Can We Automatically Generate Class Comments in Pharo?

Pooja Rani[1,*], Alexandre Bergel[2], Lino Hess[1], Timo Kehrer[1] and Oscar Nierstrasz[3]

[1] *Software Engineering Group, University of Bern, Switzerland*

[2] *RelationalAI, Switzerland*

[3] *feenk GmbH , Switzerland*

Abstract

Code comments support developers in understanding and maintaining codebases. Specifically in the Pharo environment, code comments serve as the main form of code documentation and usually convey information ranging from high-level design descriptions to low-level implementation details. Nevertheless, numerous important classes in Pharo still lack comments as developers find writing comments to be a tedious and effort-intensive task.

Previous works in Java have recommended generating comments automatically to reduce commenting effort and save developers time. There exist several approaches to achieve this goal. One such popular approach is based on identifying stereotypes, *i.e.*, a generalized set of characteristics supposed to represent an entity (object, class). However, this approach has not been tested for other programming languages. In this paper, we adopt the stereotype-based approach to automatically generate class comments in the Pharo programming environment.

Specifically, we generated information about the class type, collaborators and key methods. We surveyed seven developers to evaluate the generated comments for 24 classes. The responses suggest that, although more information could be added to the comments, the generated class comments are readable and understandable, and the majority of comments do not contain unnecessary information.

Keywords

Comment analysis, Software documentation, Program comprehension, Documentation generation, Pharo

CEUR-WS.org/Vol-3325/regular4.pdf

1. Introduction

Developers spend a significant amount of time to understand source code [?]. It is well-established that code comments are heavily used for code comprehension [?]. Object-oriented

IWST'22: International Workshop on Smalltalk Technnologies

*Corresponding author.

pooja.rani@unibe.ch (P. Rani)

https://seg.inf.unibe.ch (P. Rani); http://bergel.eu (A. Bergel); https://seg.inf.unibe.ch (L. Hess); https://seg.inf.unibe.ch/people/timo/ (T. Kehrer); https://feenk.com/about/ (O. Nierstrasz)

0000-0001-5127-4042 (P. Rani)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

programming languages support comments at various levels. For instance, class comments in Java are expected to present a high-level overview of a program whereas method comments present implementation details of a method [?]. In contrast to Java, Smalltalk class comments contain high-level as well as low-level implementation information [?]. Class comments in Pharo are considered to be a primary source of code documentation, and they are used to obtain the high-level overview as well as its implementation details.

Although Rani *et al.* have shown that there has been an increase in classes being commented in Pharo versions over time [?], many key classes still lack comments, and many existing comments have become outdated or inconsistent over time. Several other programming languages show the same symptoms of outdated or missing comments due to rapid project schedules or developer neglect [?].

To address these concerns, researchers started to investigate various automatic code summarization and comment generation approaches [? ? ? ?]. One basic and popular code summarization technique is template-based stereotype-identification (SI), which has been used by Moreno *et al.* to generate summaries (or comments) for Java classes [?]. They defined a template for various stereotypes that classify the functionality, characteristics and general idea of classes and methods. Although their approach has been shown to be effective for the Java environment, it has not been tested for other programming languages, specifically for languages where a comment template already exists. For example, Pharo presents a default comment template to guide its developers to write class comments. Rani *et al.* studied comments of multiple programming languages, and showed that, although developers write various similar kinds of information in Java and Smalltalk comments, they use different conventions to write them [?]. Such differences can make it a difficult task to adopt techniques across languages. Since there has been an increase in the number of multi-language environments, it is essential to test such techniques across multiple languages to better generalize them.

Given the importance of replication studies in Software Engineering (SE) [?], we replicated the SI-based approach proposed by Moreno *et al.* and Dragan *et al.* to generate comments in a fast and uniform way [? ? ?]. As Smalltalk and Java environments differ in many aspects, it is necessary to adapt their approach in various ways.

To evaluate the generated comments, we surveyed seven experienced and novice Pharo developers. The participants were asked to evaluate the generated class comments based on their adequacy, conciseness, and comprehensibility. The evaluation showed that some areas of our adapted approach can still be improved, but it also showed that the majority of our generated class comments are adequate, mostly contain no unnecessary information, and are easily readable and understandable. Our replication package (RP) (including all scripts and evaluation results) is available on Zenodo [?].[1] Future work will focus on augmenting comments with additional important information and expanding the survey.

[1]https://doi.org/10.5281/zenodo.6622011

2. Background

Numerous approaches exist to extract important information from source code that can help developers to understand source code better, such as *Information Retrieval* (IR), *Machine Learning* (ML-based), *Artificial Neural Networks* (ANN), and *Stereotype identification* (SI-based) [**?**]. Each approach has its own advantages and disadvantages. For instance, IR-based approaches require extensive effort [**?**], but fail to capture precise keywords if methods or variables are poorly named. ML-based or ANN approaches often require a significant number of (typically pre-labeled) datasets. A popular yet basic technique is the heuristical *SI-based* approach. It has been successfully used to identify indicators of code smells [**?**], adding comprehension to unit test cases with stereotype-based tagging [**?**] to create signature descriptions for software systems based on method stereotype distributions [**?**], or to generate summaries for classes [**?**].

Stereotypes consist of a set of characteristics of an object or person, and they are often used to classify the functionality, intent, and behavior of classes or methods. For example, a method that only retrieves data will be classified as an *Accessor* method stereotype, and a class with only such methods will be classified as a *Data provider* class stereotype. A stereotype-based summarization exploits such method or class stereotypes to fill out a pre-defined summarization template.

Moreno *et al.* used this approach to generate summaries for classes. As class comments contain other information than just that of its methods, identifying the stereotypes of all methods and bundling them together in a class comment would not be a viable option. They leveraged various heuristics to choose which method stereotypes should be included in a class comment and which ones not. They conjecture that the method types and their distribution in a class denote some design decisions, which eventually reflect the main goal of the class. They considered three main aspects to generate a summary:

A. The information to include in the summary.

B. The level of details to include in the summary.

C. The SI-based approach to generate and present the information.

Their approach first identifies the class stereotype (*i.e.*, represent the intent of classes in a system's design) [**?**] and its methods' stereotypes (*i.e.*, represent the responsibilities of the methods in a class) from a list of stereotypes [**?**]. The list includes 15 method stereotypes and 13 class stereotypes. It is then combined with predefined heuristics to filter the information to present in the comment. We adopted their approach and adapted it for Pharo.

3. Study Design

Each phase described by Moreno *et al.* to generate comments required various adaptions for Pharo. We describe these adaptions in the following subsections.

3.1. Information to include in the summary

Moreno *et al.* defined a class summary template that consists of information such as the names of superclasses, interfaces, inner classes, class attributes, and methods [?]. In contrast to their custom-made template, a default class comment template is available in Pharo, and it is presented to developers when they add a class comment for the first time. Rani *et al.* identified seven different types of information in this template namely, *Intent*, *Responsibility*, *Implementation Points*, *Public APIs*, *Examples*, *Instance Variables*, and *Collaborators* [?]. They also found that these template-suggested information types are written more frequently compared to other information types found in comments. We aim to generate these seven recurrent information types for our summaries.

3.2. Level of details to include in the summary

Not all of these information types can be easily generated. Also, the level to which their details can be captured from source code is limited.

For instance, *Intent* and *Responsibility* information are rather crucial for class comments, but they are implicit by nature [?]. This means that such information types do not have an explicit header or specific common keyword, or the information is not separated by any formatting structure (space, special symbols). This makes it hard to automatically identify or generate them. We represented these information types through the descriptions of their class stereotypes in the class comment. For example, a class that contains many controller methods will be controlling some kind of data flow and will thus be assigned the *controller* stereotype. We additionally combined this information with a list of relevant keywords, to give a broad sense of what the *Intent* and *Responsibility* of a class are. For example, the class OrderedCollection in the Pharo base image is used by a total of 974 classes, but itself only uses five classes (*e.g.*, Array class) for its functionality.

Internal details or *Implementation Points* refer to the internal representation of the objects, or particular implementation logic about the object state, and settings important to understand the class. For *Internal details* we used a broader approach, by displaying a general overview of the internal details using relevant keywords, without meticulously displaying every facet of them.

Public APIs are the key methods and public APIs of the target class. In our work, these have been separated into two main categories, specifically: *internal* and *external* usage of methods. When other classes call a method, it is defined as *externally* used, whereas when a method is used within the class, it is defined as *internally* used. For example, a helper method that filters a string and is used only within the class in which it is defined will be categorized under the *internal* usage, whereas a setter method of the class used by another class to set an object will be categorized under the *external* usage. This helps us to display how a class functions internally in contrast to how it is used by other classes.

Examples are simple code fragments that show how the class is to be instantiated or used. Since we considered only the source code of the target class, and this information requires other classes to be considered, we omitted it for now.

Instance Variables are the private fields (or slots) of an instance of the class.
The next subsection describes the approach used to extract these pieces of information and present them in the summaries.

3.3. SI-based approach to generate and present information

In this approach, we first identify the method stereotypes for all methods of a class, then a stereotype for the class, and lastly extract the relevant information for the identified stereotypes. Figure 1 illustrates these main steps.

Table 1
Description of main method stereotypes

Method Stereotype	Description	Heuristics
Accessor	Accesses or retrieves data.	Returns an instance variable or primitive
Mutator	Changes the state of an object.	There are no return types or only boolean returns
Creational	Provides an interface to create or instantiate objects.	Returns a created object
Collaborator	Represents the collaboration between multiple objects.	At least one variable which is manipulated or passed in the method is an object
Degenerate	A method that does not belong to above stereotypes.	

1. *Identify method stereotypes.* As a first step, we identify the method stereotypes for each method of a class based on the predefined heuristics for each method stereotype. We adopt a total of five method stereotypes and three sub-stereotypes, shown in Table 1, from the work of Dragan *et al.* [? ?].[2] We discard some of their method stereotypes due to their inapplicability to Smalltalk, such as void-accessor methods which return void (Smalltalk methods always return a non-void value). Some stereotypes require adaptation, for instance, the concept of primitive types in Smalltalk is not the same as in Java. We use the AST-based (abstract syntax tree-based) approach to extract particular information about the methods to determine their stereotypes, *e.g.*, the node RBReturnNode in an AST presents the return values of a method, and RBVariableNode node represents a variable in it. Thus, we could access the relevant information about the method and check it against various method stereotypes to find its stereotype.

2. *Identify class stereotypes.* By aggregating the method stereotypes (based on their frequency), the class is assigned a class stereotype. Dragan *et al.* defined 13 class stereotypes and heuristics to identify a class stereotype [?]. We add an *Empty* stereotype to allow us to cover certain classes in Pharo, *i.e.*, classes that contain empty or no methods, *e.g.*, Errors or annotations [?]. Similar to method stereotypes, we adapt the heuristics according to the Pharo environment.[3] The list of class stereotypes with their descriptions is shown in Table 2.

[2] The detail of method stereotypes with examples is presented in File "RP/Appendix"
[3] The heuristics for class stereotype are presented in File "RP/Appendix"

Table 2
Description of class stereotypes

Class Stereotype	Description
Boundary	Contains a high percentage of collaborator methods, a low percentage of controller methods, and a few factory methods.
Commander	Contains mainly mutator methods.
Controller	Controls other objects. It consists of controller and factory methods.
Data	Contains only getter and setter methods.
Data Provider	Consists mostly of accessor methods.
Degenerate	Contains mostly degenerate methods.
Empty	Contains empty methods or no methods, *e.g.*, annotations or errors.
Entity	It encapsulates data and behavior.
Factory	Contains mostly factory methods.
Large	Contains various functionalities (multiple method stereotypes) and has high number of LOC.
Lazy	Contains getter or setter methods, a high percentage of degenerate methods, and a low percentage of other methods.
Minimal Entity	A kind of an entity class that consists entirely of getter and setter methods.
Pure Controller	Contains entirely controller and factory methods.
Small	Contains just a few methods (up to two methods) or not at all.

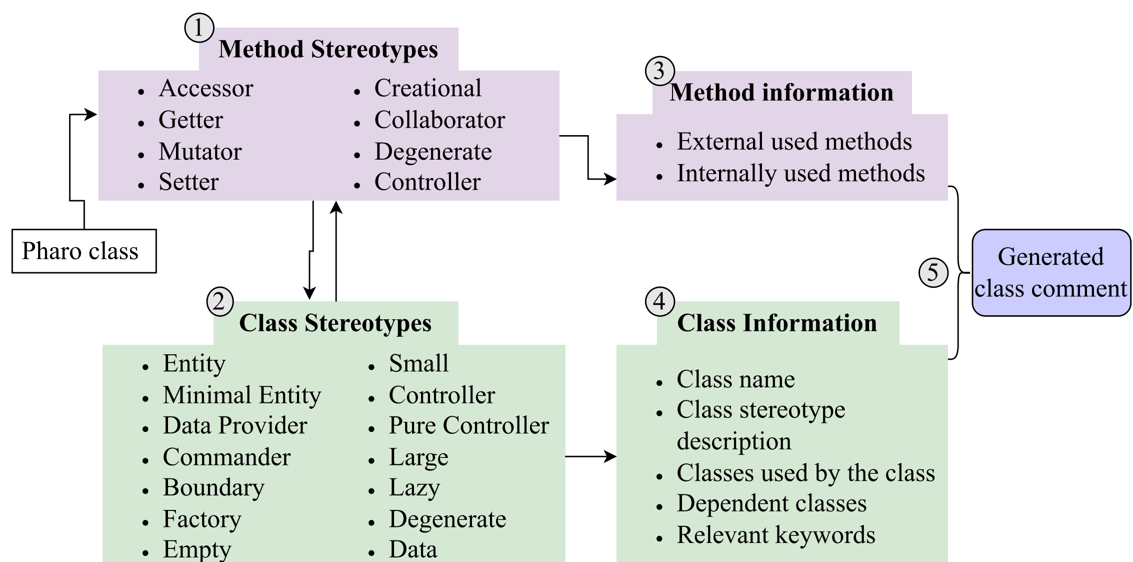

Figure 1: Process of generating a class comment

3. *Extract relevant information for a class stereotype.* In the last step, we extract the relevant information for the corresponding class stereotype and display it in the class comment. For example, in a *Data Provider* class, relevant information will mostly be based on its *Accessor* and *Getter* methods, as these comprise the class stereotype's root characteristics. If a class belongs to more than one class stereotype, all methods relevant for each stereotype

are collected in the relevant set of methods. Once the relevant methods are collected, we rank them based on their internal or external usage in the class, and display the five most-used methods per class. We restrict the output to five methods to not clutter the class comments and overwhelm developers.

4. *Present the information in the template.* To keep our generated comments similar to the existing template, we present the extracted information types in the comment in the same order and style. Similar to the template, there is no separate section defined for the *Intent* or *Responsibilities* of the class, but the top lines in the comment indicate these information types. Rani *et al.* described in their study that such information types are extremely difficult to automatically identify or generate. However, we covered them by describing the class stereotypes in a broad sense. For example, in the class RSShape shown in the Listing 1, the lines I have class stereotype: DataProvider, and I encapsulate data. I consist mostly of accessor methods indicate *Intent* and *Responsibilities* of the class. Similarly, the *Collaborator* section is covered by mentioning the classes that the target class uses (*i.e.*, using the method dependentClasses in Behavior class) and that use the target class (*i.e.*, References to the class). We use *Public APIs* are covered under the *relevant public method* section. *Examples* are not available in our generated class comments as we focused only on the source code of the target class. For future work, we plan to consider analyzing source code of other classes as well and extracting the usage of the class in other classes. The list of instance variables covers the corresponding *Instance variables* section in the template. We emulated the *Internal details* using the frequent keywords under the list of keywords. This provides a general feeling and understanding of the language used in the source code of a class, and can give an initial impression of the structure of a class.

   A final automatically generated class comment can be seen in the Listing 1. The figure shows the class name, identified class stereotype, and its description to provide the intent of the class in a broad sense. In contrast to the work of Moreno *et al.*, we added the collaboration relation of the class (shown as "using the classes" and "used by classes") [**?**]. Overall, we expressed four of the seven information types in an explicit way and the other three in a more indirect way, to produce a broader description of the class.

```
Classname: RSShape

I have class stereotype:
  – DataProvider

I encapsulate data. I consist mostly of accessor methods.

I am using the classes:
– RSObjectWithProperty
– RSShapeAddedEvent
– Color

I am used by classes:
– RSCanvas
– RSComposite
– RSCustomCPController
– RSTContainer

I have relevant public methods which are ordered by their usage:
Externally:
– models
– width
– height
– extent
– parent

Internally:
– shape
– extent
– canvas
– model
– encompassingRectangle

My instance variables are:
– paint
– path
– border
– parent
– isFixed
– encompassingRectangle
– model

My defining keywords are:
border, is, shape, with, paint, color, parent, rectangle, encompassing, has
```

Listing 1: Automatically generated class comment for the class RSShape

3.4. Evaluation

To assess and gain better insight into our adapted SI-based approach, we evaluated it by:

a) analyzing the distribution of stereotypes, and

b) surveying developers.

Specifically, we analyzed the distribution of 350 class stereotypes (a statistically significant sample achieved with confidence 95% and 5% error margin from all classes available in Pharo 9), and the distribution of method stereotypes of 500 classes. The classes were selected randomly and their generated comments were manually validated. In the next evaluation step, we compared the information types of our generated class comment with the template-suggested information types.

To keep our evaluation study design close to that of Moreno *et al.* [**?**], we surveyed 12 Pharo developers with varying degrees of experience in Smalltalk and different domains. Four of them had been working on projects such as *Roassal* or *GToolkit*,[4] two were moderately experienced in Pharo, and the rest were doctoral students working with Pharo.

We randomly selected the classes for one questionnaire specifically from the *Roassal* system. The remainder of the Pharo classes were selected randomly, and sorted based on their stereotypes and their quality in representing specific class stereotypes. We selected two classes per class stereotype, for a total of 24 classes to be evaluated. We wanted each class stereotype to be evaluated by at least two different developers, and preferably by three. Unfortunately, five of the invited developers did not complete our evaluation, thus reducing the results to seven developers in total. This led to a total of 24 classes being evaluated in four different evaluation forms by a total of seven developers.[5] We selected these 24 classes from Pharo 9 version. We used Google's online survey tool for the evaluation to reduce the Hawthorne effect[6] and to provide an easy way to collect answers.

The participants filled out their questionnaires independently with no time restrictions and in whatever environment they wanted to use. They could spend as much or as little time on the different classes as they wanted. The first section focused on the demographic information of the developers, while the next section focused on the kind of information they write or look at in comments. Our aim with this information is to reflect on the first aspect (information to include in the summary).

Each participant received a questionnaire with a total of 6 classes. These classes represented different class stereotypes. The distribution of the classes had been established beforehand, to ensure that each participant would have an even distribution of classes and class stereotypes to evaluate. Classes were distributed such in a manner that each questionnaire roughly contained the same number of methods, so that the effort to evaluate one questionnaire was not significantly greater than another. One of the questionnaires was specifically filled with only Roassal classes to incorporate the expertise of certain developers, which were core developers in Roassal.

---

[4] https://github.com/ObjectProfile/Roassal3, http://gtoolkit.com/

[5] https://bit.ly/3Hc4a2a

[6] A type of reactivity in which individuals modify an aspect of their behavior in response to their awareness of being observed

To ensure that developers understood the class, we asked participants to familiarize themselves with the major functionality and structure of the systems by reading brief descriptions of a class and, if needed, by executing the system. They were allowed to look at existing class comments, if there were any. After having understood the class we asked them to write their own description for the class. With these steps establishing a certain baseline of knowledge about the target class, participants were asked to evaluate their understanding based on the CRC design and the template. To evaluate their understanding, we asked various questions (related to the class), proposed by Moreno *et al.* (presented in the RP), expecting no predefined answers. Once their own evaluation of the target class had been completed, we presented them with the automatically-generated class comment of that class, produced by our approach. We asked them various questions to evaluate the generated comment based on various quality attributes, such as adequacy, conciseness, and expressiveness of the generated class comments [**?**] (Please see the RP for the questions.)

Table 3
Questions and possible answers to evaluate adequacy, conciseness and expressiveness of the generated class comments

Question	Possible Answers
Do you think the comment is complete?	• It is not missing any important information • It is missing some information but the missing information is not necessary to understand the class • It is missing some very important information that can hinder the understanding of the class
Do you think the comment is concise?	• It is has no unnecessary information • It is has some unnecessary information • It is has a lot of unnecessary information
Do you think the comment is understandable?	• It is easy to read and understand • It is somewhat readable and understandable • It is hard to read and understand

4. Results and Discussions

Distributions of stereotypes Figure 2 shows the distribution of method stereotypes from 500 classes and Figure 3 shows the distribution of class stereotypes from 350 classes. Figure 2 shows that accessor methods are frequently present, closely followed by the collaborator stereotype. However, we did not find many controller methods. One of the reasons for such discrepancies can be differences in the definition of objects in Java and Smalltalk. According to Dragon *et*

al.'s definition [?], collaborator and controller stereotypes handle objects, but not everything is considered to be an object in Java in contrast to Smalltalk. Other reasons for the discrepancies include the widespread usage of class side methods, lambda expressions and reflection. For future work, we propose to consider more empirical experiments to understand the differences further.

In terms of class stereotypes, Figure 3 shows that the *Data Provider* stereotype is the most frequent (two out of five classes). We did not find any *Pure Controllers* in the selected sample of classes. In our current study, we did not tune the defined thresholds in the heuristics (used to identify the class stereotypes). Also, since the Pharo environment includes various other design patterns (*e.g.*, Models, Views, and Facades), method conventions, or method protocols, more sophisticated methods can be adopted to improve the identification of class stereotypes. Another challenge with our approach is related to using ASTs. Although AST-based approaches are powerful, they can be deep to traverse, they fail to capture run-time details for objects, and special considerations for handling exceptions. Future work can leverage combination of advanced approaches such as type inferences, or neural network-based approaches that can extract deep or hidden features from source code to identify the method and class stereotypes more accurately.

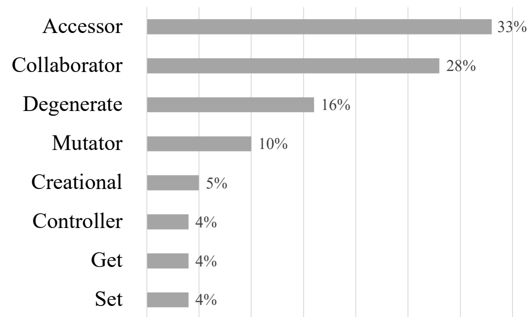

Figure 2: Distribution of method stereotypes for 500 random classes

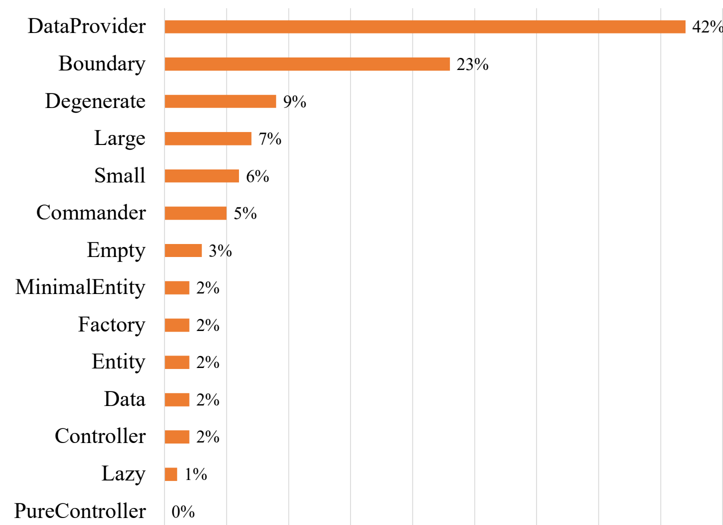

Figure 3: Distribution of Class stereotypes for 350 random classes

Surveying developers To further find out how accurate and helpful the comments are, we created an online evaluation, where multiple developers assessed our class comments for their completeness, conciseness, and comprehensibility.

All participants came from a software development background. Roughly 29% had one to four years of experience in Smalltalk and Pharo. One participant claimed to have between four and seven years of experience in Smalltalk, whereas two stated that they had that amount of experience in Pharo. Another participant claimed to have between seven and ten years of experience in Smalltalk, yet none of the participants had that amount of experience in Pharo. A total of 43% of the participants stated that they had more than ten years of experience in Smalltalk, as well as in Pharo.

In answer to how often they write class comments, many of the participants claimed they write them *Fairly often* or at least *Sometimes*. None of the participants claimed they do not write any documentation at all or just rarely. That said, no participant claimed to write documentation all the time.

We received a total of 42 evaluations for the 24 class comments we generated. The results for each category, namely adequacy, conciseness and comprehensibility of our generated class comments can be seen in Table 4 to Table 6.[7] Overall, the results are positive, with a majority of the responses being positive, though only marginally sometimes. We can say that the majority of automatically-generated class comments are understandable and readable, the information is somewhat adequate, though we sometimes miss important information, and the majority of comments only have some or no unnecessary information. With that said, we have to

[7]Folder "RP/Dataset/Online_evaluation"

Table 4
Distribution of participants' responses to adequacy

Adequacy	
Answer	Percentage
• It is not missing any important information	20%
• It is missing some information but the missing information is not necessary to understand the class	36%
• It is missing some very important information	44%

Table 5
Distribution of participants' responses to conciseness

Conciseness	
Answer	Percentage
• It has no unnecessary information	28%
• It has some unnecessary information	26%
• It has a lot of unnecessary information	46%

acknowledge that the distributions for the adequacy and the conciseness of the generated class comments have potential for improvement. We present a more thorough assessment for the different criteria in the following sections.

The adequacy of our generated class comments is the most relevant criterion. It showed that the majority of evaluated class comments do not miss any, or only some, important information of a class. Classes with the class stereotypes *Controller*, *Empty*, *Commander*, and *Minimal entity* prove to be the most adequate class comments, with 75% to 100% of the class comments have no or little missing information. On the other hand, classes with the *Degenerate* stereotypes are considered to be missing important information. This is expected as such classes contain empty or small methods. Similarly, *Entity* and *Large* classes are also found to be missing information. Large classes can have several responsibilities, thus summarizing such classes can be a challenge. Entity classes encapsulate data and behavior, thus choosing which behavior to include in the summary can be a challenge. In general, the answers indicated that if important information is missing, it proved to be one of two categories. The first category entails missing information about structure, architecture, and integration of the target class, and second concerns missing relevant methods. The problem of missing methods can be due to the methods neglected due to a certain class stereotype. For example, *Data Provider* classes tend to display more accessor methods than other types of methods. Nevertheless, such biases can be tackled by balancing the method stereotypes for certain class stereotypes.

Regarding the *Conciseness* attribute, a total of 54% of evaluated class comments contain no or just some unnecessary information. Classes with the stereotypes *Commander*, *Controller*, *Data Provider*, *Empty* and *Minimal Entity* contain 0 to 25% unnecessary information. The *Data* classes were evaluated to contain too much information by all participants. This stems from the fact that they usually only contain getter and setter methods and do not hold much functionality,

Table 6
Distribution of participants' responses to comprehensibility

Comprehensibility	
Answer	Percentage
• It is easy to read and understand	46%
• It is somewhat readable and understandable	41%
• It is hard to read and understand	13%

so they do not need a lot of documentation. Classes with the *Large* stereotype were flagged by three out of four participants as containing too much information. Contrary to the way in which *Data* classes do not hold much functionality, *Large* classes contain a lot of functionality. In particular, participants commented on the excess of information, especially the long lists of classes that are used by, or use the target class.

Some methods mentioned in class comments were marked as being unnecessary. Participants remarked on missing methods and in turn unnecessary methods being described in the class comment. As previously stated, this should be adopted by reviewing the process in which we decide which methods are representative of a class stereotype. We plan to improve this process to include more relevant information.

The expressiveness or comprehensibility of our generated class comments is by far the best evaluated characteristic. A total of 87% of the class comments were either easy to read and understand or at least somewhat readable and understandable. Many class stereotypes were evaluated as easy to understand or read. Only the class stereotypes of *Boundary*, *Degenerate*, *Entity*, *Factory* and *Small* were declared as being hard to understand or read. These class stereotypes though were evaluated to be hard to read by a maximum of 33% of the participants.

Overall, in comparison to the work of Moreno *et al.*, we focused on various recurrent information types in the Pharo environment. Some information types, such as *Collaborators*, *Implementation Points* are not covered in their work. In terms of evaluating adequacy of generated class comments, classes were found to be missing more information in the Pharo environment — *Entity* by 75% of the participants, *Large* 75%, and *Factory* 66% — compared to the evaluation of Moreno *et al.* for Java classes — *Entity* 29%, *Large* 0%, and *Factory* 43%. Regarding the conciseness of our generated class comments, we found a similar difference for the (*Entity*, *Large*, and *Factory*) classes. We speculate that such differences can exist due to (i) different conventions used in writing Java and Smalltalk code, *e.g.*, existence of metaclasses, duck typing, strict encapsulation in Smalltalk, and (ii) the different kinds of information developers are expected to write in their class comments, *e.g.*, Java developers are expected to write high-level design overview in a class comment [?], whereas Smalltalk developers are expected to write information ranging from high-level design information to low-level implementation details. Therefore, Smalltalk developers can perceive missing information more. However, such speculations need some carefully designed control studies to investigate them further.

5. Threats to validity

Numerous factors can influence our results and evaluation.

Threats to internal validity. This mainly concerns the definition of various class and method stereotypes and the heuristics used to identify them. To limit this threat, we adopted the initial definition and corresponding heuristics from the work of Moreno *et al.*. We contacted two Pharo developers, both with several years of experience, to review the heuristics and stereotypes definition.

The template used to generate a summary can also introduce bias in the comment generation process. In contrast to the custom template built by Moreno *et al.*, we used the default template available in Pharo. The information suggested by the template might not represent all the information developers would like to see in a class comment. However, developers write these information types more often compared to other information found in class comments. We found in our evaluation study that roughly 56% of our class comments are not missing any relevant information.

Threats to external validity. We only evaluated two summaries per stereotype, for a total of 24 classes. The classes belong to only one software system, the core Pharo environment. However, this environment covers diverse domains, *e.g.*, Files, User interface. While we focused on maximizing the number of evaluations given the participants we had — we invited twelve developers and seven responded — it is difficult to generalize the results. Many participants were graduate students who, although they had as good knowledge of Smalltalk or Pharo as the industrial developers of various projects, their results might vary. As various project communities might propose their own commenting guidelines, their expectation of what should be included in a template can vary and can eventually influence the results.

Other factors that can influence our results are, the pre-existing knowledge of developers and the learning effect. We prepared different evaluation forms for developers with different backgrounds, *e.g.*, we assigned one class from the Roassal system to an expert developer who is an expert in Roassal, and one novice developer in Roassal. Some learning effects might have occurred when the participants judged generated comments. For example, developers could read the existing comment of a class, if there is any, and could be influenced in terms of their expectations from an automatically generated comment. Since we wanted the inexperienced developers also to obtain a better understanding of a class, we let them explore all available sources. We recommend controlling such factors for future work. Another learning effect that can influence the results is that a participant evaluating the first summary might think ahead about the next summary as they knew they were expected to evaluate two summaries. To limit the such bias, we assigned each participant two different classes, with different stereotypes. Also, to avoid fatiguing participants, we provided them with short questionnaires, and a limited number of classes to evaluate, *i.e.*, each participant evaluated six classes.

6. Related Work

Code summarization. Code summarization generates readable summaries based on source code. Zhu *et al.* systematically analyzed 41 code summarization studies, their techniques, and the evaluation used for each study [?]. They found five main data extraction methods, *Information Retrieval* being one of the most frequently-used methods (41% of the studies) followed by the methods based on *Machine Learning and Artificial Neural Networks* (32% of the studies) and *Stereotype Identification* (20% of the studies). They also found that *Template-based* summarization is one of the most common natural language summary generation methods, with 46% of studies using this method to generate summaries. In terms of the evaluation method, they found *Manual evaluation* (56%) still being the most used method followed by *Statistical Analysis etc.* They reported that 10% of the studies *did not use any evaluation* at all. We used the *Stereotype-based* approach and manual evaluation.

Template-based summarization. Such approaches contain a predefined set of summary templates to be filled in by the target code segment and further information [?]. Dawood *et al.* [?] and Hammad *et al.* [?] filled the predefined templates with the required information, such as program structure information. In contrast, Wang *et al.* [?] used an NLP-based approach to find actions, themes and secondary arguments, and fill them into their template. Zhu *et al.* also discovered that one of the more prominent and closely-related methods for generating template-based summarization information is based on stereotype identification [?]. We used the Pharo default class comment template for our approach. As the template has been in the environment since its first version and Rani *et al.* showed that developers follow this template while writing class comments, we used it to generate class comments.

Stereotype-based approach. Several researchers have used the stereotype-based approach in recent years [? ?]. Abid *et al.* used it to generate natural language summaries for C++ methods [?]. Moreno *et al.* defined a heuristics-based approach for stereotypes, to identify code and build comment structures for the Java environment [?]. They argued that one cannot just add all comments of the different methods together to automatically create viable class comments, as (i) classes contain other information than just methods, such as data that the methods operate on, (ii) bundling all method descriptions would result in an enormous comment, which defeats the purpose of what they set out to do, and (iii) some methods may just not be relevant for the behavior of a class. Aligned with their approach intent, we replicated their study for Pharo comments.

Pharo comment analysis. Rani *et al.* found that the trend of commenting classes increased rapidly for initial Pharo versions, to then be maintained in subsequent versions [?]. Also, they found that developers keep changing comments of old classes to keep them up to date. Our tool was developed with the mindset to support commenting classes more frequently.

7. Conclusions

Given the importance of code comments for program comprehension, we focused on supporting developers in writing informative and understandable class comments in the Pharo environment. Moreno *et al.* proposed a SI-based approach to automatically generate a class summary (or comment) for Java classes. We adopted their approach for the Pharo environment and attempted to generate various information types suggested by the Pharo comment template.

We evaluated the approach by conducting an online survey with seven developers. The responses suggest that 87% of the summaries are easily understandable, and 56% of summaries are complete and concise. The generated class comments are aimed to support developers in creating class documentation. Future work will focus on adapting and extending the heuristics more accurately to the Pharo environment. We plan to develop a tool to support developers in writing class comments.

Acknowledgments

We thank Dr. Nitish Patkar for reviewing the paper on a short notice.

References

[1] A. Ko, B. Myers, M. Coblenz, H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, Software Engineering, IEEE Transactions on 32 (2006) 971 –987. doi:10.1109/TSE.2006.116.

[2] S. C. B. de Souza, N. Anquetil, K. M. de Oliveira, A study of the documentation essential to software maintenance, in: Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information, 2005, pp. 68–75.

[3] E. Nurvitadhi, W. W. Leung, C. Cook, Do class comments aid Java program understanding?, in: 33rd Annual Frontiers in Education, 2003. FIE 2003., volume 1, IEEE, 2003, pp. T3C–T3C.

[4] P. Rani, S. Panichella, M. Leuenberger, M. Ghafari, O. Nierstrasz, What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk, Empirical Software Engineering 26 (2021) 1–49. URL: http://scg.unibe.ch/archive/papers/Rani21b.pdf. doi:10.1007/s10664-021-09981-5. arXiv:2005.11583.

[5] R. C. Seacord, D. Plakosh, G. A. Lewis, Modernizing legacy systems: software technologies, engineering processes, and business practices, Addison-Wesley Professional, 2003.

[6] S. Haiduc, J. Aponte, L. Moreno, A. Marcus, On the use of automated text summarization techniques for summarizing source code, in: 2010 17th Working Conference on Reverse Engineering, IEEE, 2010, pp. 35–44.

[7] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, K. Vijay-Shanker, Automatic generation of natural language summaries for Java classes, in: 2013 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 23–32.

[8] P. W. McBurney, C. McMillan, Automatic source code summarization of context for Java methods, IEEE Transactions on Software Engineering 42 (2015) 103–119.

[9] Y. Zhu, M. Pan, Automatic code summarization: A systematic literature review, 2019. arXiv:1909.04352.

[10] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, O. Nierstrasz, How to identify class comment types? A multi-language approach for class comment classification, Journal of Systems and Software 181 (2021) 111047. URL: http://scg.unibe.ch/archive/papers/Rani21d.pdf. doi:https://doi.org/10.1016/j.jss.2021.111047. arXiv:2107.04521.

[11] O. S. Gómez, N. Juristo, S. Vegas, Understanding replication of experiments in software engineering: A classification, Information and Software Technology 56 (2014) 1033–1048.

[12] N. Dragan, M. L. Collard, J. I. Maletic, Reverse engineering method stereotypes, in: 2006 22nd IEEE International Conference on Software Maintenance, IEEE, 2006, pp. 24–34.

[13] N. Dragan, M. L. Collard, J. I. Maletic, Automatic identification of class stereotypes, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.

[14] L. Hess, Generating automatically class comments in Pharo, Bachelor's thesis, University of Bern, 2021. URL: http://scg.unibe.ch/archive/projects/Hess21a.pdf.

[15] M. J. Decker, C. D. Newman, N. Dragan, M. L. Collard, J. I. Maletic, N. A. Kraft, Which method-stereotype changes are indicators of code smells?, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2018, pp. 82–91.

[16] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, Aiding comprehension of unit test cases and test suites with stereotype-based tagging, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 52–63.

[17] N. Dragan, M. L. Collard, J. I. Maletic, Using method stereotype distribution as a signature descriptor for software systems, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 567–570.

[18] K. A. DAWOOD, K. Y. SHARIF, K. T. WEI, Source code analysis extractive approach to generate textual summary., Journal of Theoretical and Applied Information Technology 95 (2017) 5765–5777.

[19] M. Hammad, A. Abuljadayel, M. Khalaf, Summarizing services of Java packages, Lecture Notes on Software Engineering 4 (2016) 129.

[20] X. Wang, L. Pollock, K. Vijay-Shanker, Automatically generating natural language descriptions for object-related statement sequences, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 205–216.

[21] J. Hu, S. Qian, Q. Fang, C. Xu, Attentive interactive convolutional matching for community question answering in social multimedia, in: Proceedings of the 26th ACM International Conference on Multimedia, MM '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 456–464. URL: https://doi.org/10.1145/3240508.3240626. doi:10.1145/3240508.3240626.

[22] N. J. Abid, N. Dragan, M. L. Collard, J. I. Maletic, Using stereotypes in the automatic generation of natural language summaries for C++ methods, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 561–565.