Generating automatically class comments in Pharo

Bachelor's Thesis

Lino Hess

from

University of Bern

Faculty of Science, University of Bern

31.07.2021

Prof. Dr. Oscar Nierstrasz

Pooja Rani

Software Composition Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Code comments play a vital role in the development, comprehension and maintenance of code-bases as shown by many works before. Developers use code comments to explain various types of information such as rationale, examples, future tasks, usage of a class or method and many more. In Pharo, a Smalltalk based environment, code comments comprise up to 15% of the code. They are the main form of code documentation in Pharo. However, developers find writing code comments a tedious and boring, or even hard task. Therefore, code often lacks comments and makes the code hard to understand and follow.

Previous works have proposed to automatically generate comments to reduce the commenting efforts and to support developers. Automatically generating comments helps developers spend less time on writing code comments and writing uniform code comments. One of the more popular approaches to generate comments automatically is based on stereotypes. A stereotype is a generalized idea or set of characteristics believed to represent an object or person. In software development class or method stereotypes are often used to classify the functionality of the methods or classes in broad strokes. We choose a similar stereotype-based approach for Pharo code comments. Our work adapts heuristics of similar works, to allow us to generate method and class stereotypes for classes in Pharo. With this in mind we created a tool that lets us generate class comments automatically based on these class stereotypes. We illustrate our approach with the help of a running example in Pharo.

Using the tool we created, we discovered that a total of roughly 32% of methods are accessor methods, 27 % are declared as collaborator methods and about 15 % of methods are declared as degenerate methods in the Pharo base image. In relation to this we discovered that majority (41%) of the classes are Data Provider classes, 22% are Boundary classes followed by Degenerate (8%) and Large (7%) classes.

A number of developers evaluated the generated output of our tool, and determined that our automatically generated class comments are are understandable and readable, the information is adequate, and the majority of comments do not contain unnecessary information.

Contents

1 Introduction **1**

2 Related Work **6**

2.1 Code summarization . . . 6
2.2 Template based code summarization . . . 7
2.3 Stereotype identification based code summarization . . . 7
2.4 Pharo comment analysis . . . 8

3 Comment Generation **9**

3.1 Introduction . . . 9
3.2 Methodology . . . 11
3.2.1 RQ_1: Required information types for the generated class comment . . . 11
3.2.2 RQ_2: Approach to extract relevant data for the information types . . . 13
3.2.2.1 Identifying Method Stereotypes . . . 14
3.2.2.2 Identifying Class Stereotypes . . . 17
3.2.2.3 Extracting relevant data . . . 23
3.2.3 RQ_3: How can various information types be presented in the class comments? . 24
3.3 Results . . . 25
3.3.1 RQ_1 What kind of information is needed to create an adequate class comment in Pharo? . . . 25
3.3.2 RQ_2 How can relevant data be extracted from source code? . . . 25
3.3.2.1 Implication and Discussion . . . 28
3.3.3 RQ_3 How can various information types be presented in the class comments? . . 29
3.3.3.1 Implication and Discussion . . . 29

4 Evaluation **31**

4.1 Evaluation . . . 31
4.2 Evaluation subjects . . . 31
4.3 Structure of the evaluation . . . 32
4.4 Results . . . 36

4.4.1 Demographics 36
4.4.2 What do developers write and look for in class comments 36
4.4.3 Generated class comments 40

5 Threats to validity **45**

6 Conclusion and future work **47**

7 Anleitung zu wissenschaftlichen Arbeiten **49**
7.1 Installation instructions 49
7.2 Tool architecture 51
7.2.1 The CommentGenerator class 51
7.2.2 The Visitor classes 52
7.2.3 Extensions used in our Tool 52
7.2.3.1 The CgGenerateClassCommentCommand class 52
7.2.3.2 The ClassDescription class 52
7.3 User guide 52
7.3.1 Run the tool from the Playground 53
7.3.2 Run the tool directly from the interface 54
7.3.3 Replacing all CRC class comments with a generated class comment 55
7.4 Other functions 55
7.4.1 Extracting method stereotypes 55
7.4.2 Extracting class stereotypes 56
7.5 General usage 57
7.5.1 Finding classes and methods in the Pharo image 57

1 Introduction

Understanding the code base of a piece of software is a crucial and integral point in multiple stages of the software development and maintenance. To achieve it, developers rely mostly on its code and documentation [2]. Code comments are one of the most-used forms of documentation for code comprehension [6].

Depending on the type of language paradigm, various types of code comment can exist. For example, object-oriented programming languages support comments at package, class and method levels. Class comments help developers to get a high-level overview for classes [3] and understand complex programs [26]. Programming languages vary in how they use specific notations to declare comments in their code base [12] and how they embed different kinds of information in their comments [30, 38]. Such notations are for example the tags like, `@param` and `@author` in Java or `<summary>` or `<include>` tags in C#. As an example in Java, class comments give an overview of the high-level design of a program *e.g.*, purpose of a class, what a class does and what other classes it interacts with [26]. The Smalltalk class comment in contrast contains high-level as well as low-level implementation information, *e.g.*, the application programming interfaces (APIs) the class provides, instance variables a class has and key implementation features for a class. [29].

As on of the oldest object-oriented and dynamically-typed programming languages, Smalltalk is still widely used in various software systems worldwide. It was released in the early 80s and has had a separation of source code from class comments ever since Smalltalk-80 [13].

Pharo is an integrated development environment (IDE) that is based on the Smalltalk programming language. Just as its basis Smalltalk, Pharo is a pure object-oriented, dynamically typed and reflective programming environment. Pharo is an open-source environment under the MIT license.[1]

The documentation process in Pharo is structured as such, that majority of the code documentation is written in so called class comments. The Pharo class comment are presented in a separate pane, which holds most information about a class. A class comment represents the main source of documentation for Pharo developers. Usually class comments contain important information such as intent and responsibility of a class. Class comments are written using complete sentences, and using the personal pronoun "I" *e.g.*, `I represent a calculator....` To help give developers a common structure for writing class comments, Pharo provides a default semi-structured *class comment template* as seen in Figure 1.1.

Even though, as Rani showed, there has been a rise in frequency of classes being commented or adapted throughout the different versions of Pharo, Pharo's commenting practice is still not as consistent as desired. Many different programming languages show the same symptoms of code being commented sparsely or not at all [29]. While commenting code is undeniably one of the most important tools to understand, maintain and write code, developers do not always have the capabilities or simply do not want to take the time to write or update code comments [32]. Lack of adding or updating comments can lead to inconsistencies. Modernizing documentation often falls short, be it deliberately or by lack of time or motivation. These inconsistencies accumulate over time and result in hard to understand code which proves even harder to maintain [31]

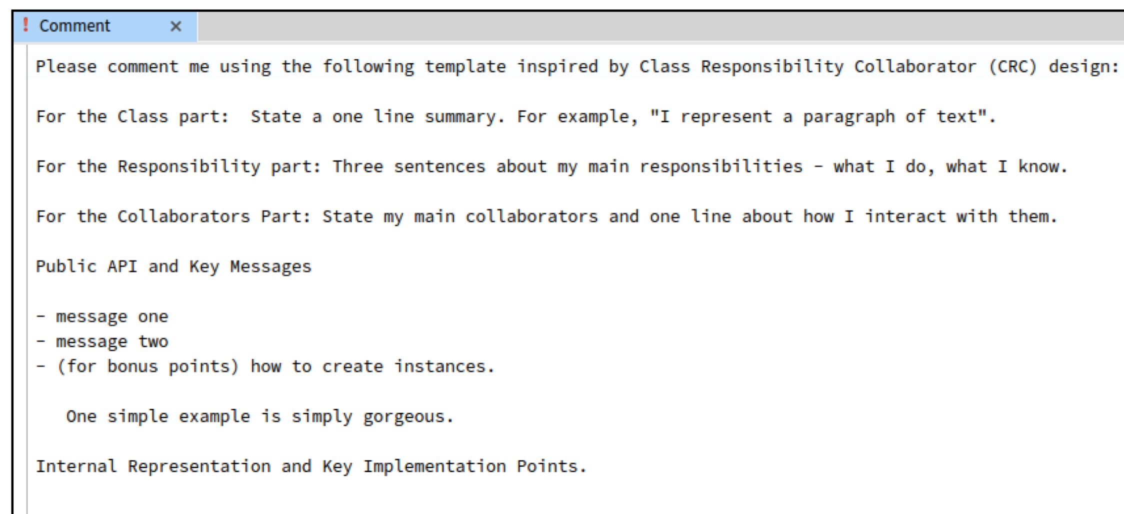

Figure 1.1: A Pharo class comment template

Code commenting holds a big responsibility in code comprehension [4, 7, 21], with the lack of code commenting or inconsistent ways of code commenting correspondingly representing a just as harmful potential. To address these concerns, many researchers started investigating the aspect of code comments.

[1] `https://pharo.org/about` accessed July 11, 2021

For instance, with how to summarize code [14], assess code comments to evaluate code quality [17], detect bugs [34] and improve software quality in general [33], the numbers of studies concerning code comments have been ever growing. In recent years there also has been an increased interest in investigating comment characteristics such as commenting habits of developers [32] and quality of comments [16, 22], with one work relating to analyzing *class comments* and *commenting practices* in the Pharo Smalltalk environment [29].

With the ever growing research into analyzing comments and how they can be written automatically, several approaches have been proposed to generate code comments automatically. These include methods, such as mining questions and answers from large code referral websites [37], mining repositories for similar code to use the code comments in said repositories for the other similar code segments [36], or processing natural language in the code itself to generate comments [18]. Automatically summarizing code [23, 40] has become a helpful tool for writing time consuming comments, reducing developer effort in writing comments and maintaining expansive code comment bases. There are many tools in programming languages, such as Javadocs[2] for Java or Doxygen[3] for C, Objective-C, C#, PHP and many more. One of the popular stereotype-based approaches was defined by Moreno *et al.* in the *Java* environment [24]. They introduce definitions for class and method stereotypes as well as heuristics for identifying these stereotypes. The stereotypes are used to classify the functionality, characteristics, and general idea of classes and methods. A stereotype that was assigned to a class or method, in that way represented a simplified intent and behavior of the class or method itself. However, their approach has not been tested on many other programming languages.

In this thesis, we replicate their approach for *Pharo*, *i.e.*, using a similar method and class stereotype dependent approach, to create a similar tool for Pharo. We adapt their stereotypes for methods and classes according to the Pharo environment, and thus generate a final class comment. This automatically generated class comment should help novice developers, by giving them an easy entry point in the structure of a class, as well as expert developers, by providing them with key information about a class. Additionally, the generated class comment should enable faster work processes for developers, by giving them a structure upon which to build their class comment.

The full process to install the tool is described in detail in section 7.1. To simply clone the repository of the tool one can use the following command:

```
1 git clone git@yogi.inf.unibe.ch:thesis-bsc-lhess
```

The goal of this study is to automatically generate class comments for Pharo classes. In the first stage, we mapped a set of heuristics proposed by Dragan *et al.* [9] to adapt method stereotypes for Pharo. We, then, mapped methods of a class to the method stereotypes. A method inside a class could be labeled as one or more of these stereotypes. In the second stage, we defined class stereotypes. Similarly to method stereotypes, we mapped and adapted heuristics proposed by Dragan *et al.* [11] to find class

[2] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html` accessed July 11, 2021

[3] `https://www.doxygen.nl/index.html` accessed July 11, 2021

stereotypes applicable to the Pharo environment. Lastly, we used class and method stereotypes to create the class comment for a target class. By automatically generating the class comments, we want to provide a framework to help experienced and novice Pharo developers. The approach can be leveraged with the help of our tool. Having a tool that generates class comments automatically can prevent problems such as inconsistencies or unavailability of comments, which in turn can prevent the long term problems associated with these issues described above. We can help developers understand their code base more easily, by giving them a rough layout of a class, as well as showing them key elements used inside the body of a class. Additionally, we can provide developers with an idea about functionality of a class without them having to scavenge through the code first. Further maintenance of a code base becomes easier for developers, as we give them a crucial tool and basis for commenting their code. We used the method proposed by Moreno *et al.* and Dragan *et al.* to create comments, which in turn leads to more comments being created in a fast and uniform way, so that it is accessible for experienced as well as novice developers [9, 11, 25].

To evaluate the generated comments, we surveyed various experienced and novice Pharo developers. We created four different online evaluation forms in which seven developers of different experience levels were asked to analyze six classes each in the Pharo environment. In the survey, developers were first asked to get familiarize with each class, and then assess the generated class comment for that class. The participants were asked to evaluate the class comments based on their adequacy, conciseness and comprehensibility. The evaluation showed some areas of our tool which still can 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.

Research Questions. To generate comment automatically and build such a tool for the Pharo environment, we formulate the following research questions:

- RQ_1**:** What kind of information is needed to create an adequate class comment in Pharo?
  Rationale: With this RQ, we want to find out what what kind of information is needed to generate a class comment in Pharo.
- RQ_2**:** How can relevant data be extracted from source code?
  Rationale: With this RQ, we investigate various approaches to generate the required information types from code.
- RQ_3**:** How can various information types be presented in the class comments?
  Rationale: With this RQ, we investigate the ways to organize the generated information in the final class comment.

In this thesis, we first review several research studies concerned with comment generation, to establish what kind of information is needed to generate a class comment in Pharo. We orient ourselves mainly towards the work of Rani *et al.* [29] and the default class comment template presented in Pharo. Once the information types to generate are captured, we use the template-based approach proposed by Moreno *et al.* and adapt the suggested heuristics to extract the needed information from the source code. We create

a process which allows us to map a class and its methods to create relevant data. Finally, we bundle the generated data into the final class comment. We structure the generated class comment to resemble the Pharo class comment template in a similar manner.

With our work we created a tool that lets us automatically generate stereotype based class comments for the Pharo environment. Extending our tool to use class stereotype heuristics allowed us to bundle various types of information based on the class stereotype, to finally generate a cohesive class comment. We present a way to display our final class comment, in that our final output resembles the CRC class comment format in intent and rough structure.

In summary, this thesis makes the following contributions:

1. Replicate the previous work by Moreno *et al.* for Pharo and adapted their heuristics to identify method and class stereotypes in the Pharo environment
2. A structure for displaying class stereotype based information
3. A tool to automatically generate class comments

The rest of this thesis is organized as follows. In chapter 2, we show the related work, in relation to our tool. In chapter 3, we describe the methodology and steps needed to create the commenting tool. This we do by showing what information is needed for the class comments (RQ1). Next, we extract the relevant data (RQ2) with subsections for the definitions of method and class stereotype and their heuristics. Additionally, in the result section of this chapter we provide a distribution of method stereotypes among 500 random classes, and a distribution for class stereotypes among 350 random classes. In the last part of this chapter we show how we generated our final class comment (RQ3) and discuss the generated output. In chapter 4 we show the process and results of the online evaluation we carried out. chapter 5 shows possible threats to the validity of our work. In chapter 6 we provide a conclusion of our work and future work to be done. Finally, chapter 7 concludes the thesis with a description and a manual of the tool we created.

2

Related Work

2.1 Code summarization

Code summarization is the process in which one generates readable summaries based on source code. It helps developers understand the functionality and character of source code more easily. Zhu *et al.* systematically analyzed 41 code summarization studies, their techniques, and the evaluation used for each study [40]. They gathered an overview of the state of the art techniques and evaluations to automatically summarize code in three main steps. In the first step, they looked at how data was being extracted. The second step focused on finding the ways natural language descriptions were generated and what kinds of summaries can be automatically generated from summarizing source code. In a third step, Zhu *et al.* analyzed what different kinds of evaluating procedures have been used to assess the results in each paper.

They found a total of five main data extraction methods, with *Information Retrieval* making up 41% of the extraction methods, and 32% of the extraction methods being based on *Machine Learning and Artificial Neural Network*. A total of 20% belonged to *Stereotype Identification*. *Natural Language Processing* (17%) and *External Description Usage* (10%) made up the rest of the extraction methods.

In their second step they describe a total for four different methods of creating summarizations. They found that *Template Based* summarization is one of the most common natural language summary generation methods, with 46% using this method to generate summaries. A total of 29% can be accounted

for by *Machine Learning* based techniques and 17% they found to be *Term* based generation of summaries. The remaining 10% were allocated to *External Description* based techniques.

Finally they found a total of four general evaluation methods, *Manual evaluation* (56%) still being the most used evaluation method to date and a fraction (10%) of the studies *not using any evaluation* at all. *Statistical Analysis* is a close second method for evaluating code summaries with (39%). The remaining roughly 30% are divided between *Gold Standard Summary* (17%) evaluations and *Extrinsic* (12%) evaluations.

Another work on code summarization by Moreno *et al.* focuses on the state of the art methods of creating automatic software summarization [25]. In this work they categorize summarization techniques into four categories: 1. *Text-To-Text*, 2. *Code-To-Text*, 3. *Code-To-Code* and 4. *Mixed Artifact Summarization*. With this paper they lay a foundation to common definitions in automatic summarization and show ways for the evolution of software summarization approaches.

2.2 Template based code summarization

Zhu *et al.* showed that template based code summarization is one of the most frequently used ways to generate natural language summaries for code bases. Template based summarization contains a predefined set of summary templates to be filled in by the target code segment and further information. [40]. This way of summarizing source code allows numerous ways for templates to be used and filled in. Dawood *et al.* [5] and Hammad *et al.* [15] for example all used processes where they simply filled in predefined templates with the needed information, like program structure information. Wang *et al.* [35] on the other hand used a natural language processing (NLP) approach to find actions, themes and secondary arguments, that they then could fill 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.

2.3 Stereotype identification based code summarization

A stereotype is a generalized idea or set of characteristics believed to represent an object or person. In software development class or method stereotypes are often used to classify the functionality, intent and behavior of classes or methods. A stereotype based summarization identifies such method or class stereotypes to choose templates accordingly. There have been several works in recent years [18, 23] using a stereotype based approach, such as Abid *et al.* who worked on using stereotypes in the automatic generation of natural language summaries for *C++ methods* [1]. Nurwidyantoro *et al.* showed that *Machine Learning* based approaches can be used to define class stereotypes [27]. We did not consider a machine learning approach in this work, as there were not enough existing data sets and we decided to choose a NLP approach instead. Moreno *et al.* defined a heuristics based approach for stereotypes, to identify code and build comments structures for the *Java environment* [24]. They argued that to automatically create

viable class comments one cannot just add all comments of the different methods together, as (i) classes contain other information than just methods – they contain 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 ; (iii) some methods may just not be relevant for the behavior of a class. Faced with these problems they generated an approach to automatically generate structured natural-language descriptions for Java classes. Their methodology was based on the conjecture that the types and distribution of methods inside a class are representative for the design and intent of a class. As such the summarization technique they created first determines the stereotypes of a class and each one of its methods to then build an overall class comment, which should encapsulate previously described internal structure and information details. Our approach, though similar in nature, sets out to create a class comment that is based on the CRC class comment template that is provided in the Pharo base image. The information we gather is based on class and method stereotypes, but this only as a pre selection to then fill in more specific information in the provided sections in the class comment template.

2.4 Pharo comment analysis

Though there has been an surge in analyzing various types of comments and summarizations for source code, the topic has scarcely been researched in the *Pharo Smalltalk environment*. Rani *et al.* paved the way by analyzing the structure of Pharo class comments and their adherence to the provided comment template [29]. They analyzed the evolution of Pharo classes and their class comments over the years and major releases of Pharo, from Pharo 1 to Pharo 7 (2008 to 2019). Gathering the various classes and class comments allowed them to understand the distribution of the frequency of classes being commented and how classes were being commented. Their work proved to be a substantial ground work, which we could use to find the kind of information that was provided by developers, and what kind of information we would have to write into our generated class comments. A major part of the information definitions was inspired by their work.

They found 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 so as to keep them up to date. Our tool was developed with the mindset to support commenting classes more frequently.

They analyzed the commenting practices in Pharo across the different versions, and reported 23 kinds of information found in class comments. They also analyzed if developers follow the class comment template. They found it suggests writing seven different types of details, such as *Intent*, *Responsibility*, *Public API*, *Example*, *Instance Variable*, *Collaborators*, and *Internal details*.

3

Comment Generation

3.1 Introduction

Code comments assist developers in understanding the associated code and in modifying the program. Developers explain various types of information in them such as the rationale behind a code fragment, examples to use the code, future tasks to do, or warnings in using the code. Previous studies highlight various types of information developers embed in code comments. [28, 39] Rani *et al.* investigated code comments in Pharo and found that class comments are one of the main sources of code documentation. Code comments present high-level design information along with low-level implementation details. Therefore, in order to generate them automatically, it is important to establish which type of information should be included in the class comment, that can reduce developers effort in writing comments and support them in having an overview of the class.

Haouari *et al.* describe code comments as being *fair+* if "they are presenting the functionalities of the related code, as well as other information" and as *fair* if "the functionality of the code being commented is adequately described" [16]. There are many relationships which can be considered inside a code base and many different ways of approaching them, to create such fair code comments. Rani *et al.* discovered 23 types of information in Pharo class comments, some of them occurring more frequently than others [29]. Moreno *et al.* on the other hand defined the information of their class summaries in four sections, namely a general description, a structural description based on the class stereotype, a behavioral description of its

methods and a list of inner classes [24]. As our aim with this study is to support developers in reducing their effort in writing comments while giving them an adequate information to understand the class, we identify the information types that facilitate in achieving this aim. Therefore, we formulate the following research question:
RQ_1: *What kind of information is needed to create an adequate class comment in Pharo?*

To select the recurrent information types, we selected the seven types of information that are suggested by the default class comment template and found to be frequent by the previous work in Pharo[29]. They also confirmed that developers follow the template in writing these information types. These seven information types include *Intent, Responsibility, Implementation Points, Public APIs, Examples, Instance Variables,* and *Collaborators*. Once the kind of information we needed to extract had been defined, the question of how to find these types of information in the code. Generating information out of source code can be done in various different ways, as seen in section: ”2.1 Code summarization”.

There have been various types of approaches used to extract information from source code, such as *Information Retrieval*, *Machine Learning and Artificial Neural Networks* and *Stereotype identification* [40]. All of these have their individual advantages and disadvantages. For instance, Information retrieval proved to be a too extensive effort for creating an *automatic* approach to generating comments [40]. Machine learning or Artificial Neural Network approaches would have been more agreeable with *automation* of our desired comment generation, but they often require a huge amount of pre labeled data sets. Due to the lack of such data sets in Pharo, we decided not to pursue this direction for now. This lead us to the following research question::
RQ_2: *How can relevant data be extracted from source code?*

Another very popular and effective technique used in this direction is the **stereotype based** approach, shown in “2.3 Stereotype identification based code summarization”. It has been successfully used in software engineering to identify indicators of code smells [8], adding comprehension to unit test cases with stereotype-based tagging [20] or to create signature descriptions for software systems based on method stereotype distributions [10]. Based on the effectiveness of this technique and previous work, like the ones done by Dragan *et al.* or Moreno *et al.*, we decided to use a stereotype based approach as well [9, 11, 24].

Having defined the information we want to collect, and a method to generate said information, the final step of our work consisted of creating a coherent class comment that presents all this information. With this in mind we formulate our third research question:
RQ_3: *How can various information types be presented in the class comments?*

Zhu *et al.* analyzed various studies related to comment and code generation [40]. We chose to analyze this set of papers, as well as the predefined class comment template provided by Pharo itself. Based on that we decided that our final output should closely resemble the template or at least provide us with a similar variety of information as the Pharo template provides. As mentioned above, we focused on the seven frequent information types defined by Rani *et al.* in the context of Pharo [29].

3.2 Methodology

3.2.1 RQ_1: Required information types for the generated class comment

Rani *et al.* showed frequent information types found in class comments. They found that the most recent Pharo class comment template suggests writing seven different types of details, namely *Intent*, *Responsibility*, *Public API*, *Example*, *Instance Variable*, *Collaborators* and *Internal details*. However, not all of the data for each information type can be generated easily in an automatic fashion. For instance, some of the information types as described by Rani *et al.*, such as *Intent* and *Responsibility*, which are rather crucial for class comments, are implicit by nature. This means the information type does not have an explicit header or specific common keyword, or information is not separated by any formatting structure (space, special symbols). Thus, it makes it hard to identify and extract some of these information types automatically. In the following paragraph we describe the challenges faced in attempt to generate each information type:

Intent of a class describes the purpose of the target class. In Pharo usually Intent is described by developers in the first line of a comment using the pattern `I represent ...` *Responsibility* answers the questions “what do I know?” and “what do I do?” and can mostly be found combined in the same sentence as the intent of the class. Both of these information types were found to be the most prevalent information type in comments according to Rani *et al.*. [29]

The role of such information types is quite crucial and important but also immensely difficult to create by only parsing source code. Our approach thus tried to focus on these implicit information types as well. This means we tried to declare information types such as *Intent* and *Responsibility* in a broader and more implicit way as well. These kinds of information types would be represented through the descriptions of their class stereotypes in the class comment. For example a class that was assigned the *Controller* stereotype, will be controlling some kind of data flow and be made up of many controller methods. Additionally, we combined this with a list of relevant keywords, to give a broad sense of what the *Intent* and *Responsibility* of a class are. *Internal details* or *Implementation Points*, refer to the internal representation of the objects, particular implementation logic, conditions about the object state, and settings important to understand the class. For *internal details* we also adapted a broader approach, by displaying a general overview for the internal details, without meticulously displaying every facet of them.

For less implicit types of information such as *Public API*, *Examples*, *Instance Variables* and *Collaborators*, the information to be gathered was more straight forward and does not require as much consideration in what more information we wanted to find for each information type. This is because less implicit (or explicit) information types can be found in a quite explicit manner (due to the recurrent patterns in writing them), and thus does not require the information to be put together from other data.

Public APIs are the key methods and public APIs (Application Programming Interfaces) of the target class. In our work these have been separated in to two main categories, specifically: *internal* and *external* method calls. Internal method calls are calls to methods from the target class itself to methods defined

inside the target class. For example a method that filters a String needed in the target class. The helper method that filters this string is defined in the target class and is used by the target class, as such it is used internally. External method calls are calls from different classes to methods defined inside the target class, such as getter or setter methods. For example the setter method of the target class, that is used by another class to set the String from the previous example, for the target class. Such a method is defined as being used externally. This allows us to better display how a class functions internally in comparison to how it is used by other classes *i.e.*, a class' functionality externally. For example, the methods used in a class to access its instance variables, *i.e.*, getter and setter methods.

Examples are simple code fragments, showing how the source code or class is to be used and can be instantiated. This helps give readers a better overview of the usage of a class.

Instance Variables being the variables generated whenever a new instance of the target class is created, and which are available throughout the instance of said class, though Instance variables in Smalltalk are private to the instance itself.[1]

With *Collaborators* being slightly more implicit, we decided to focus on simple relations between the target class and other classes. *Collaborators* for a target class as such have been decidedly separated in two categories, namely classes that *use* the target class and classes that *are used* by the target class. As an example one can look at the class *OrderedCollection* in the Pharo base image. It is used by a total of 974 different classes for its functionality, but only uses 5 different classes, including the class *Array* to create its functionality.

Implementations of how these information types have been found and displayed in the class comment are further discussed in detail in subsection 3.2.2 and subsection 3.2.3.

[1] `http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html` accessed May 29, 2021

3.2.2 RQ_2: Approach to extract relevant data for the information types

With the stereotype based approach, we analyzed source code of a class, by using certain predefined heuristics to define stereotypes for methods inside a class. For instance, if a method retrieves an instance variable of a class it was assigned the *Getter* method stereotype. Based on the identified stereotypes of each method, a class stereotype is decided by aggregating all method stereotypes. The list of various method stereotypes can be seen in Table 3.1. The according list of class stereotypes can be found in Table 3.3. This aggregation of method stereotypes was done by selecting the most used method stereotypes and the logic visible in section: 3.2.2.2. The logic defined in this section is used to assign class stereotypes according to the method stereotypes found in a class. Thus a class that has many *Getter* methods, and additionally only *Setter* methods, is assigned the *Data* class stereotype. Once we generated the class stereotypes for a class, we could extract the corresponding information that was important for a class stereotype and display this information in the class comment.

Once the information types are identified and stereotypes are assigned to a class and its methods, we can proceed to extract the relevant information from source code itself. As such our stereotype based approach worked in three stages:

1. Identifying and assigning the method stereotypes for the a method of the target class
2. Identifying and assigning class stereotypes
3. Extracting relevant data for the corresponding class stereotypes

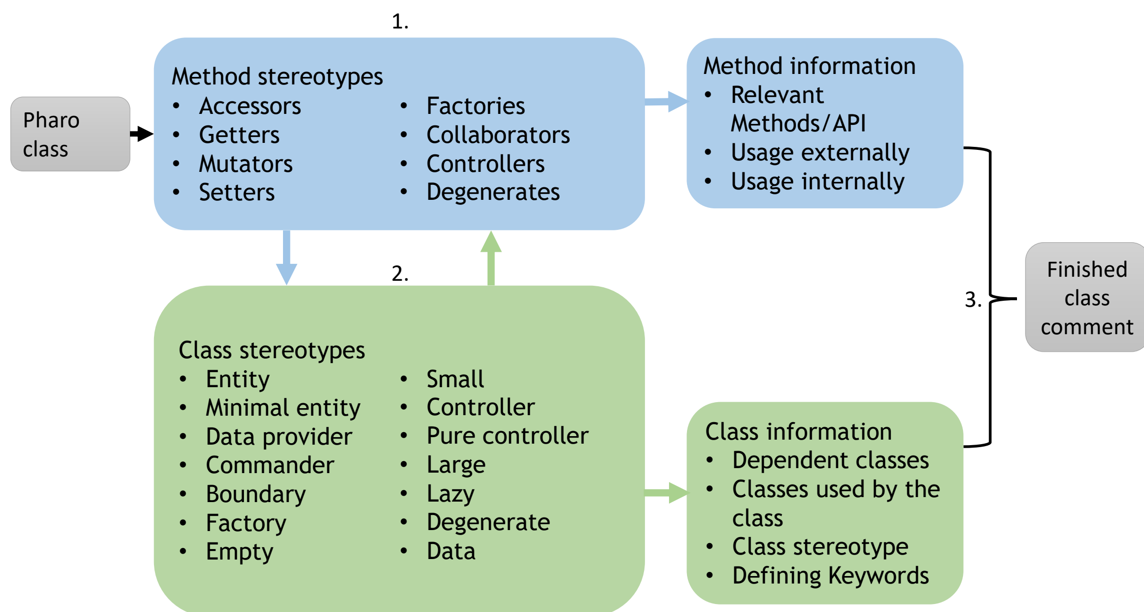

Figure 3.1: Process of generating a class comment

3.2.2.1 Identifying Method Stereotypes

As previously described, the first step in finding a class stereotype is based on finding different stereotypes for the methods within a class. There are numerous approaches in which one can separate or define method stereotypes, differentiating between minuscule differences and programming language specific distinctions or declaring them in broad strokes, which has been described more in depth in section: ”2.3 Stereotype identification based code summarization”. Working with Dragan *et al.* [9] as an inspiration allowed us to find a common ground between the two extremes, and declare solid method stereotypes for the Pharo environment. We used a total of *five method stereotypes* and three *sub stereotypes*, which represent the main functionality of a method. Similarly to Dragan *et al.*'s work we worked with the five main method stereotypes made up of Accesor, Mutator, Collaborator, Creator (also commonly referred to as Factory) and Degenerate methods, as well as the three sub categories which they defined as Getter, Setter and Controller methods [9]. Other method stereotypes mentioned by Dragan *et al.* were discarded as they were not applicable to Pharo, such as void-accessor methods which return void, a value that is not available in Pharo We discarded a total of three sub stereotypes, that they used to bundle specific functionality of methods used in the Java environment, namely void-accessors, copy-constructors and destructors.

Method stereotype definitions. Each method stereotype has specific characteristics. In this paragraph we show the rationale and responsibilities associated with the eight method stereotypes, on which we based our approach. These definitions represent the ways the method stereotypes have been set and the ways in which they were used. To help better understand the different stereotypes we provide some examples for each of the definitions.

Table 3.1: Description and Examples of method stereotypes

Method Stereotype	**Description**	**Example Method**
Accessor	A method that accesses or retrieves data. It can return boolean or primitive values.	*#index* in the *RSShape* class. It retrieves the index value, without mutating it.
Getter	A sub stereotype of an Accessor method. It can **only return data members** *i.e.*, Instance variables. As such a getter method only accesses a single value, which has to be an instance variable for our purposes. The small distinction between accessing different value types and only instance variables is needed in a later stage for differentiation in the class stereotyping process	*#parent* in the *RSShape* class. It functions the same as an accessor, but returns an instance variable.
Mutator	A method that controls changes to variables. It also includes methods that perform complex changes to the state of an object.	*#noPaint* in the *RSShape* class. It changes the variable paint to nil.
Setter	A sub stereotype of a Mutator method. It allows the setting *i.e.*, changing, of only instance variables.	*#path* in the *RSShape* class. It changes the instance variable path to a specific value passed to it.
Factory	A method that provides an interface to create objects, in object oriented programming (OOP). As "everything is an object" in Pharo, we defined Factory methods, as methods that instantiate Objects and return these objects.	*#gtCanvasForInspector* in the *RSShape* class. Creates a new RSCanvas object adds a copy of *self* to it and then returns the created RSCanvas.
Collaborator	A method that represents the collaboration between multiple objects. This includes working with other objects by passing them parameters or changing the state of other objects.	*#deepShapeFromeModel:result:* in the *RSShape* class. It compares *self* with a different object and adds *self* to its result value if they are the same.
Controller	a sub stereotype of a Collaborator method. It controls changes in external objects, but does not change the state of the target class object itself.	*#deepShapeFromeModel:result:* in the *RSShape* class. Takes an event or eventClass and changes its canvas and shape to *self* canvas and *self*
Degenerate	A method that is neither an *Accessor*, *Mutator*, *Factory* nor *Collaborator* method.	*#labled* in the *RSShape* class. As it does not have any functionality it doesn't respond to any of the other method stereotypes.

Rules for method stereotype identification. To find out if a method was assigned a certain method stereotype, every method had to be reviewed individually for its functionality and if its actions represented

one or multiple of the method stereotypes. We achieved this by finding each method of a class and then iterating over each method's source code. We did so by using certain heuristics for each method stereotype as described in the following section. Each method of a class is mapped to the method stereotype. To identify each method stereotype, Dragan *et al.* defined several heuristics, which we adapted to be usable in the Pharo environment [9]. The heuristics defined by Dragan *et al.* had to be adapted in various ways to be applicable to the Pharo environment. Many of them had to be simplified and especially be changed in such ways that they allowed for Pharo specific handling of functionality. Especially specific return and method types used in the Java environment had to be changed to fit the same functionality in Pharo. For example there are no primitive types as in the Java environment, so the idea of primitives had to be adapted for the Pharo environment.

Table 3.2: Heuristics to identify method stereotypes in Pharo

Method Stereotype	**Heuristics**
Accessor	• The method returns an instance variable or primitive
Getter	• The method must be an Accessor already • The method returns only instance variables
Mutator	• The method changes the state of an object • There are no return types or only boolean returns
Setter	• The method must be a Mutator already • The method changes only the state of an instance variable
Factory	• The method returns an object that it created
Collaborator	• At least one variable that is manipulated or passed in the method has to be an object • The object being handled cannot be a primitive
Controller	• The method only creates or manipulates *external* objects *i.e.*, objects that are created or manipulated are not of the same class as the target class
Degenerate	• None of the other method stereotypes are fulfilled

Finding the information inside the source code of a class entailed multiple steps. The main way we

mapped and monitored source code was by working with abstract syntax trees (AST). AST encodes the syntax as a cascading tree structure, as seen in Figure 3.2 , which allowed us to visit different nodes in compiled source code.

```
RBProgramNode
    RBDoItNode
    RBMethodNode
    RBReturnNode
    RBSequenceNode
    RBValueNode
      RBArrayNode
      RBAssignmentNode
      RBBlockNode
      RBCascadeNode
      RBLiteralNode
      RBMessageNode
      RBOptimizedNode
      RBVariableNode
```

Figure 3.2: Cascading structure of different Nodes in an AST

These nodes have distinct characteristics, such as a *RBMethodNode* which displays a whole method, *RBReturnNodes* which displays return values, or *RBVariableNodes* which represents a variable in the source code. As such one can find a return variable in the *RBVariableNode* inside a *RBReturnNode*. To actually access the information in these nodes, we used a visitor design pattern based approach in which a specific type of visitor can visit a node. Using this approach, we could pinpoint the relevant portions of a method and extract the important variables and information needed to decide method stereotypes.

Once we could access the required information, the heuristics defined in the section above allowed us to simply check if a method fulfilled the prerequisites to be assigned a specific method stereotype or not. Then we generated a Dictionary representing all the methods inside a class and the method stereotypes they are assigned. Figure 3.4 presents the distributions of the method stereotypes and also shown in section 3.3.2.

3.2.2.2 Identifying Class Stereotypes

In a second step, as seen in Figure 3.1, we declared class stereotypes, based on the aggregation of the available method stereotypes. Deciding which class fulfilled which class stereotype's definition proved to be a simpler process in terms of how we accessed the necessary information. Compared to the more difficult accessing of information in a method, the declaration for a class stereotype included less profound processing of source code via ASTs or the like. As defined beforehand, our approach to defining class

stereotypes was based on the *functionality of a class i.e.*, what kind of methods could be found inside a class. Similarly to Dragan *et al.* we identified a total of 13 class stereotypes, with an additional class stereotype of our own *Empty* stereotype, which helped us define or exempt certain data structures given in Pharo [11]. The 14 class stereotypes and their descriptions can be seen in Table 3.3.

Table 3.3: Table of class stereotypes for Pharo

Class Stereotype	**Description**	**Example class**
Boundary	Communication points between other classes, as such they are the boundary between multiple classes. They have many collaborator methods but few controller and fewer factory methods. Boundary classes often are also *Data Provider* classes, when they access other class's data or *Commander* when they send data to other classes.	A good example for a Boundary class is *RSCamera* in the *Roassal3-Core* package. In contrast the *RSCanvas* class in the same package is a good representation of a class that identifies as a Boundary class, as well as a Data Provider.
Commander	Contains mainly mutator methods and tries to encapsulate behavior. A large part of the methods work to execute changes to the state of an object. This may also include changes to objects of different classes.	*RSLocation* class in the *Roassal3-Layouts* package.
Controller	Work for the most part outside of themselves. They process data from other classes and access their functionalities. Controller classes consist of a multitude of controller methods.	*RSWrapStrategy* in the *Roassal3-Interaction* as it contains many controller methods.
Data	Only contain getter and setter methods. They only store data and do not operate on the data.	*RSNiceStep* in the *Roassal3-Chart* package, which as necessary only contains getter and setter methods.
Data Provider	Consists mostly of accessor methods. It encapsulates data.	*RSChart* in the *Roassal3-Chart* package. Most of its methods are accessors.
Degenerate	Contain mostly degenerate methods. Its methods mostly do neither read nor write data or the object's state.	The *RSParallelAnimation* in the package *Roassal-Animation* is a degenerate class as most of its methods are degenerate. Additionally it matches the criteria to be declared as a data provider.
Empty	Mostly contains empty methods or no methods. For example annotations or errors.	The class *RSWorldMenu*, in the *Roassal3-Menu* package. It contains 94 lines of code but no actual methods.
Entity	Represent the entity of an object. They encapsulate data and behavior. They have a variety of Accessors and Mutators, more Collaborator methods than non-collaborator and factory are not null.	*RBProtectVariableTransformation* class in the *Refactoring2-Transformations* package.
Factory	Quite intuitive as they are made up mostly of factory methods. These kinds of classes are used to create objects.	The *RSPharoMetricsProvider* in *Roassal3-Shapes* is a simple factory class that creates an object and returns it.
Large	These classes have quite a bit of functionality. They often have many more lines of code (LOC) than other classes. They incorporate many characteristics of various different classes. They can usually be divided into more specific classes, with clearer responsibilities. We declare a class not only large if it has many LOC but also if it tries to combine functionality of multiple class stereotypes in one.	*FreeTypeFontProvider* class in the *FreeType* package. It contains 372 LOC and a multitude of all the different method stereotypes.
Lazy	A class with very little functionality. To quantify this they are classes that contain getter and setter methods and a low percentage of other methods. If the most part of methods are degenerate methods a class may be declared as a lazy class.	
Minimal Entity	Kind of an entity class, but they only contain getter, setter and mutator methods. Can often be considered as lazy classes, as they usually contain very trivial actions.	The *RSDraggableForce*. Quite representative for a Minimal Entity as it is a Subclass of an RSInteraction.
Pure Controller	These classes are quite rare. All their methods are controller and factory methods. They only work on other classes. They function as a sort of God class. A god class can occur in source code, when one class knows too much and does too much. It bundles most of a systems actions and functionality and relies on external data. The god class is one of the best known examples of so called "code smell".	
Small	Contain little methods if not even any at all. The limit in our work is up to two methods in a class.	Can be seen in *RSChartSpineDecoration* in the *Roassal3-Chart* package.

Class stereotype definitions. Table 3.3 describe the rationale and definitions we used for the individual class stereotypes. We explain the role and responsibilities of a class with the corresponding class stereotype. Additionally, we show an example or more to help better understand each class stereotype.

Heuristics for class stereotype identification. Keeping the definitions described above for various class stereotypes, we proceed to map a class to one or multiple class stereotypes.

Similar to method stereotypes, Dragan *et al.* defined various heuristics to identify a class stereotype [11]. To utilize the heuristics automatically, various metrics are defined and controlled based on preset thresholds. However, not all of the heuristics fit to the Pharo context. For example, some thresholds have been used such as frequencies, averages and standard deviations of method stereotypes as defined by Lanza *et al.* [19]. Furthermore the boundary for a majority of a method stereotype A in a class, was set to 2/3 of the method stereotypes counting towards stereotype A.

We'll be using the following notations to display the heuristics used for the automatic class stereotype identification:

- $|A\ method\ stereotype|$ describes the size of the set of a specific method stereotype in the target class. *e.g.*, $|Accessors|$ represents the size of the set of accessors in a target class.
- *Methods* represents the set of all methods in a class.
- The method stereotypes that have been used to decide class stereotypes are corresponding to the ones defined in section 3.2.2.1. In addition to those we used a further set called *NonCollaborators* which consists of all methods that are not included in *Collaborators*.
  $|NonCollaborators| = |Methods| - |Collaborators|$
- Unless specified otherwise all the points for a class stereotype have to be met, so that a class fulfills the stereotype's condition and can be labeled as that class stereotype.
- *Class Stereotypes* is the set of all stereotypes that match a class.

In order to satisfy a particular stereotype for a class, the class has to fulfill various criteria. Closely related to the work by Dragan *et al.* [11] and adapted for Pharo, we describe the criteria for each stereotype in the following.

For a class to identify as the class stereotype of ***Boundary*** the class has to fulfill the following points:

- The class needs more collaborators than noncollabortors.
  $|Collaborators| > |NonCollaborators|$
- Factory methods make up less than half of the methods.
  $|Factory| < (|Methods|/2)$
- The class has even less controller methods.
  $|Controllers| < (|Methods|/3)$

For a class to identify as the class stereotype of ***Commander*** the class has to fulfill the following points:

- The class contains at least twice as many Mutators as Accessors.
  $|Mutators| > 2\times|Accessors|$
- The class has low control over other classes. Controllers and factory methods combined can only be less than half the size of Mutators.
  $|Mutators| > 2\times(|Controllers| + |Factory|)$

For a class to identify as the class stereotype of ***Controller*** the class has to fulfill the following points:

- Controller and Factory methods make up the majority of the methods.
  $|Controllers| + |Factory| > 2/3 \times|Accessors|$
- There are also methods available that do not only work on external objects *i.e.*, Accessors **OR** Mutators are available.
  $|Accessors| \neq 0 \; || \; |Mutators| \neq 0$

For a class to identify as the class stereotype of ***Data*** the class has to fulfill the following point:

- The class contains only Getter and/or Setter methods.
  $(|Getters| + |Setters|) \neq 0$ **&&** $|Methods| - (|Getters| + |Setters|) = 0$

For a class to identify as the class stereotype of ***Data Provider*** the class has to fulfill the following points:

- The class contains at least twice as many Accesors as Mutators
  $|Accessors| > 2\times|Mutators|$
- The class has low control over other classes. Controllers and factory methods combined can only be less than half the size of Accessors.
  $|Accessors| > 2\times(|Controllers| + |Factory|)$

For a class to identify as the class stereotype of ***Degenerate*** the class has to fulfill the following points:

- It can not be Empty.
  $|Methods| \neq 0$
- The majority of methods are Degenerate methods.
  $|Degenerates| > 2/3\times|Methods|$

For a class to identify as the class stereotype of ***Empty*** the class has to fulfill the following points:

- A class contains no methods **OR** the methods are empty.
  $|Methods| = 0$

For a class to identify as the class stereotype of ***Entity*** the class has to fulfill the following points:

- Entity classes have to have other Accessors than Getters.
  $(|Accessors|$ - $|Getters|) \neq 0$
- Entity classes have to have other Mutators than Setters.
  $(|Mutators|$ - $|Setters|) \neq 0$
- There are twice as many Collaborators than NonCollaborators.
  $(|Collaborators| \div |NonCollaborators|) = 2 \pm 0.1$ (This equals a margin of 5%)
- No Controller methods are available.
  $|Controllers| = 0$

For a class to identify as the class stereotype of ***Factory*** the class has to fulfill the following point:

- The majority of methods are Factory methods.
  $|Factory| > 2/3 \times |Methods|$

For a class to identify as the class stereotype of ***Large*** the class has to fulfill the following points:

- Neither Accessors, Mutators, Factory nor Controllers can be empty.
  $|Accessors| \neq 0$ **&&** $|Mutators| \neq 0$ **&&** $|Factory| \neq 0$ **&&** $|Controllers| \neq 0$
- Accessors and Mutators are a big part of the methods but not the majority
  $1/5 \times |Methods| < |Accessors| + |Mutators| < 2/3 \times |Methods|$
- **AND** Factory and Controllers are a big part of the methods but not the majority.
  && $1/5 \times |Methods| < |Factory| + |Controllers| < 2/3 \times |Methods|$

For a class to identify as the class stereotype of ***Lazy*** the class has to fulfill the following points:

- The class needs to either have Getter or Setter methods.
  $(|Getters| + |Setters|) \neq 0$
- At least $1/3$ of the methods are Degenerate.
  $(|Degenerate| > 1/3 \times |Methods|$
- There are relatively little method stereotypes, other than Getter, Setter and Degenerate.
  $|Methods|$ - $(|Getters| + |Setters| + |Degenerate|) <= 1/5$

For a class to identify as the class stereotype of ***Minimal Entity*** the class has to fulfill the following points:

- The only method stereotypes available are Getter, Setter and Mutators.
  $|Methods|$ - $(|Getters| + |Setters| + |Mutators|) = 0$
- There are twice as many Collaborators than NonCollaborators.
  $(|Collaborators| \div |NonCollaborators|) = 2 \pm 0.1$ (This equals a margin of 5%)

For a class to identify as the class stereotype of ***Pure Controller*** the class has to fulfill the following points:

- Controllers needs at least one method.
  $|Controllers| \neq 0$
- Controllers and Creators together can not be empty.
  $|Controllers|\ |Creators| \neq 0$
- No Accessors, Mutators or Collaborators are available.
  $(|Accessor| + |Mutators| + |Controllers|) = 0$

For a class to identify as the class stereotype of ***Small*** the class has to fulfill the following point:

- The class contains less than 3 methods in total.
  $|Methods| < 3$

3.2.2.3 Extracting relevant data

Lastly, having assigned stereotypes to the methods and finding stereotypes for the target class, we extracted the relevant information based on the class stereotypes. Each class stereotype has methods that are more relevant or less relevant for its architecture. With that in mind relevant method stereotypes are the method stereotypes, which are essential for the definition of a specific class stereotype. For example the relevant information in a *Data Provider* class will mostly be based in its *Accessor* and *Getter* methods, as these comprise the class stereotypes root characteristics. Thus *Accessor* and *Getter* are the relevant method stereotypes for a Data Provider class. As such we only selected methods, that had been tagged with the relevant method stereotypes, discarding the rest. If a class contained more than one class stereotype, all methods relevant for each stereotype would be collected in the relevant set of methods. Extending our previous example a class receives the stereotypes *Data Provider* and *Degenerate*, then the *Accessor* and *Getter* methods as well as the *Degenerate* methods of a class would be added to its pool of relevant methods.

Once the relevant methods for a class in accordance with its stereotype have been found, we ranked them and displayed the five most used methods per class. We restricted the output to the five most used methods, so as not to clutter up the class comment and display only the most relevant methods. This we did by separating methods into internal and external usage of a class' methods, internal meaning used by the class itself, external meaning used by other classes. By comparing how often a method is called internally we could rank the methods accordingly to their relative importance and display them in a more manageable way. The same strategy was used for calls to the methods from objects which do are created from other classes than the target class.

3.2.3 RQ_3: How can various information types be presented in the class comments?

Based on the information we decided on in section 3.2.1, we looked at the seven information types that Rani *et al.* found to be present in the class comment template of Pharo. As a reminder the class comment template contains the information types *Intent*, *Responsibility*, *Public API*, *Example*, *Instance Variable*, *Collaborators* and *Internal details*. Now with the stereotypes we gathered in section 3.2.2, we could narrow down the abundant information inside a class to display only the stereotype relevant information. Using the CRC comment template as a basis we structured our generated output in a similar manner. A final automatically generated class comment can be seen in the figure below.

```
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 :
- model
- 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
```

Figure 3.3: Automatically generated class comment for the class RSShape

At the top of the document we declare the name of the target class. Following the name of the target class, we display the different class stereotypes a class is assigned. Additionally under each stereotype there is a small description of said stereotype, to help understand the rough structure and intent of what this class represents in a very broad sense.

After the class stereotypes there is a list of up to five of the classes that are most used by the target class, and a note of how many other classes the target class uses, if there are more than five.

Following the used classes, we have a list of up to five classes that most often use the target class, and a note of how many other classes the target class is used by, if there are more than five.

The following section displays all the relevant public methods of the target class in respect to its class stereotype. These are separated into two categories and ordered by how often they are called:

- External usage of a method *i.e.*, other classes that call this method
- Internal usage of a method *i.e.*, the target class itself calls the method

Then there is a list of the instance variables of the class.

And for a final addition we created a list of the most common found keywords in a target class. This allows us to get a general feeling and understanding of the language used in the source code of a class, and can give a quick impression of the structure of a class.

3.3 Results

3.3.1 RQ_1 What kind of information is needed to create an adequate class comment in Pharo?

Considering the importance of what makes a *fair* or *fair+* code comment according to Haouari *et al.* and the fundamental structures of the Pharo commenting style, we focus on the few prevalent information types which are also suggested by the Pharo class comment template [16]. This means the types of information we want to gather include: *Intent*, *Responsibility*, *Internal Details or Implementation Points*, *Public APIs*, *Examples*, *Instance Variables* and *Collaborators*.

3.3.2 RQ_2 How can relevant data be extracted from source code?

Inspired by stereotyping approaches like [9, 11, 24] we created a process, that lets us discern eight different method stereotypes as seen in Table 3.1. Working with these method stereotypes, and the heuristics that we defined in section 3.2.2.2, we could expand our process and find the relevant class stereotypes as seen in Table 3.3. These two extensive processes allowed us to understand the class structures and target class functions in a more simple but also in depth way. Knowing a target class' stereotype we then could define what information we wanted to extract and find, as the class and method stereotyping allowed us to see which functionality appeared to be relevant for a class.

Distributions of stereotypes. In this section we display method and class stereotype distributions across the systems. We took a total of 500 random classes from the Pharo base image and analyzed their methods distribution and correspondingly the class stereotype distributions for 350 random classes.

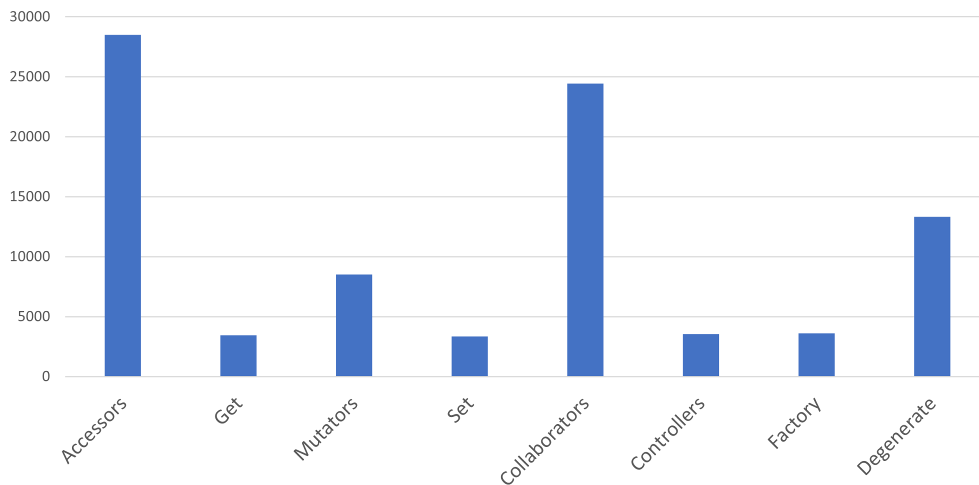

Figure 3.4: Distribution of method stereotypes for 500 random classes

Table 3.4: Summary of method stereotype distribution

Method Stereotype	Occurences	%
Accessor	28481	32.12
Getter	3438	3.88
Mutator	8510	9.60
Setter	3345	3.77
Factory	3592	4.05
Collaborator	24446	27.57
Controller	3544	4.00
Degenerate	13314	15.01

The first graph, seen in Figure 3.4, shows the distribution of method stereotypes for 500 random classes.[2] As one can see, methods most often are declared as accessor methods, closely followed by the collaborator stereotype. Almost half the accessor methods are represented by the degenerate stereotype, meaning there are half as many methods that do not fulfill any of the other stereotypes as there are methods that fulfill the prerequisites for being accesor methods. Mutator methods make up roughly a quarter of the sum of accessor methods, with about a quarter of the total mutator methods being more specific Setter methods. Compared to that Getter methods only represent about a tenth of all accessor methods. Factory

[2]Folder "RP-Automatic-comment-generation/Results/RQ2" in the Replication package

and controller methods similarly to the getter and setter methods are declared about ten times less in relation to accessor methods.

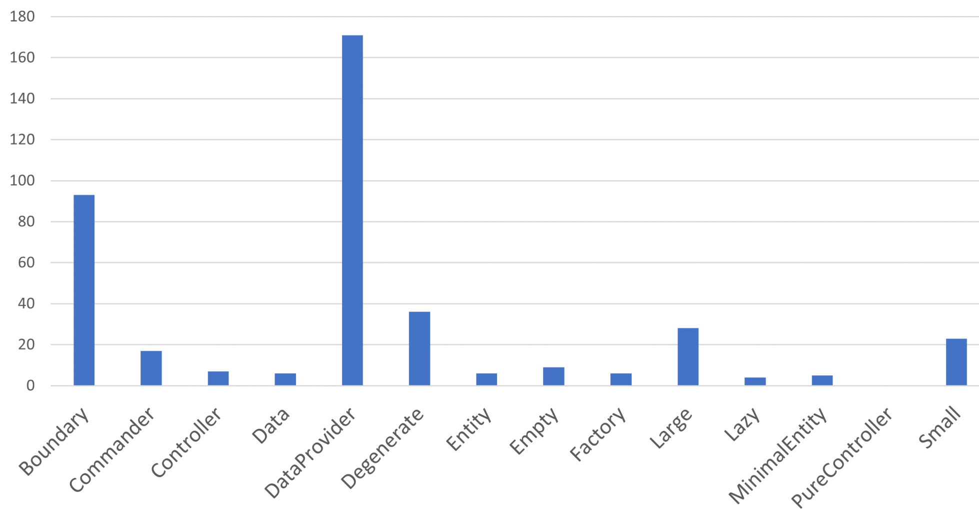

Figure 3.5: Distribution of Class stereotypes for 350 random classes

Table 3.5: Summary of class stereotype distribution

Class Stereotype	**Occurences**	**%**
Boundary	93	22.63
Commander	17	4.14
Controller	7	1.70
Data	6	1.46
Data Provider	171	41.61
Degenerate	36	8.76
Entity	6	1.46
Empty	9	2.19
Factory	6	1.46
Large	28	6.81
Lazy	4	0.97
Minimal Entity	5	1.21
Pure Controller	0	0.00
Small	23	5.60

In accordance with the method stereotype distribution one can see in Figure 3.5 that the Data Provider stereotype is declared the most often with an incredible frequency of more than two out of five classes

receiving the Data Provider stereotype.[3] A little more than half as many classes are assigned the Boundary stereotype. In contrast to the method stereotype distribution, Degenerate classes only make up half as many stereotypes in the class stereotype distribution relative to their occurrence in the method stereotype distribution. Roughly seven percent of classes are declared as Large classes and a little less than 6 percent are made up of Small classes, making about an eighth of all classes either exceptionally large or small. A little more than every 25th class receives the class stereotype commander. The Empty class stereotype makes up roughly two percent of all class stereotypes declared. Many of the other class stereotypes such as Controller, Data, Entity, Factory, Lazy and Minimal Entity are not selected more than two percent of the time. Lastly we did not find any Pure Controllers in the 350 random classes used for this distribution.

3.3.2.1 Implication and Discussion

As seen in Table 3.4 and Table 3.5 above, all of the method stereotypes and almost all of the class stereotypes could be found in the Pharo base image. Even though we could find a variety of the stereotypes, there still seems to be room for improvement. Our approach made use of many method stereotype heuristics defined by Dragan *et al.* and then adapted these to function in the Pharo environment, as such there have been some method heuristics, which have been broken down or generalized to work for the Pharo environment [9]. Thus we propose revisiting method stereotypes in a manner which redefines the heuristics in Table 3.2 only in minor ways. A sensible approach would be to revisit the way Collaborator and Controller methods work. Since Collaborator and Controller methods, per Dragan *et al.*'s definition, handle objects, problems might arise in what sense we define objects, because Pharo runs on the principle that everything is an object. We propose to only reference to objects if they are not of the primitive type, such as Characters, Strings, Numbers and Booleans. With this we won't to assure that only specific objects will be investigated.

Further we propose to look for alternatives in how the information is gathered, to decide which method stereotype is applicable to a method. Though ASTs are an immensely useful tool and allow for a simple overview of a method, they also bring some disadvantages with them. Analyzing ASTs is quite heavy in calculating costs with bigger methods. Additionally it requires the handling of many specific exceptions. They have to be declared individually for most of the nodes and their substructures seen in Figure 3.2. If the AST structure does not lead to the desired variable or node there might be loss of information that might be important for the method and subsequently class stereotype declaration. We propose to maybe revisit Machine Learning and Artificial Intelligence approaches, that were discarded in the beginning of this work. They might allow for a more profound analysis of code. The more precisely we can analyze the code itself and find the relevant key variables and objects, the better we can clearly identify stereotypes.

Class stereotyping shows a bigger discrepancy in the distributions of stereotypes compared to method stereotypes, as seen in Figure 3.5. The class stereotype definitions seem to hold up overall, as there are barely eight percent of classes that only receive a degenerate class stereotype. Many of those usually receive further class stereotypes, meaning they are not only filled with unidentifiable methods. With this in

[3] Folder "RP-Automatic-comment-generation/Results/RQ2" in the Replication package

mind, we would recommend to revise the class stereotype heuristics in 3.2.2.2, to be more lax for some stereotypes. Especially the stereotypes that appear to not be selected often, merit to be adapted in a way that makes them more accessible. For example defining the majority as $> 1/2$ instead of $>2/3$. In general more fluent borders and threshold values promise to be a good candidate to relax restrictions on the class stereotype selection.

Additionally we would recommend investing further efforts in declaring more specific class stereotypes, which are represented in the Pharo environment. There are some structures such as Models, Views, Facades and many more, which might allow us to interpret the source code in a more specific way. With further class stereotypes there are more possibilities to identify classes in more specific ways, which in turn allows us to write clearer and more specific class comments.

3.3.3 RQ_3 How can various information types be presented in the class comments?

To find out if our automatically generated class comments are accurate we compared them to the information that was suggested in the CRC class comment template, to see if we wrote a similarly helpful comment as proposed by the template. Comparing our automatically generated class comment, one can quickly see that there are no explicit sections where we declare intention or responsibility in the way we described them in section 3.2.1. As we showed in that section, these types of information are extremely difficult to extract and write automatically. Because of that we try to give a broad sense in what the intention of a class might be by showing what the intentions and responsibilities of the class stereotypes assigned to it are. Public APIs are covered under the section where we display public methods. Explicit examples are not available in our generated class comments. The list of instance variables covers the corresponding section in the CRC template. The sections describing classes that the target class uses and that use the target class represent the collaborators in the class comment. As with intent and responsibility we do not have an explicit description of the internal details, though we try to emulate and reduce this information into a list of keywords. All in all we manage to express three of the information types in a explicit way and another three of the seven information types in a more indirect way, to produce a broader sense of the class.

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. The process and results of this evaluation are described in more detail in the next chapter.

3.3.3.1 Implication and Discussion

In order to support developers with our generated comments, we developed a tool in Pharo. The tool can help developers understand source code in a more simple and faster way. With that in mind our process also laid focus on not creating a tool which would take away the responsibilities of a developer to write documentation. It should aid developers in writing documentation but still oblige them to insert their own intentions and implementations in the comments. By no means should this tool be used to spare developers from writing comments, but rather encourage and enable them to write comments more easily and consistently.

As shown multiple times though, we are trying to automatically create a comment which resembles and functions in the same manner as the CRC template provided by Pharo. In the template there are a few implicit information types, which we have not been fully able to create. Though the idea of the tool we created was never to write full natural language summaries, which contain all the details of a class, there still remains a potential to improve these types of information.

Intent and responsibilities might be improved by building upon the keyword list provided and using pre generated templates for each stereotype specifically, and then finding the required keywords to fill in the blanks in said template. A different approach might rely on similar methods as described in 2.2 where Machine Learning and AI based processes try to compile a coherent class comment, or other stereotype based methods like described in 2.3. Additionally we propose to implement a few minor improvements in how the class comments are displayed, for example adding hyperlinks to classes and methods referenced in the class comment.

The missing examples proved to be more challenging than expected. As simple as it would seem to extract examples, we quickly discovered that there are a number of different ways in which to extract examples from source code. Many of those ways rely on the fact that there already have been some kind of examples written, be it as an inline comment or the instantiation of a test case. Based on this an extended analysis of the ASTs might be a possible approach to find code examples by looking for their tags, as well as finding corresponding tests that can work as examples for the usage of a class.

All in all we managed to generate a tool which provides a variety of information for developers. The extent to which a developer wants to solely rely on the generated comment lies with themselves, though we always recommend writing and reading documentation to a certain degree. This helps prevent confusion and allows for more implicit information to be adjusted more easily.

4
Evaluation

4.1 Evaluation

To assess the approach we took with our work, we analyzed the distribution of class stereotypes for 350 randomly selected classes and method stereotypes for 500 randomly selected classes as can be seen in section 3.3.2. This kind of evaluation allowed us to gather insights into the distributions and workings of the script we created. We randomly picked the classes and manually validated their generated class comments.

To gather further *qualitative* feedback, we chose a second evaluation process, in which we let various developers assess the generated class comments. This evaluation was devised to gather observations regarding the adequacy, conciseness and expressiveness of the generated class comments. We conducted this evaluation through an online questionnaire by which developers could anonymously share their opinion.

4.2 Evaluation subjects

We attempted to keep our evaluation study design as close as possible to Moreno *et al.*'s work [25]. As a form of evaluating our method, we chose to let developers of different levels of experience look at the output we generated and have them appraise certain aspects of the automatically generated class comments.

We asked a total of twelve developers with varying experience in programming within the Pharo Smalltalk environment to fill out the online-forms. Four of the developers had been working on various projects such as *Roassal* or *GToolkit* in Pharo beforehand. In addition we had two moderately experienced developers in Pharo, with the rest of the participants being made up of multiple doctoral students and multiple undergraduate and graduate students of the University Bern with various experience levels in Pharo.

We contacted various expert developers in Pharo and got response from the expert developers of the *Roassal* system. We designed one questionnaire's classes (selected randomly) specifically from the *Roassal* system. The reminder of the Pharo classes selected randomly, are sorted based on their stereotypes and their quality in representing specific class stereotypes. As such we selected 24 classes to be evaluated in total, accumulating two classes per class stereotype. We wanted each class stereotype to be evaluated by at least two different developers, and preferably by three. Sadly five of the invited developers did not complete our evaluation, so we managed to get results from seven developers in total. This led to a total of 24 classes being evaluated in 4 different evaluation forms by a total of seven developers. The various different forms[1] are stored online. The used Pharo version from which the classes where selected, was Pharo 9[2].

4.3 Structure of the evaluation

We used the google online survey tool for the evaluation. With this approach we could i) easily provide the participants with the online questionnaires, ii) let the participants fill out their forms without bias of a so called Hawthorne effect[3], iii) collect the responses of the participants in a simple manner.

The participants filled out their questionnaires independently with no time restrictions. Participants were allowed to fill out the form in whatever environment they wanted to. They could spend as much or as little time on the different classes as they wanted. The Pharo environment we recommended was the newest version of Pharo 9.

In the first section of the survey form, we gathered the background demographics information about the developers such as their experience with Pharo, or country they live in. The second section focuses on the kind of information they write and look for in a class comment. The purpose is to partly reflect on the information types we selected in RQ1. In this section, we asked them the questions depicted in Table 4.3

Each participant received a questionnaire with a total of 6 classes. The classes they were presented all represented different class stereotypes. The distribution of the classes had been done beforehand, to allow for each participants to have an even distribution of classes and class stereotypes to evaluate. Classes were distributed in a manner that each questionnaire roughly contained the same amount of methods, so that the effort to evaluate one questionnaire was not significantly bigger than another. One of the questionnaires

[1] `https://drive.google.com/drive/folders/1PQlN0qeBJPpwYFH21Dg4-7a5jm8rpGb6?usp=sharing`
[2] `https://pharo.org/download.html`
[3] A type of reactivity in which individuals modify an aspect of their behavior in response to their awareness of being observed

was specifically filled with only Roassal classes to incorporate the expertise for certain developers, which were core developers in Roassal.

As developers had many different levels of experience with the selected classes, we wanted to ensure that the developers evaluating the comments understood the class they were being asked to evaluate. In a first step 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. Following this, we asked participants to study and understand the entire class. This they could do in whatever manner they chose to. 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 found in Table 4.1, expecting no predefined answers as before.

Table 4.1: Questions to ascertain the understanding of a class

Question
How many years of experience do you have in programming the class specific domain?
What is the intent of the class?
What are the responsibilities of this class?
Who are the collaborators (other classes that this class uses in order to fulfil its responsibilities) of this class?
What are the important methods (Key messages and Public APIs) of this class?
What are the instance variables of this class?
How are objects of this class created?
Any other details you feel important for other developers to understand this class?

Once their own evaluation of the target class had been done, we presented them with the automatically generated class comment of that class, generated by our tool. We asked them various questions shown in Table 4.2 with the provided answers.

Table 4.2: 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

Table 4.3: Questions and possible answers regarding what kind of information developers write and look for

Question	Possible Answers
How often do you write class comments for your projects?	• Never • Rarely • Sometimes • Fairly Often • Always • Others... (Personal input of participant)
Which code comments do you find useful to understand a class?	• Class comments • Method comments • Inline comments in methods • Others... (Personal input of participant)
What kind of information do you look for in the class comment?	• Intent of the class • Responsibility of the class • Collaborators of the class • Example to instantiate the class • Code examples to use the class and its methods • Implementation of specific details • Warnings related to the class • Description of instance variables • Description of Key or Public APIs, their parameters and usage • Contracts of the methods or class • Dependencies of the class • Recommendations about the class • Subclasses explanation • How to extend the class • License • ToDo • Bug information related to the class • Coding Guidelines • Exceptions • Others... (Personal input of participant)
What kind of information do you write in the class comment?	• Same answers as to the previous question
Do you follow or adhere to the class comment template to write class comments?	• Never • Rarely • Sometimes • Fairly Often • Always • Others... (Personal input of participant)
Does your team follow any other style guideline to write comments like the way you should structure your comment, use English, Grammar, Tone?	• Yes • No • Others... (Personal input of participant)
What do your style guidelines cover?	• What content to write in the comments • How to structure the text • Formatting details • Grammar • Tone • Others... (Personal input of participant)

4.4 Results

4.4.1 Demographics

Our participants all hailed from a software developer or engineer background. Most of our participants were male, with just one female participant. Roughly 29% had between one and four years of experience in Smalltalk and also 29% claimed to have between one and four years of experience in Pharo. One participant stated to have between four and seven years of experience in Smalltalk, where as two claimed 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.

4.4.2 What do developers write and look for in class comments

Before we analyzed the feedback on the generated class comments we assessed the information about what the participants write and look for in the class comments. As seen in Table 4.4 most participants find class comments to be useful to understand a class. Many of the participants also find comments for methods or inside of methods helpful to understand the class. One participant claimed, that comments about the package of a class and its architecture help to understand a class. Another participant emphasized that class comments of superclasses help further understand a subclass.

Table 4.4: Results to ”Which code comments do you find useful to understand a class?”

Answer	**Percentage of participants**
Class comments	85.7%
Method comments	71.5%
Inline comments in Methods	71.5%
Package level / Architecture documentation	14.3%
Comments of superclass	14.3%

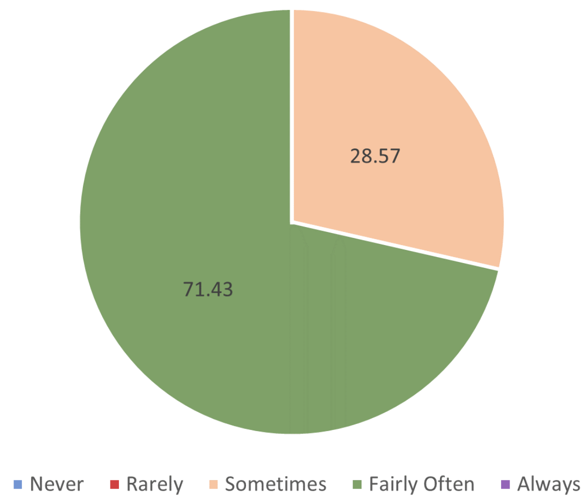

Figure 4.1: Results to "How often do you write class comments for your projects?"

In answer to how often do they do write class comments, many of the participants claimed they write class comments *Fairly often* or at least *Sometimes*. None of the participants claim they do not write any documentation at all or just rarely. That said no participant claimed to write documentation at all times, as seen in Figure 4.1.[4]

[4]Folder "RP-Automatic-comment-generation/Dataset/Online_evaluation" in the Replication package

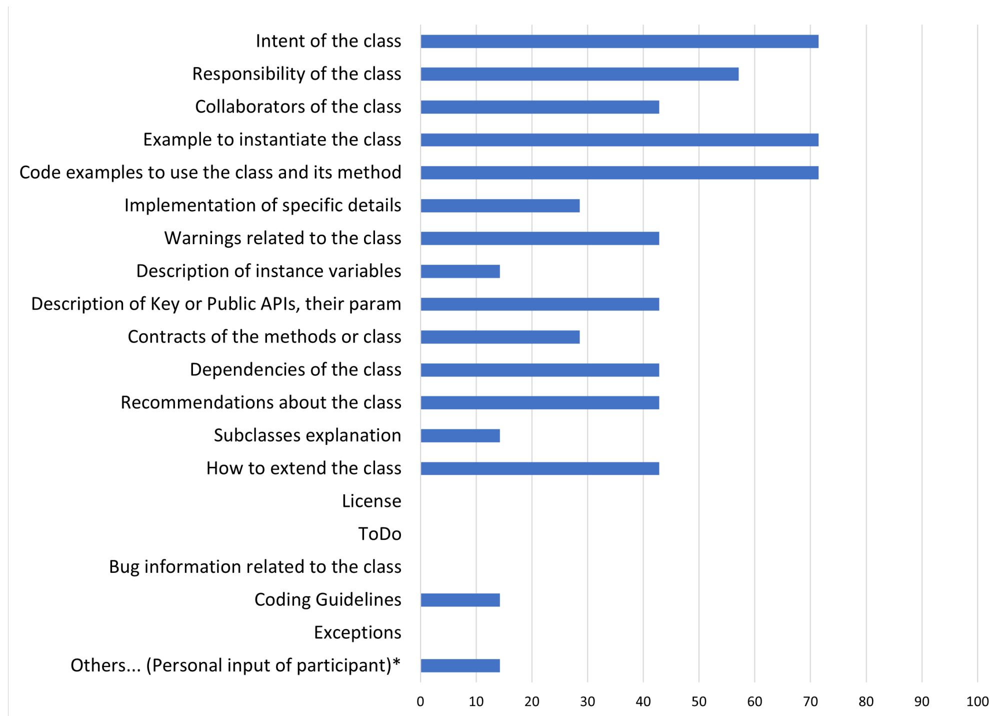

Figure 4.2: Results to ”What do you look for in class comments?”

The distribution in Figure 4.2 shows that the three major information types looked for in the class comment are the intent of a class, examples to instantiate the class and code examples to use the class and its methods.[5] As one can see most of the relevant information types mentioned by Rani *et al.* are represented by what developers look for in class comments [29]. Information types available in our class comment, such as *Responsibility*, *Collaborators*, *Intent*, *Public APIs* and *Dependencies*, can be found as well in the list of most looked for information types in a class comment. The only information type we generate that not many of our participants look for in a class comments are the *Instance variables* of a class. Only one participant claimed that they look for the *Instance variables* in a class comment. Roughly 40% of the participants stated that they look for information about warnings, dependencies, recommendations concerning the class and how to extend the class. Furthermore an approximate 30% expressed that they look for implementation examples of specific details or contracts of methods for the class. One participant indicated that they look for subclass explanations or coding guidelines. In addition one participant stated that they look for the context of a class, in the case that a class is not ’standalone’ or has a public entry point, specifically discerning the mentioned context from the simple ’collaborators’ context.

[5]Folder “RP-Automatic-comment-generation/Dataset/Online_evaluation” in the Replication package

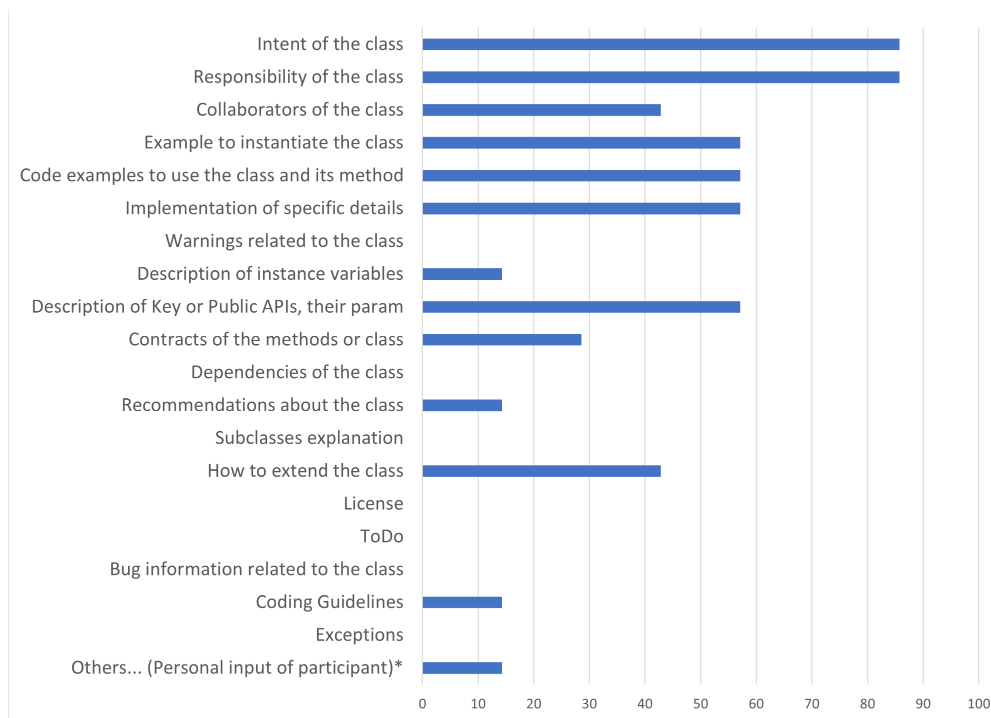

Figure 4.3: Results to "What kind of information do you write in the class comment?"

Many of the participants confirmed the main information they look for in class comments, by stating that they also write the corresponding information in class comments. In Figure 4.3 we can see that even more of the participants write information about the intent and responsibility of a class (∼86%) than look for it in a class comment (∼71% and ∼57%).[6] In addition to that also more of our participants, roughly 57%, claim to write descriptions of public APIs and their parameters or implementations of specific details, where as only 43% claimed to look for information about public APIs and barely 29% looked for implementations of specific details. Differences also appear in that less of our participants write examples to instantiate the class or examples to use the class and its methods, than are looking for this information in a class comment. The biggest difference in this can be seen, when comparing the expectancy of information about dependencies of a class and warnings related to a class, to their frequency of being written in a class comment. In both cases, roughly 43% of participants stated that they look for these kinds of information types in a class comment, with no participant claiming that they write about dependencies of a class and only one participant claiming that they write about warnings related to a class.

[6]Folder "RP-Automatic-comment-generation/Dataset/Online_evaluation" in the Replication package

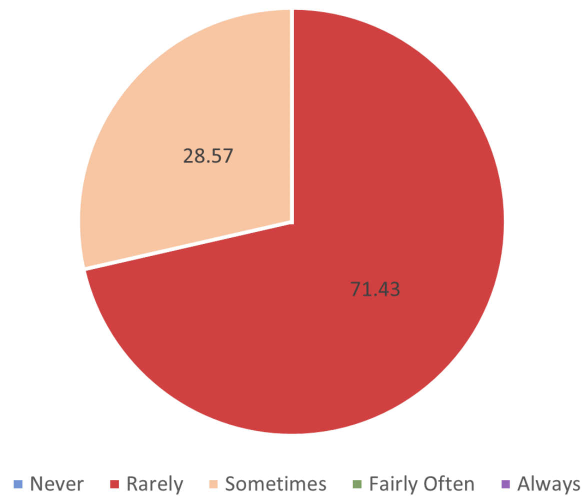

Figure 4.4: Results to ”Do you follow or adhere to the class comment template to write class comments?”

Roughly 28.5% of participants stated that they follow the class comment template sometimes, none follow it fairly often or always and the remaining 72.5% follow it rarely or never. Only one participant said they follow a different style guideline than the class comment template, the rest do not follow any other style guideline. In the case of the one participant, they declared that their style guide focused on ’what content to write in the comments’ and on what grammar to use.[7]

4.4.3 Generated class comments

This section shows the results of the evaluation concerning the classes in the online forms. It shows the bundled answers for the questions in Table 4.2, for each generated class comment. After all we received a total of 42 evaluations for the 24 class comments we generated. Before analyzing the contents of the evaluation for the generated class comments, the author analyzed the summaries written by the participants. Each summary was assessed to see if participants understood the intent of a class. The assessment of these summaries, showed that one class, namely the *SpMorphicTreeTableCellBuilder* was deprecated in the Pharo 9 image. Other than that there appeared to be one participant who had trouble understanding the *GrafPort* class and stated so in a comment as well. Though he did not say what specifically he did not understand. This helped us get a more balanced opinion of participants who understood classes and who did not. With this in mind we set out to received somewhat informed information about the classes by the participants. Thus we decided to display the generated data in two versions. The values that are displayed normally, do not include these three cases, whereas the values in brackets include the three cases. The adjustment of these three cases did not significantly change the evaluations outcome.

The results for each category, namely adequacy, conciseness and comprehensibility of our generated

[7]Folder “RP-Automatic-comment-generation/Dataset/Online_evaluation” in the Replication package

class comments can be seen in Table 4.5 to Table 4.7.[8] The specific distributions of results for each class stereotype can be seen in Table 4.8. [9] In all tables the values in brackets signify the results including the cleaned up variables, where the three cases stated before had been ignored. These values are only displayed if they are changed in comparison to before ignoring these three cases. Overall, the results are decent , with a majority of the responses being positive, though only marginally sometimes. We can say that the majority of automatically generated class comments are 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 acknowledge that the distributions for the adequacy and especially 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.

Table 4.5: Distribution of participants' responses to adequacy of generated class comments

Adequacy	
Answer	Percentage
• It is not missing any important information	20% (19%)
• It is missing some information but the missing information is not necessary to understand the class	36% (34%)
• It is missing some very important information	44% (47%)

The adequacy of the class comment's content is the most crucial point for our evaluation, as it represents if the selection process to define class stereotypes and extracting the according relevant information of our tool did a good job. Over all a total of 56% of the class comments we generated miss only some or no relevant information. As one can see in Table 4.8 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 having no or only little missing information. Classes with the stereotypes *Degenerate*, are considered by a majority of participants to be missing important information. This was to be expected, as they contain mostly empty or degenerate methods and some getters and setters. *Entity* classes were found to be missing information by three out of four participants. The participants noted that most of the missing information included the integration of the class inside the package architecture. *Large* classes proved to have missing information. This was to be expected to a certain degree, as large classes contain quite a lot of different functionality. Most participants commented on the code flow of these classes not being visible or easily understandable from the class comments.

The majority of remarks on missing information in the class comments, focused upon two points which can be used to improve our technique. The first point is that, the generated class comments describe structure, architecture and integration of the target class in too little detail. Participants remarked that the placement of a class inside the workflow of a package or script was hard to understand. They mentioned

[8] Folder "RP-Automatic-comment-generation/Dataset/Online_evaluation" in the Replication package
[9] Folder "RP-Automatic-comment-generation/Dataset/Online_evaluation" in the Replication package

especially that cases like for example tests, utility classes, abstract classes or errors should be marked as such. The second point is that, some participants remarked upon missing methods. The missing methods appear to occur because of the selection process based on the class stereotype. This leads to irrelevant methods for a class stereotype being missing in the class comments, though they seem important to the functionality of a class. For example, *Data Provider* classes tend to display mostly accessors as they are relevant for this class stereotype, though this proved to lead to some missing information about relevant methods in the class comment. This is a problem we plan to address in the further work.

The adequacy of our generated class comments is the most relevant criteria. It proved to be lacking in some areas, but it also showed that the majority of our generated class comments does not miss any or only some, important information of a class. It showed as well that if important information is missing, it proved to be one of two categories of missing information, which can certainly be improved upon in further versions.

Table 4.6: Distribution of participants' responses to conciseness of generated class comments

Conciseness	
Answer	Percentage
• It has no unnecessary information	28% (29%)
• It has some unnecessary information	26% (24%)
• It has a lot of unnecessary information	46% (47%)

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 75% to 100% no unnecessary information. The *Data* classes were evaluated to contained 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, considering that, they do not need a lot of information. Classes tagged with the *Large* stereotype were remarked by 3 out of 4 participants to contain too much information. Contrary to the way in which *Data* classes do not hold much functionality, *Large* classes contain a lot of functionality. As such participants commented on the overflow of information, especially concerning the large lists of classes that are used or do use the target class, that only showed how many of these collaborators are related to a class, but do not display all of them.

Some methods mentioned in class comments were marked as being unnecessary. This proved to come from a combination with the previous problems for adequacy. Participants remarked 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.

Table 4.7: Distribution of participants' responses to comprehensibility of generated class comments

Comprehensibility	
Answer	Percentage
• It is easy to read and understand	46% (45%)
• It is somewhat readable and understandable	41% (38%)
• It is hard to read and understand	13% (17%)

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 having no difficult to understand or hard to read structure. 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.

Most of these class stereotypes have been remarked upon in either one of the previous two sections. This shows that the heuristics for these class stereotypes have to be re-evaluated to generate more accurate class comments *i.e.*, it hold more adequate and concise class information, which in turn allows us to generate easier to understand class comments. Additionally, there might be some easier ways of presenting collaborators and APIs by creating hyperlinks to their source, to enable developers to find relevant methods and collaborators in a faster and traceable ways.

The fact that the evaluation for the comprehensibility of our class comments turned out in such a positive manner reflects that our tool can produce expressive natural language phrases that work with the Pharo commenting practices. Furthermore, we can say that the structure and way in which we present the generated data is good in a global sense.

In comparison to the work of Moreno *et al.* our tool is barely not as adequate or concise as the process they use for their work. Their tool though is only more adequate in about 13% of the cases. On the other hand they do have much less, about 43%, unnecessary information than we include in our output. There is still quite some potential for improvement, as they can be seen in comparison with their work. In contrast to that though, we can say that only 9% more of our comments were considered hard to read, and as such show a first benchmark, where we arrived at a comparable level.

Table 4.8: Distribution of participants' responses for each class stereotype

Class stereotype	**Adequacy**			**Conciseness**			**Comprehensibility**		
	No missing info	Some info is missing	A lot of info is missing	No unnecessary info	Some unnecessary info	Unnecessary info	Easy to understand	Somewhat understandable	Hard to understand
Boundary	33.33% (25.00%)	33.33% (25.00%)	33.33% (50.00%)	33.33% (50.00%)	-	66.66% (50.00%)	33.33% (50.00%)	33.33% (25.00%)	33.33% (25.00%)
Commander	-	75.00%	25.00%	-	75.00%	25.00%	50.00%	50.00%	-
Controller	66.66%	33.33%	-	66.66%	33.33%	-	100.00%		
Data	50.00% (25.00%)	-	50.00% (75.00%)	-	-	100.00%	100.00% (50.00%)	-	(50.00%)
Data Provider	-	66.66%	33.33%	33.33%	-	66.66%	33.33%	33.33%	33.33%
Degenerate	-	33.33%	66.66%	33.33%	-	66.66%	66.66%	33.33%	
Empty	50.00%	25.00%	25.00%	25.00%	50.00%	25.00%	50.00%	50.00%	-
Entity	-	25.00%	75.00%	-	25.00%	75.00%	25.00%	50.00%	25.00%
Factory	-	33.33%	66.66%	33.33%	-	66.66%	-	66.66%	33.33%
Large	-	25.00%	75.00%	-	25.00%	75.00%	-	100.00%	-
Minimal Entity	33.33%	66.66%	-	66.66%	33.33%	-	100.00%	-	-
Small	33.33%	-	66.66	33.33%	-	66.66%	33.33%	33.33%	33.33%

5
Threats to validity

In this section we discuss possible threats to the validity of this work.

Threats to internal validity are primarily concerned about the processing of how a class or method stereotype is assigned. The heuristics we adapted might just prove to be invalid in the way they return class or method stereotypes. For this purpose we had two additional developers, both with multiple years of experience in Pharo, reviewing and giving continuous feedback on the way we selected method and class stereotypes. We analyzed different classes together and adapted our code until there was a consensus for each method and class stereotype.

Similarly we decided on a template based approach, to display the generated results. The way we present this data is personally influenced, as in that it was created in accordance with the main developer's personal views and preferences and confirmed by the aiding developers. This representation of information might not represent all the information that developers would like to see, or might just lead to information being forgotten in the comment if developers do not update their comments with the specific information they want it to entail. As we found in our evaluation roughly 56% of our class comments are not missing any relevant information. As discussed in subsection 4.4.3 the major concerns about inadequacy of the class comments can be addressed by the way we use the heuristics in subsection 3.2.2.

Threats to external validity include points related to our results. One, the results for the method and class stereotype distributions in subsection 3.3.2 have been built on the choice of 500 or 350 random

classes in the Pharo base image. We believe that the large enough size of classes used should make up for any distortion effect that might happen in smaller samples.

A further threat to external validity concerns the results of our online evaluation. Firstly, we chose only two classes for each class stereotype. This might prove to not be representative enough for the specific class stereotypes. Though, we reviewed all the classes for their functionality and decided that all of them are representative enough in their own right. We also did take our classes at random from the Pharo base image, except for one questionnaire which was filled with classes in the Roassal system, to profit from the fact that we managed to have developers of Roassal evaluate our class comments. However, we selected the classes for our study randomly from the Roassal system. This might have lead to a more laborious effort needed from developers, who didn't work on Roassal and didn't receive the Roassal questionnaire. They might have had to investigate multiple classes and packages in more depth to understand the classes they were assigned. However, this helped us in getting a balanced opinion from novice and expert developers. We tried to prevent fatiguing participants with short questionnaires, where each participant only had to evaluate 6 different classes in total, which limited the mass of information they had to look at.

The participants have been chosen from different experience levels, domains and organizations. We had participants who had multiple years of experience, whereas others were novices in the Pharo environment. This might lead to slight distortions of the evaluation, where expert developers focus on different points than novices would. Though this specifically represented the goal, in that we wanted our class comments to be helpful to both ends of this spectrum, we allowed participants their own time schedule, so each participant could inform themselves about class behavior as much or as little as they wanted.

A further point corresponding to the understanding of the classes, included the fact that we allowed the participants to look at the already existing class comments. This could generate a certain expectancy from the participants, as they might be influenced by the already present class comment and thus they expected specific information to be available in the automatically generated class comment. That in turn could lead to a biased approach to the automatically generated class comment. Said behavior, was allowed, so that inexperienced developers could get a better and easier understanding of a class. We suggest prohibiting participants to look at existing class comments in future evaluations.

In addition to these points there might also be a learning effect, that took hold of participants. A participant, after having finished the first class summary evaluation, might already think ahead for certain criteria, she realized in the previous summary, and answer the second class' summary in a biased manner. Though we did not realize any explicit effect of this in the results.

6
Conclusion and future work

Understanding the code base of a software system is a crucial point in software development, as such code comments are an integral part of comprehending and maintaining the structure of code. They provide all kinds of information for developers, such as high-level intentions of code, as well as low-level details of implementation or API documentation. We gather that creating class comments automatically helps developers write and maintain code comments. In that context, we present a tool for automatic class comment generation in Pharo. Although the techniques used in this work are adapted from previous work, it is the first tool that attempt to create automated class comments for Pharo.

In the final evaluation by several developers, we discerned that our tool generates and output with 87% of the summaries being easily understandable, 56% of summaries being complete and 54% percent of summaries being concise *i.e.*, there is no or just some unnecessary information. This shows that our tool can generate easily readable and understandable class comments, which generally contain adequate information and for the most part do not contain unnecessary information.

The generated class comments are structured in such a way that they should support developers in their process of creating documentation. The proposed class summaries are not meant for any specific point of development, rather than being used as a jumping-off point for more detailed class comments. Our tool uses an adapted approach, which is completely automated and does not take any preexisting documentation into consideration.

In that context our work exposed certain points in which our tool is yet lacking. As described above, our tool does not take any preexisting documentation into consideration. For further development we plan to improve on generating class comments which work with already provided information. Additionally, as the evaluation in Chapter 4 has shown, we want to further refine the heuristics used to define class and method stereotypes as described in subsubsection 3.3.2.1, as well as finding different approaches to generating a final class comment as described in subsubsection 3.3.3.1.

All in all we foresee many ways in which our tool might be upgraded. Considering this we would lay focus on (i) adapting the used heuristics to find further diversification in method and class stereotypes, (ii) evaluate different approaches to generate class comments which contain more implicit information types, such as intent and responsibility, by using methods like Machine Learning or different approaches entirely, and (iii) try to include even more information, especially information that developers look for in the class comments, such as examples to instantiate or expand the class. In that way we want to support developers even further in writing documentation.

7
Anleitung zu wissenschaftlichen Arbeiten

In this section we will show how our tool can be installed, used and extended. The first section shows how to install the tool. After that we show the architectural design of the tool, and we give brief explanations which classes handle which action. In the following section we present multiple ways in which our tool can be used to generate various stereotypes and class comments. Finally, we show in a simple way how to access the methods and classes used as examples in Table 3.1 and Table 3.3.

7.1 Installation instructions

The installation process for our tool is quite simple. First, users have to clone the git repository. This can be done by opening a terminal in the desired folder and entering the following command:

```
git clone git@yogi.inf.unibe.ch:thesis-bsc-lhess
```

When prompted, users will have to enter their username and password, upon which the cloning process should run.

After the repository was cloned, users will have to open a Pharo image, preferably running Pharo 9.[1] Once inside the Pharo image they will have to use a tool called *Iceberg*. The Iceberg tool offers a list of

[1] https://pharo.org/download.html

the repositories loaded in a Pharo image. As a next step, import and load the project into the image. There is a button in the upper right corner labeled ‘add’, which returns a menu that allows users to ‘import from existing clone’. Users can then simply enter the path to the folder where they cloned the github repository. Once the repository has been added, users simply have to load the new package by right-clicking on the new repository and clicking on ‘load’ so the package will be loaded into their image.

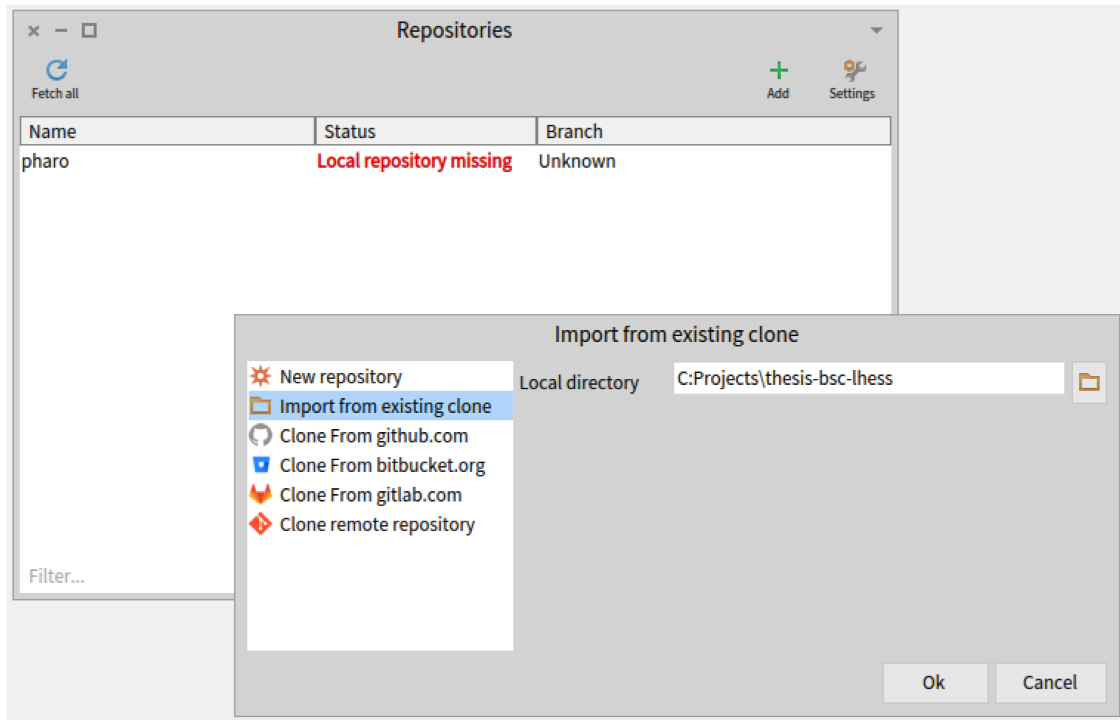

Figure 7.1: Iceberg window to load repository

7.2 Tool architecture

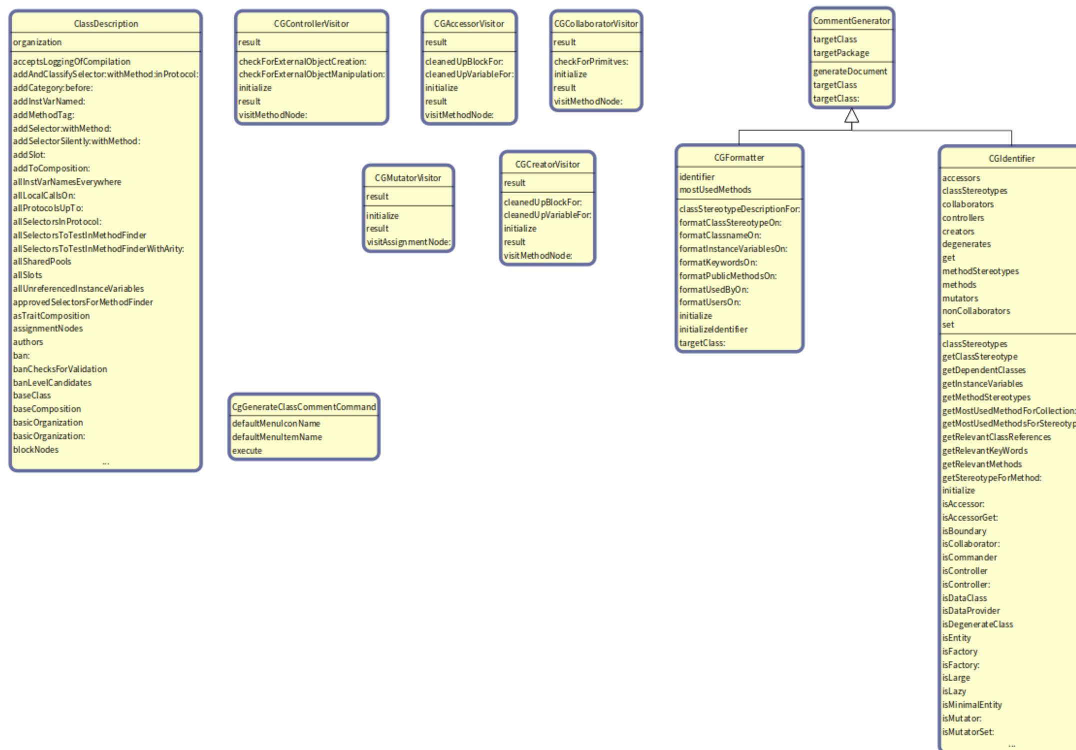

Figure 7.2: UML diagram of the tool architecture

7.2.1 The CommentGenerator class

The main class used to automatically generate a class comment is called `CommentGenerator`. It handles the input of a ‘targetClass’ and starts the process to generate a class comment by running `generateDocument`. How to run this can be seen in the subsection 7.3.1. The CommentGenerator instantiates two of its sub-classes called `CGFormatter` and `CGIdentifier`.

The class `CGFormatter` is used to format the writestream output that is returned by the Comment-Generator. Its functionality consists of organizing the generated data and styling it in such a way that we can present it as seen in Figure 3.3.

The class `CGIdentifier` contains most of the logic to identify the information we need to generate the class comment. It contains multiple methods to quantify method and class stereotype frequencies, as well as the logic to which information is used or discarded for the final comment. Additionally it also instantiates the visitors, which hold the logic to decide method stereotypes for a method.

7.2.2 The Visitor classes

The visitors we created allow us to map over the ASTs of a method. They are crucial to finding the information needed to decide if a method is tagged with a method stereotype. We have created multiple such visitors that look at different nodes of a method. The visitors always have a boolean value called ‘result‘, which represents *true* = yes the method receives the method stereotype or *false* = no the method does not. We set up visitors in the following way:

```
1 | method visitor |
2   method := targetClass >> aSymbol.
3   visitor := CGAccessorVisitor new.
4   method ast acceptVisitor: visitor.
5   visitor result
```

The important values here are ‘aSymbol‘, as it is the symbol with the method name and ‘CGAccessorVisitor‘ as it represents the specific method stereotype visitor we want to use. Only the ‘CGAccessorVisitor‘should be exchanged with another visitor, if one wants to check for another method stereotype.

7.2.3 Extensions used in our Tool

In this section we present the two classes used to extend the functionality of our tool with pre existing functionality of the Pharo base image. These classes add functionality to our tool that makes the way the tool can be used even simpler. They mainly interact with the system browser or the standard class comment template. We extended the two classes as follows.

7.2.3.1 The CgGenerateClassCommentCommand class

It is this class that allows us to create class comments from anywhere in the system browser. The `CgGenerateClassCommentCommand` class contains little functionality other than extending logic from the *Commander-Core* package, letting us assign a personalized title and icon to the right-click menu of a class in the system browser.

7.2.3.2 The ClassDescription class

The `ClassDescription` class is responsible for filling empty class comments with our automatically generated class comments per default. It contains but a single method which runs our ‘`CommentGenerator generateDocument`’ script instead of adding the CRC class comment template.

7.3 User guide

The way we created our tool, there are two different ways in which users can get a generated class comment and one additional functionality that lets the user replace all the class comments, which only hold the class comment template, with an automatically generated class comment. The methods are as follows:

1. Run the tool from the Playground
2. Run the tool directly from the interface
3. Replacing all CRC class comment templates with a generated class comment

7.3.1 Run the tool from the Playground

The instantiation and call to create a class comment are fairly easy. Once the user opens a new playground in Pharo, he or she simply has to enter the following:

```
|cg|
cg := CommentGenerator new.
cg targetClass: RSShape.
cg generateDocument.
```

The only variable a user has to change to his or her desire is the 'targetClass' value. As the name implies this is the value that decides which class the comment will be generated for. Once the user runs this, the Pharo IDE should open an inspector window with the output found in Figure 7.3.

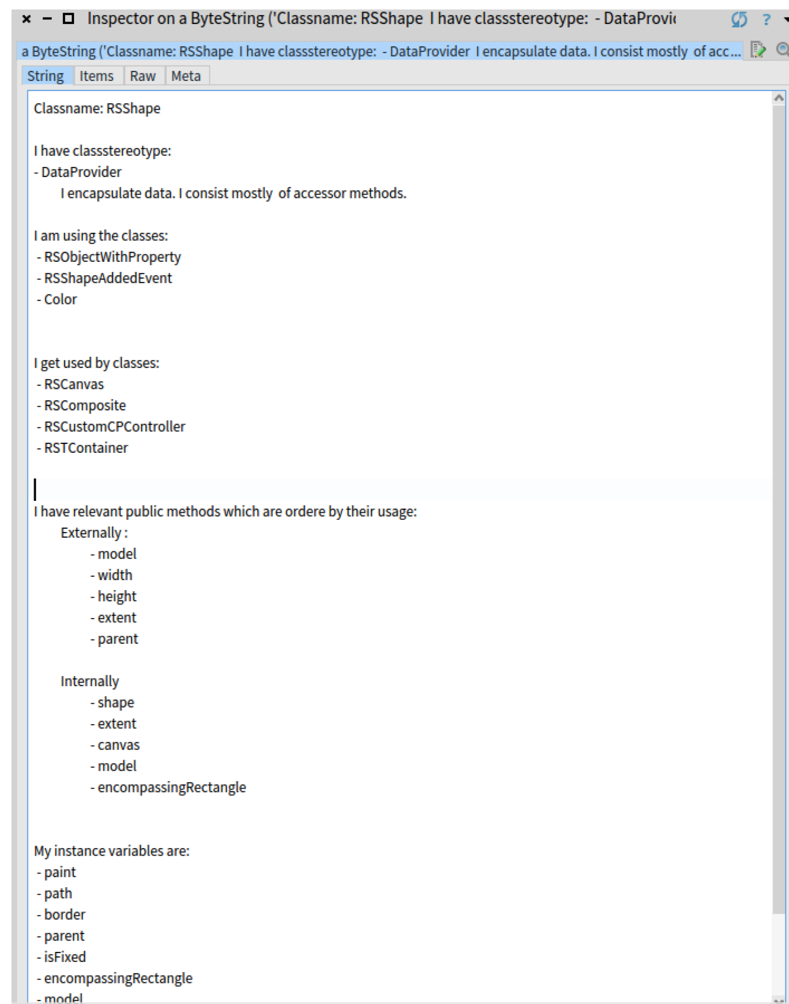

Figure 7.3: The Inspector window after generating the class comment for RSShape

7.3.2 Run the tool directly from the interface

To make it even easier for users we created a method which allows them to generate class comments directly from the system browser. For this the user simply has to search his desired target class in the system browser and right click on it. A menu will appear, displaying various actions for the class. We implemented an action called ‘Generate stereotype-based Class Comment‘ as seen in Figure 7.4. Once the user clicks on the field, the Pharo IDE should once again open an inspector window with the same output found in Figure 7.3.

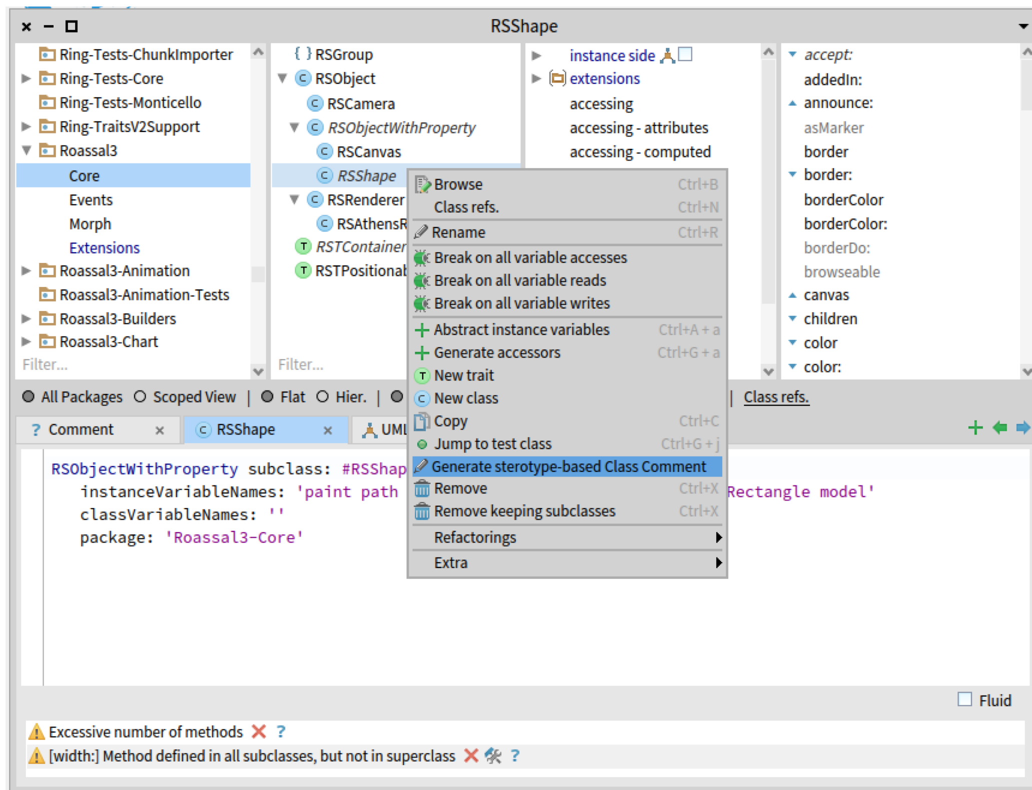

Figure 7.4: Menu accessing of ‘Generate stereotype-based Class Comment‘ in the system browser

7.3.3 Replacing all CRC class comments with a generated class comment

As a final possibility we give the option to overwrite the standard handling of empty class comments. Usually when a developer creates a class, its class comment initially is empty. The class comment template is then automatically added to the empty class comment, so that each new class contains the class comment template. We rewrote some of the base functions of the ClassDescription class, allowing us to always automatically generate a stereotype-based class comment, if there is no class comment available. In this way, classes without comments will still be flagged with a red exclamation mark, that remarks that there is no class comment available. But they will now have a generated class comment as a starting point.

7.4 Other functions

7.4.1 Extracting method stereotypes

As the process of generating a class comment includes extracting method stereotypes, developers can easily check the stereotypes found for the methods in a class. To find out the method stereotypes developers have to use the class *CGIdentifier*. The process works similarly to the process in 7.3.1. Developers can open the

Pharo Playground again and run the following:

```
1 |ci|
2 ci := CGIdentifier new.
3 ci targetClass: RSShape.
4 ci getMethodStereotypes.
```

Just as in the previous script, the ‘targetClass‘can be adjusted to the user’s desire. The Pharo IDE should open a new Inspector again displaying the method names in the left column and its method stereotypes in the right hand column as can be seen in Figure 7.5.

Inspector on a Dictionary [78 items] (#accept:->a Set('Collaborator') #addedIn:->a Set('Mutat

a Dictionary [78 items] (#accept:->a Set('Collaborator') #addedIn:->a Set('Mutator' 'Collaborator') #announce:->a Set('Controll...

Items Keys Raw Meta

Key	Value
#paint:	a Set [3 items] ('Mutator' 'MutatorSet' 'Collaborator')
#noPaint	a Set [1 item] ('Mutator')
#hasChildren	a Set [1 item] ('Accessor')
#parent	a Set [2 items] ('Accessor' 'AccessorGet')
#index	a Set [1 item] ('Accessor')
#height	a Set [1 item] ('Accessor')
#gtCanvasForInspector	a Set [2 items] ('Creator' 'Accessor')
#border	a Set [2 items] ('Accessor' 'AccessorGet')
#withBorder	a Set [1 item] ('Degenerate')
#postCopy	a Set [1 item] ('Degenerate')
#shape	a Set [1 item] ('Accessor')
#hasEventCallback	a Set [1 item] ('Accessor')
#borderColor:	a Set [1 item] ('Collaborator')
#sessionChanged	a Set [1 item] ('Degenerate')
#intersects:	a Set [2 items] ('Accessor' 'Collaborator')
#copyWithAnnouncer	a Set [1 item] ('Accessor')
#depth	a Set [1 item] ('Accessor')
#schildren	a Set [1 item] ('Accessor')
#borderColor	a Set [1 item] ('Accessor')
#shapeWithActionForPosition:	a Set [2 items] ('Accessor' 'Collaborator')
#isInACanvas	a Set [1 item] ('Accessor')
#deepShapeFromModel:result:	a Set [1 item] ('Collaborator')
#topParent	a Set [1 item] ('Accessor')
#labeled	a Set [1 item] ('Degenerate')
#canvas	a Set [1 item] ('Accessor')
#encompassingRectangle	a Set [2 items] ('Mutator' 'Accessor')
#isFixed:	a Set [3 items] ('Mutator' 'MutatorSet' 'Collaborator')
#paint	a Set [2 items] ('Accessor' 'AccessorGet')
#color	a Set [1 item] ('Accessor')
#path:	a Set [3 items] ('Mutator' 'MutatorSet' 'Collaborator')
#sparent	a Set [1 item] ('Accessor')
#path	a Set [2 items] ('Accessor' 'AccessorGet')
#paintOn:	a Set [2 items] ('Accessor' 'Collaborator')
#popup	a Set [1 item] ('Degenerate')
#addedIn:	a Set [2 items] ('Mutator' 'Collaborator')
#shapeWithAction:forPosition:	a Set [2 items] ('Accessor' 'Collaborator')
#resetPath	a Set [1 item] ('Mutator')
#isNode	a Set [1 item] ('Accessor')

Figure 7.5: Inspector output with Dictionary for all method stereotypes of RSShape

7.4.2 Extracting class stereotypes

Just as we can extract only method stereotypes, we can also only extract class stereotypes. The process functions exactly the same as the process in 7.4.1. The only thing we can adjust here is the ‘targetClass’ once again. Other than that we only have to change the method call of the CGIdentifier to return class stereotypes. The script to this looks like this:

```
|ci|
ci := CGIdentifier new.
ci targetClass: RSShape.
ci getClassStereotype.
```

This will open a new inspector with a list of the available class stereotypes.

7.5 General usage

7.5.1 Finding classes and methods in the Pharo image

To access the example methods and classes proposed in Table 3.1 and Table 3.3 one can browse the Pharo base image. Classes can be found quite easily by using the so called `Spotter` in the Pharo IDE. The Spotter can be opened by either clicking on the menu 'Tools' in the toolbar of the IDE or by simply clicking on the background and opening the 'World-menu'. It can also be opened by using the short-cut 'Shift-Enter'. Once the Spotter window pops up, one can simply enter the class they are looking for and the Spotter will open a new system browser with the desired class opened. With this, methods in specific classes can be found just as easily.

Bibliography

[1] N. J. Abid, N. Dragan, M. L. Collard, and 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)*, pages 561–565. IEEE, 2015.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. An empirical investigation on documentation usage patterns in maintenance tasks. In *2013 IEEE International Conference on Software Maintenance*, pages 210–219. IEEE, 2013.

[3] A. Cline. Testing thread. In *Agile Development in the Real World*, pages 221–252. Springer, 2015.

[4] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[5] K. A. DAWOOD, K. Y. SHARIF, and K. T. WEI. Source code analysis extractive approach to generate textual summary. *Journal of Theoretical and Applied Information Technology*, 95(21):5765–5777, 2017.

[6] S. C. B. de Souza, N. Anquetil, and 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*, pages 68–75, 2005.

[7] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. Which documentation for software maintenance? *Journal of the Brazilian Computer Society*, 12(3):31–44, 2006.

[8] M. J. Decker, C. D. Newman, N. Dragan, M. L. Collard, J. I. Maletic, and 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)*, pages 82–91. IEEE, 2018.

[9] N. Dragan, M. L. Collard, and J. I. Maletic. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 24–34. IEEE, 2006.

[10] N. Dragan, M. L. Collard, and J. I. Maletic. Using method stereotype distribution as a signature descriptor for software systems. In *2009 IEEE International Conference on Software Maintenance*, pages 567–570. IEEE, 2009.

[11] N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[12] M. Farooq, S. Khan, K. Abid, F. Ahmad, M. Naeem, M. Shafiq3a, and A. Abid. Taxonomy and design considerations for comments in programming languages: a quality perspective. *Journal of Quality and Technology Management*, 10(2), 2015.

[13] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

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

[15] M. Hammad, A. Abuljadayel, and M. Khalaf. Summarizing services of java packages. *Lecture Notes on Software Engineering*, 4(2):129, 2016.

[16] D. Haouari, H. Sahraoui, and P. Langlais. How good is your comment? a study of comments in java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 137–146. IEEE, 2011.

[17] J. H. Hayes and L. Zhao. Maintainability prediction: A regression analysis of measures of evolving systems. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 601–604. IEEE, 2005.

[18] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[19] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

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

[21] W. Lidwell, K. Holden, and J. Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[22] Y. Liu, X. Sun, and Y. Duan. Analyzing program readability based on wordnet. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–2, 2015.

[23] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2015.

[24] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE, 2013.

[25] L. Moreno and A. Marcus. Automatic software summarization: the state of the art. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 511–512. IEEE, 2017.

[26] E. Nurvitadhi, W. W. Leung, and C. Cook. Do class comments aid java program understanding? In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 1, pages T3C–T3C. IEEE, 2003.

[27] A. Nurwidyantoro, T. Ho-Quang, and M. R. Chaudron. Automated classification of class role-stereotypes via machine learning. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 79–88. 2019.

[28] L. Pascarella and A. Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE, 2017.

[29] P. Rani, S. Panichella, M. Leuenberger, M. Ghafari, and O. Nierstrasz. What do class comments tell us? an investigation of comment evolution and practices in pharo. *arXiv preprint arXiv:2005.11583*, 2020.

[30] R. Scowen and B. A. Wichmann. The definition of comments in programming languages. *Software: Practice and Experience*, 4(2):181–188, 1974.

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

[32] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*, pages 83–92. Ieee, 2013.

[33] L. Tan. Code comment analysis for improving software quality. In *The Art and Science of Analyzing Software Data*, pages 493–517. Elsevier, 2015.

[34] L. Tan, D. Yuan, and Y. Zhou. Hotcomments: how to make program comments more useful? In *HotOS*, 2007.

[35] X. Wang, L. Pollock, and 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)*, pages 205–216. IEEE, 2017.

[36] E. Wong, T. Liu, and L. Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389, 2015.

[37] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567, 2013.

[38] A. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. *ACM SIGSOFT software engineering notes*, 30(4):1–5, 2005.

[39] J. Zhang, L. Xu, and Y. Li. Classifying python code comments based on supervised learning. In *International Conference on Web Information Systems and Applications*, pages 39–47. Springer, 2018.

[40] Y. Zhu and M. Pan. Automatic code summarization: A systematic literature review, 2019.