this class is a *Commander*.

```
An AbstractPlayer extension for m player
handlers. This entity class consists mostly
of mutators to the m player handler's state.

It allows managing:
 - mute;
 - volume; and
 - next with no gap.
It also allows:
 - finishing m player handler;
 - handling next;
 - playing audio file f;
 - stopping m player handler;
 - playing m player handler; and
 - handling previous.
```



```
public class MPlayerHandler extends AbstractPlayer {

    public static final boolean GAP = false;

    private static final String LINUX_COMMAND = "mplayer";
    private static final String WIN_COMMAND = "win_tools/mplayer.exe";

    private static final String QUIET = "-quiet";
    private static final String SLAVE = "-slave";

    private Process process;

     * @stereotype CONSTRUCTOR
    public MPlayerHandler() {

     * @stereotype COLLABORATOR
    private static boolean testMPlayerAvailability() {

     * @stereotype SET
    private void play(AudioFile f) throws IOException {

     * @stereotype COMMAND
    public void finish() {
```

Fig. 1. Fragment of the class `MPlayerHandler` from the aTunes system and its automatically generated summary

III. Evaluation

The goal of the automatic summary of a class is to provide developers with a quick overview of its main responsibility, which can be easily read. Accordingly, we performed a study involving potential users, in a manner similar to previous work [4], in order to evaluate the following properties of the generated summaries:

- *Content adequacy*: Is the important information about the class reflected in the summary?
- *Conciseness*: Is there extraneous information included in the summary?
- *Expressiveness*: How readable and understandable is the summary?

For this study we asked 22 programmers to judge the content adequacy, conciseness, and expressiveness of automatically generated summaries for 40 Java classes.

A. Subjects and Objects of the Study

The study included 22 graduate students in computer science: 11 from the University of Delaware, 5 from Wayne State University, and 6 from Universidad Nacional de Colombia. We surveyed their programming knowledge and background. All of them reported good or very good knowledge of Java programming, and some of them have industrial experience as Java developers.

We used two open source Java systems in the study, namely aTunes 1.6.0 and ArgoUML 0.22. aTunes is an audio player, consisting of 218 classes and 1,852 methods. ArgoUML is an UML modeling tool with 1,548 classes and 10,341 methods.

Since stereotypes are the foundation of our summarization technique, for both systems we selected one class representing each stereotype and two classes for the most frequent stereotypes in these two systems. The five most frequent stereotypes were: *Entity*, *Boundary*, *Boundary+Data Provider*, *Boundary+Commander*, and *Controller*. Moreover, we omitted *Pure Controller* classes from the study, since they are almost nonexistent in both systems.

We selected 20 classes per system, by grouping the classes according to their stereotype, and randomly selecting two for the five most frequent stereotypes and one for each of the other 10. While we wanted to ensure stereotype coverage, we also wanted to make sure each class summary is evaluated by at least three participants. In total, 40 classes were selected.

B. Experimental Design and Procedure

An expert with knowledge of a particular Java class (ideally its main developer) would give the most trustworthy evaluation of a summary of that class. Since we did not have access to the developers of the object systems, we ensured that the participants understood the summarized classes as much as possible. Therefore, we asked participants to familiarize themselves with the major functionality and structure of both systems by reading brief descriptions and, if needed, by executing the systems. After that, they were asked to (i) study and understand the entire class, (ii) write their own description for the class, (iii) read the provided automatic summary, and finally, (iv) evaluate the summary by answering the three questions presented in Table II. We asked the participants to write their own summary (with no predefined format or content requirement) in order for us to assess later whether they really understood the class or not. Our assumption is that if a participant did not understand a class properly, their judgment of the summary may be useless. When evaluating the summaries, if they selected the third choice for content adequacy, or the second and third for conciseness (see Table II), then they were asked to write down what was missing or extraneous in the summary.

The assignation of the 40 selected classes and their summaries to the participants was designed in such a way that:

- Each summary is reviewed by three participants;
- Each participant evaluates an equal number of classes from aTunes and from ArgoUML, and
- None of the participants receives summaries of classes with the same stereotype from the same system.

Thus, each participant was provided with six classes, three per system, and their corresponding automatic summaries. In order to achieve our goals, 20 subjects would have been enough (three judgments per summary and six summaries per participant). The distribution of classes was done before inviting the participants. We invited 25 participants (who were contacted individually) assuming that some will not be able to complete the study. The extra five evaluations were replicating five of the first 20 evaluations. Three invited participants did not complete the evaluation, so we collected data from 22. Hence, two of the summaries got only two judgments, 12 summaries received four judgments, one summary got five judgments, and the rest (25) of the summaries got three judgments. The subjects judged the classes in different order, and both systems were intermingled (i.e., some subject started with aTunes and some with ArgoUML).

We used a web page and an online survey tool to perform the study. This allowed us to (i) provide participants with the necessary resources and instructions, (ii) control the steps that they had to perform, and (iii) collect their answers. Additionally, they used an IDE to import the source code of both systems and study the Java classes assigned to them. The participants carried out the evaluation independently, i.e., at the time and space of their choice, without our supervision, and using the IDE of their choice. They could also stop in the middle of the evaluation and resume at any later time. We asked them to provide the time they spent on studying each class to ensure they did spend enough time on each class, and it did not take them too much time to complete the evaluation. We estimated that it would take on average between 90-120 minutes to complete the evaluation (15-20 minutes per class). On average, the participants accomplished this part of their task in 10 minutes, for either system. The two classes that took longest were one *Commander* and one *Boundary* class from ArgoUML, yet still within our expectations (20 minutes on average). One of those classes represented a functionality not fully implemented in this version of the system (org.argouml.cognitive.ToDoList). In the case of simple classes such as *Minimal Entities*, the average was below five minutes. We concluded that the participants did not get too tired through the study and they spent an adequate amount of time understanding the code.

TABLE II. QUESTIONS AND POSSIBLE ANSWERS USED IN THE EVALUATION STUDY

Criteria	Question	Possible answers
Content adequacy	Considering only the content of the description and not the way it is presented, do you think that the description?	1. Is not missing any information 2. Is missing some information but the missing information is not necessary to understand the class 3. Is missing some very important information that can hinder the understanding of the class
Conciseness	Considering only the content of the description and not the way it is presented, do you think that the description?	1. Has no unnecessary information 2. Has some unnecessary information 3. Has a lot of unnecessary information
Expressiveness	Considering only the way the description is presented and not its content, do you think that the description?	1. Is easy to read and understand 2. Is somewhat readable and understandable 3. Is hard to read and understand

C. Results and Discussion

In total, we collected 132 evaluations of the summaries of 40 classes. Before analyzing the results, two of the authors assessed separately each summary written by the participants. The authors spent two days studying and understanding the 40 classes. Even before the study, they were quite familiar with both systems. Each participant summary was assessed on a two-level scale: it was considered *good* when it captured the intent of the class, or *bad*, otherwise. Conciseness or clarity of the summaries were not considered at this stage. Clearly these evaluations are subjective; hence we ensured that the two authors agreed on their evaluations. The assessments given by the authors were compared to confirm the high or low quality of the descriptions. In most cases the authors agreed on the results. Where some disagreement existed, it was carefully discussed, until they reached an agreement.

In the end, based on this analysis, 24 out of the 132 participant descriptions indicated a poor understanding of the class. One of the classes was particularly difficult to understand by the participants (i.e., all of its evaluator descriptions were misleading or incorrect). This is a *Factory* class from ArgoUML (i.e., org.argouml.ui.explorer.rules.Go TransitionToGuard). The rest of the *bad* descriptions were spread uniformly across all the classes, stereotypes, and systems, i.e., they did not point at any specific stereotype, system, or participant. While analyzing the results, we duplicated each analysis with and without the responses corresponding to the 24 *bad* participant summaries. We expected that the participants who misunderstand a class would be prone to choose the first answer in each category, in order to avoid justifying their choice. However, these 24 cases did not significantly affect the results (see Table III for details).

Table III reports the percentage of automatic descriptions that were rated in each category for content adequacy, conciseness, and expressiveness. The numbers in parenthesis reflect the results after removing the evaluations corresponding to the 24 *bad* user summaries. Overall, the results are promising as for each property of the automated summaries the largest set of answers were the most positive choice. According to these numbers we can say that the automatic descriptions are mostly understandable and readable, have little unnecessary information, but sometimes miss important information. Note that those cases (the 24 ones) where human summaries indicated deficient understanding of the classes do not influence much the overall results of the study.

We also looked at the differences in the ratings given by evaluators between the two software systems. The two systems are quite different from each other – chosen on purpose. aTunes is smaller, its domain is much simpler (e.g., there are classes like Artist, Album, Song, AudioFile, etc.), and most of its classes have very descriptive identifiers. The results indicate that more of the summaries from ArgoUML miss important information than in aTunes (38% vs. 24%). On the other hand, 3% of the ArgoUML summaries are not concise vs. 5% of the aTunes summaries. Finally, 6% of the ArgoUML summaries are hard to read and understand vs. 2% of the aTunes summaries. These numbers include the answers corresponding to the 24 *bad* user summaries.

We present a more detailed analysis of the results for each of the three evaluated summary properties. The qualitative analysis is done using the free form text the participants provided when answering with the second or third choice, while evaluating the content adequacy and the conciseness.

1) Content Adequacy. We consider this property as the most important, and at the same time, the hardest to ensure automatically. While it is the characteristic with lowest results, in 69% of the cases the evaluators considered that the automatic summaries miss little or no information relevant to the class. We analyzed the answer for each class stereotype to see whether some stereotypes prove to be more difficult to summarize than others. Table IV summarizes these results, and we discuss them in more detail.

The stereotypes that consistently got the best answer (i.e., the summaries include all the necessary information) were the following: *Minimal Entity*, *Commander*, *Data Class*, and *Large Class* (100% of the cases); *Boundary* (82% of the cases); and *Entity* (71% of the cases). We did not expect that the summaries of *Large Classes* would obtain such good scores for content adequacy. Such classes are usually complex, and combine multiple roles, having methods with various different stereotypes.

We analyzed in detail the cases where the summary did not include important information. We found that 7 evaluators out of 8 considered that the descriptions for classes with *Lazy Class* stereotype missed important information. This was expected since this stereotype represents classes whose functionality cannot be easily determined (i.e., they consist mostly of incidental and get or set methods).

Another case where automatic descriptions miss important information corresponds to classes with *Degenerate* stereotype. This stereotype is assigned to classes with little functionality, i.e., those consisting mostly of empty methods, getters and setters. *Factory* classes also proved to be difficult to describe, especially in the case of the ArgoUML system.

TABLE III. DISTRIBUTION OF THE PARTICIPANTS' RESPONSES
(the numbers in parenthesis reflect the answers after removing the 24 corresponding to the bad user summaries)

Content Adequacy		**Conciseness**		**Expressiveness**	
Response category	***Percentage of Ratings***	***Response category***	***Percentage of Ratings***	***Response category***	***Percentage of Ratings***
Not missing any information	43% (45%)	Has no unnecessary information	61% (59%)	Is easy to read and understand	67% (68%)
Missing some information	26% (26%)	Has some unnecessary information	36% (37%)	Is somewhat readable and understandable	30% (31%)
Missing some very important information	31% (29%)	Has a lot of unnecessary information	4% (4%)	Is hard to read and understand	4% (1%)

TABLE IV. DISTRIBUTION OF THE PARTICIPANTS' RESPONSES PER EACH STEREOTYPE (BASED ON ALL 132 ANSWERS)

Stereotype	**Content adequacy**			**Conciseness**			**Expressiveness**		
	No info missed	***Some info missed***	***Important info missed***	***No unnecessary info***	***Some unnecessary info***	***Unnecessary info***	***Easy to read***	***Somewhat readable***	***Hard to read***
Degenerate	17%	33%	50%	83%	17%	0%	50%	50%	0%
Minimal Entity	83%	17%	0%	83%	17%	0%	100%	0%	0%
Data Provider	43%	14%	43%	100%	0%	0%	100%	0%	0%
Commander	67%	33%	0%	50%	50%	0%	17%	67%	17%
Boundary	64%	18%	18%	27%	47%	27%	71%	14%	14%
Boundary+Data Provider	31%	38%	31%	38%	46%	15%	69%	31%	0%
Boundary+ Commander	46%	15%	38%	54%	46%	0%	85%	15%	0%
Factory	14%	43%	43%	57%	43%	0%	29%	43%	29%
Controller	21%	50%	29%	57%	36%	7%	57%	29%	14%
Large Class	57%	43%	0%	86%	14%	0%	29%	71%	0%
Lazy Class	0%	13%	88%	50%	50%	0%	63%	38%	0%
Entity	64%	7%	29%	71%	29%	0%	71%	29%	0%
Data Class	83%	17%	0%	83%	17%	0%	100%	0%	0%
Pool	29%	29%	43%	86%	14%	0%	100%	0%	0%
No Stereotype	50%	17%	33%	17%	67%	17%	17%	83%	0%

In the study, we asked participants to indicate the missing information for summaries rated as low content adequacy. We divided this information into three categories. The first group consists of relevant information reported as missed by several participants. Unmentioned methods and attributes fall in this category. The omitted methods are mainly caused by the filtering process: the stereotypes of the methods considered relevant by the developer are not necessarily relevant to the stereotype of the class (e.g., collaborational methods in *Data Provider* or *Degenerate* classes). This is an area where we can improve our technique, but more studies may be needed to lead us to ways to adjust our filtering process. Another cause for this situation is the loss of the method's intent when generating text from its signature, in some cases. For example, while constructing the second block of the behavior description, the action of the *mutator* methods is sometimes removed. In the case of attributes reported as missing, we found that they usually correspond to attributes modified in *mutator* methods, when the name of the method did not reflect them. We plan to adjust our tools to include such attributes.

The second category consists of missed relevant information, reported only by one participant, yet we considered it important to analyze. We found that these reports provide important details that can help to improve stereotype descriptions. For example, in the summary of a *Factory* class we should include the classes with which it interacts. In the case of *Pool* classes, it is important to provide the data type of the constants declared by the class. The parameters of specific methods were also reported as missing information. We found that participants referred mainly to the parameters of static methods, which we currently ignore.

The missing information in the last group refers to elements that are beyond the scope of the summaries or indicate poor understanding of the evaluation task. For instance, evaluators who were unfamiliar with some concepts in the code reported their definition as missing. The same situation occurred with the usage of the classes, although we specifically decided not to address this issue at the current stage. The generated summaries do not consider the class context and are not meant to be a dictionary of the domain concepts. Likewise, the lack of details about the inner classes (mentioned in the final part of the summary) was considered as missing information by some participants. The reason we chose not to include it is that if the developer has a particular interest for a member class, she can be referred to its corresponding summary.

Content adequacy proved to be the hardest property to judge by the subjects. When judging 14 of the 40 summaries, at least two of the subjects answered with option #1 and #3 (see Table II), respectively; this indicates a significant disagreement in those cases. In contrast, when judging the conciseness and expressiveness of the summaries, such level of disagreement occurred only for two summaries, in each case. This is another indication of how difficult it is to automate the process of selecting the relevant information to be included in the summary of a class.

2) Conciseness. It is remarkable that only in five cases out of 132, the evaluators considered that the automatic descriptions have a lot of unnecessary information. These cases correspond to classes with *Boundary* (27%), *no-stereotype* (17%), and *Boundary+Data Provider* (15%) stereotype. For 11 out of 15 of evaluated class stereotypes, the summaries have no unnecessary information, according to the evaluators

Reviewing the cases were the evaluators said that the descriptions have some extraneous information, we found that most of them correspond to classes in the *no-stereotype* category, i.e., classes that do not match with any set of rules defined for the stereotypes. This is certainly an expected result, since there is little we can say about such classes at this point.

In the study, we asked the evaluators to provide the unnecessary information existing in the summaries. We found that most of this information corresponds to the descriptions of the stereotypes. Specifically, when evaluating *Lazy* and *Controller* classes, the participants mentioned that the summaries were satisfactory when ignoring such descriptions. In the case of *Boundary* classes, the interaction between classes was considered as useful, except when those classes belong to Java libraries. A quick improvement for this situation is to filter out such classes. Another problem in this regard referred to specifying the number of communicating classes that are not included in the summary. Evaluators suggested that this number was not useful if the classes were not mentioned. We plan to work on a better way of presenting such information, without affecting the conciseness of the summary.

Some specific methods were reported as unnecessary information. We found that these methods implement the *Singleton* pattern, which, according to the method stereotype rules, are classified as *mutators*. This is an issue when describing the behavior of a class. We consider that including this kind of methods in the creational category of the stereotype taxonomy could improve the summaries.

Finally, we found that some unnecessary information was caused by the poor quality of some of the identifiers. For example, methods with identifiers similar to *minutes2second* are described as "getting minutes to seconds", but a better choice would be "converting minutes to seconds." We also found that redundant information in the natural-language form of the methods was reported as unnecessary, particularly when the name of the method and the type and name of parameters overlap. We plan to improve the heuristics used in the lexicalization process.

3) Expressiveness. This is the characteristic best evaluated by the participants. Similar to conciseness, only in five out of 132 cases the evaluators considered that the automatic descriptions were difficult to read and understand. In contrast, in 88 of the cases they considered that the descriptions were easy to read and understand, and somewhat readable and understandable, in 39 cases. In our interpretation, these results reflect the fact that the text generation tools we used (previously developed by Sridhara [4]) produce expressive natural language phrases, and also, that the organization of the information is good. For many stereotypes, all the subjects answered with the best option, i.e., all evaluations indicated that the descriptions are easy to read and understand. Such is the case for classes with *Minimal Entity*, *Data Provider*, *Data Class*, and *Pool* stereotypes.

The only case in which the results can be considered somewhat deficient is the *Factory* stereotype, where 29% of the summaries were deemed as hard to read and understand. This result suggests that our heuristics need to be improved when a class consists mostly of factory methods.

D. Threats to Validity

Several factors influence the results of our evaluation. We only evaluated between 2-4 summaries for classes with a given stereotype. These classes were from two systems only. While we tried to maximize the number of evaluations given the number of participants we had, it is difficult to generalize the results. The participants were all graduate students with good knowledge of Java. It is conceivable that using professional developers, the results may have been different. Given the time the participants spent on understanding each class (10 minutes in average) and based on the evaluation of their summaries, we concluded that (in most cases) they properly understood the classes. However, we can hardly call them experts on that code. This makes them the right subjects for the study, since our summaries are intended to novice developers. Evaluations with the developers of the code may lead to different results. Future replications involving more data and more subjects would help in generalizing the results.

In addition, some learning effect may have occurred while the subjects judged the summaries. For example, once a participant evaluated the first summary, she knew what would be asked for the second summary. It is possible that while comprehending the subsequent classes, she was already thinking ahead to answer the three questions. One option we considered was to ask the participants to evaluate the summaries after they understood all (or three) of the six classes. Given the rather short time of the study (to avoid fatigue), we realized that short term memory would be involved in remembering fast about the classes they studied, so we elicited their evaluation immediately, rather than later. With the same goal of limiting the learning effect, we assigned the classes to the participants in a counterbalanced way: none of them received two classes with the same stereotype, and they evaluated the classes and systems in different order. We also analyzed the answers relative to the evaluated summaries and compared them with the rest – we observed no significant differences. We did the same with the last evaluated summaries, in order to assess the effect of fatigue (if any), and we observed no significant difference.

IV. Related Work

Automated summarization of natural language documents has been widely investigated by researchers in text retrieval (TR), natural language understanding, and cognitive psychology [14]. Unfortunately, such techniques have been used scarcely in software engineering, especially when it comes to source code summarization. Sridhara et al. proposed a technique to automatically generate natural language comments for Java methods [4, 15] and a technique to automatically generate comments that describe the high level role of a formal parameter in a method [16]. While we use here some of the tools developed in [4], the main contribution in this paper is that we are dealing with summaries for classes, which are more complex software constructs than methods or parameters. Previous work on summarizing classes (and methods) investigated the use of TR techniques [5, 17] and found that they are not very well suited for this problem. Such approach focused on extracting the most relevant words from comments and identifiers, while ignoring the code structure. In this work, we adopt a rather opposite view – we focus on the code structure and generate *natural language* summaries for classes.

Other work on summarization/documentation generation for source code-based artifacts has attempted to generate human readable descriptions for exceptions thrown by a method [18], changes done in a program [19], and cross-cutting source code concerns [6].

Recently, there have been promising approaches for summarizing other software artifacts, such as, bug reports [20]. The summarization of large execution traces was suggested as a tool that can help programmers to understand the main behavioral aspects of a software system [21].

In a somewhat different context (i.e., reverse engineering), Kuhn [22] used TR techniques to extracts words to label source code clusters. A similar approach is used in [23], where groups of methods returned as results of a search are labeled. In each case these labels can be considered as (partial) summaries.

V. Conclusions and Future Work

We presented an approach to automatically generate structured natural language summaries for Java classes. The approach leverages information about the class stereotypes and uses existing text generation tools to compose summaries based on a set of heuristics we developed. While the techniques used to generate the summaries are adapted from prior work, this is the first technique that automatically generates natural language summaries for classes.

Twenty-two programmers evaluated 40 summaries generated for classes from two Java systems. According to their evaluations, 69% of the generated summaries do not miss important information about the classes, 96% of the summaries are concise, and 96% are readable and understandable. These results are more than promising, and we are convinced that our approach can be used to generate summaries that would help developers when browsing and reading the code. The proposed summaries are not targeted to any specific development task, but they could be used as a starting point for the generation of task-specific summaries. When interpreting the results, one must bear in mind that our summarization technique is completely automated and does not make use of existing documentation or external domain knowledge.

The study also revealed several areas where the summarization process can be improved. We plan to carry out such improvements and hope that they will result in higher quality summaries. For example, for the stereotype identification we plan to enrich both taxonomies – method and class stereotypes. This will also help in refining the heuristics to select the relevant content of a class, as discussed in Section III.C. We also plan to consider the context of the class (i.e., the relationships to other classes) to enrich the summaries. When comments are present, they may also be used for generating the summaries. We will investigate this problem.

Finally, once we release a new version of the tool that includes the improvements suggested by this preliminary evaluation, we plan to perform an extrinsic evaluation where the generated summaries will be used to support specific maintenance tasks, in order to measure the impact of their use.

Acknowledgement

This work was supported in part by grants from the National Science Foundation (CCF-0845706, CCF-1017263, and CCF-0915803).

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering (TSE),* vol. 32, pp. 971-987, 2006.

[2] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering,* vol. 30, pp. 889 - 903, 2004.

[3] J. Starke, C. Luce, and J. Sillito, "Searching and Skimming: An Exploratory Study," in *International Conference on Software Maintenance,* 2009, pp. 157-166.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 43-52.

[5] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *17th IEEE Working Conference on Reverese Engineering*, 2010, pp. 35-44.

[6] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *27th IEEE Int. Conference on Software Maintenance*, 2011, pp. 103-112.

[7] N. Dragan, M. Collard, and J. Maletic, "Automatic Identification of Class Stereotypes," in *26th IEEE International Conference on Software Maintenance (ICSM')*, 2010, pp. 1 -10.

[8] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in *22nd IEEE International Conference on Software Maintenance (ICSM)*, 2006, pp. 24 - 34.

[9] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *33rd International Conference on Software Engineering*, 2011, pp. 101-110.

[10] N. Dragan, M. L. Collard, and J. I. Maletic, "Using method stereotype distribution as a signature descriptor for software systems," in *25th IEEE International Conference on Software Maintenance*, 2009, pp. 567-570.

[11] L. Moreno and A. Marcus, "JStereoCode: Automatically Identifying Method and Class Stereotypes in Java Code," in *27th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2012

[12] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," in *International Conference on Aspect Oriented Software Development (AOSD'07)*, 2007, pp. 212-224.

[13] E. Hill, "Developing Natural Language-based Software Analyses & Tools to Expedite Software Maintenance," Ph.D., Department of Computer and Information Sciences, University of Delaware, DE, 2010.

[14] K. Sparck-Jones, "Automatic summarising: The state of the art," *Information Processing and Management: An International Journal,* vol. 43, pp. 1449-1481, 2007.

[15] G. Sridhara, "Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs," Ph.D., Department of Computer and Information Sciences, University of Delaware, 2012.

[16] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating Parameter Comments and Integrating with Method Summaries," in *the 19th IEEE Int. Conference on Program Comprehension*, 2011, pp. 181-190.

[17] S. Haiduc, Aponte, J., Marcus, A., "Supporting Program Comprehension with Source Code Summarization," in *32nd ACM/IEEE International Conference on Software Engineering - NIER track*, 2010, pp. 223-226.

[18] R. Buse and W. Weimer, "Automatic documentation inference for exceptions," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 273-282.

[19] R. Buse and W. Weimer, "Automatically documenting program changes," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010, pp. 33-42.

[20] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing Software Artifacts: A Case Study of Bug Reports," in *International Conference on Software Engineering*, 2010, pp. 505-514.

[21] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," in *14th IEEE International Conference on Program Comprehension (ICPC)*, 2006, pp. 181-190.

[22] A. Kuhn, S. Ducasse, and T. Girba, "Semantic Clustering: Identifying Topics in Source Code," *Information and Software Technology,* vol. 49, pp. 230-243, 2007.

[23] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *15th IEEE Int. Conf. on Program Comprehension,* 2007, pp. 37-46.