Università
della
Svizzera
italiana

**Software
Institute**

Pattern-based Generation of Software Documentation for Android Apps

Riccardo Gabriele

June 2020

Supervised by
Prof. Dr. Gabriele Bavota

Co-Supervised by
Dr. Csaba Nagy
Dr. Emad Aghajani

Software & Data Engineering Master Thesis

Abstract

The quality of software documentation represents an issue in several software projects. Indeed, many developers do not give code documentation enough importance and, as a consequence, source code can become difficult to understand, resulting in a lot of time spent in program comprehension activities.

Many approaches have been proposed in the literature to (partially) address this problem. For example, some approaches rely on retrieval techniques to identify code snippets similar to the one to document, and reuse their documentation. However, these approaches fail when similar code snippets are not available, or their documentation is of low quality. Others exploit Deep Learning (DL) techniques to learn how to document a given piece of code. However, the effectiveness of these DL-based approaches is still very limited.

We present POET (Pattern-based dOcumentation gEneraTion), a pattern-based technique to automatically generate documentation for Android applications at different granularities. Given a code snippet, the approach attempts to document it by applying one or more patterns that match the given code. Applying a pattern (a natural language description that can contain code-related information and can be restricted by requirements) to a piece of code generates its documentation. The technique can be continuously improved through the addition of new patterns whenever the already existing ones are not enough for the context of interest.

Our evaluation shows that POET is able, in most of the cases, to generate a readable and meaningful description of the given code, and it is superior in these aspects to DL-based techniques.

To my mother and father who never
understood what I was doing, but
were always supportive

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. Dr. Gabriele Bavota for allowing me to undertake this work and for having guided and helped me throughout every thesis' step. I am also grateful to my co-supervisors Dr. Csaba Nagy and Dr. Emad Aghajani for their continuous guidance and multiple advices, along for their effort throughout the work. I would also like to thank USI and every professor who make us students love Computer Science.

A big thank you also to my friends and colleagues who I met during my studies: Michele, Lorenzo, Aron, Alessio, Camillo, Vanessa and many others, who are always available whenever I need a hand.

Many thanks also to my friends, Giuseppe Pio, Erika, Giuseppe F., Marco, Saitto, Sebastiano, Colino, Masciulli, Francesco, Michela, Stefano and many others, including the 5 A LSA class and its teachers. Particularly, thank you to Fabrizio Lapenna and Giuseppe Sambrotta who are always there and would do anything for me: their friendship and support are essential to me. Another special thank you to Dasha who has to endure me every day and is always there when I need her.

Lastly, I would like to thank my whole family, in particular my grandparents Clarita, Antonio, Antonietta and Carlo and my uncle and my aunts Antonello, Silvia and Paola.

One last thank you to my parents and my brother, Sabatino, Alessandra and Davide who always do their best for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Developers spend a considerable amount of time (~58%) in program comprehension activities [18]. The amount of time required by such activities can be reduced through the usage of code comments describing in natural language what a code snippet does [13]. However, writing good comments is not always a top priority for developers, who may focus their efforts on having a working prototype of their products as soon as possible, fixing a bug in the shortest time possible, *etc.* Additionally, it is possible for comments to be outdated (*i.e.,* when a piece of code is modified, the comment is not updated with it) or mismatched (*i.e.,* a comment does not describe what the code snippet actually does).

For this reason, many techniques have been developed to automatically generate documentation for a given code snippet. Some of them are based on retrieval techniques: They build a dataset from a code repository (*e.g.,* Stack Overflow discussions) and, given an undocumented code snippet S, exploit the repository to find a snippet similar to S in order to reuse its documentation [1, 15]. Others are based on deep learning: These approaches train a model on a large dataset composed of pairs of code snippets and their related comment, which is usually built from the mining of open source repositories. Then, the trained model is used to generate a comment or a summary from a previously unseen piece of code [2, 8, 9]. Finally, a third category of approaches uses predefined rules, stereotypes and/or patterns [3, 4, 6, 7, 10–12, 14, 17] to generate documentation. For example, LambdaDoc [3] is a rule-based tool that reads a lambda and produces a high-level documentation of it by reading its parameters and return value.

Despite the advances in this field, the above-listed approaches still suffer from major limitations: for example, most of them cannot document code snippets at different granularity levels (*i.e.,* either they can document a method, or a class). Another limitation is that every approach either reuse an already existing piece of code (this is the case of retrieval-based approaches) or they generate it from scratch, without considering the possibility of a hybrid approach. Additionally, most of them are general purpose tools (*i.e.,* supposed to work for any type of system) which we believe is unrealistic to obtain good documentation results.

1.2 Objectives and Results

In this thesis we define an approach based on patterns to automatically generate documentation for Android applications. A pattern is a predefined template that can be used to describe a given code snippet. We have put a focus on the Android environment in order to take advantage of its well-documented APIs, but we believe that our approach can be easily adapted to other types of software. To support the patterns creation, we developed a web-based solution that allows multiple annotators to collaborate in the creation of documentation patterns. Those patterns will be then used to generate documentation for previously-unseen code snippets (snippets for which we did not create any pattern) to verify the validity of our approach.

We started by mining code snippets from the method bodies of various Android applications and split them into smaller snippets using line breaks and other heuristics. The goal of this procedure is to isolate cohesive sets of instructions within the method body. These snippets have been clustered based on their similarity and, a *commenting pattern* has been defined for each cluster of snippets. Finally, we verified the effectiveness of our approach by comparing it with other state-of-the-art approaches based on code retrieval and deep learning.

The achieved results show that most of the generated documentation can describe with good accuracy what a code snippet does. Additionally, the generated comments are often readable and well written. Compared with the current state-of-the-art techniques, POET is able to generate a better documentation for almost every code snippet of our test set, with an average score of 3.3 out of 5 compared with the 2.05 score of the current state-of-the-art deep learning approach.

1.3 Structure of the Thesis

We will now describe the structure of this thesis, along with a brief description of each chapter. Following this introductory chapter, Chapter 2 discusses the related literature, focusing on approaches for the automatic documentation of code. In particular, it gives an overview of existing pattern, retrieval, deep learning, rule and stereotype based approaches.

Chapter 3 describes the implementation of the approach, along with its architecture and examples of its usage. In this chapter we will also explain most of the technical implementation details.

Chapter 4 elaborates on our choices about the design of the empirical study along with some data and instructions on how to replicate the study and run the server and the web application on any machine.

Chapter 5 and 6 focuses, respectively, on the discussion of the obtained results and the threats to validity. Finally, Chapter 7, concludes the thesis. This chapter also discusses some possible improvements and paths to follow in order to improve the approach.

Chapter 2

State of the art

This section describes the state of the art in the field of automatic documentation generation. We categorised the previous literature on the basis of the basic technique they use to generate documentation (*e.g.,* pattern-based, retrieval-based, deep learning-based). We also show some examples and discuss differences between our and their work.

2.1 Pattern-based approaches

Pattern-based approaches are used to exploit regularities in the data. In the context of this work, matching patterns in source code can help to identify code snippets implementing a given functionality that can then be automatically documented.

Buse and Weimer [4] focus on generating documentation for Java exceptions to avoid problems with security, reliability, and encapsulation, as well as to simplify software maintenance. To do that, first they analyze every method to retrieve all the possible invocations and statements where an exception can be raised. Then, using this information, they try to understand why an exception is thrown by getting the path predicate (that describes the conditions under which that path can be taken) and translating it to natural language using a small number of recursive pattern-matching translations to phrase common Java idioms more succinctly. For example: *true* becomes *always* or *x instanceof T* becomes *x is a T*.

A paper from McBurney and McMillan [10] discusses a common problem of approaches to automatically generate comments: they do not take into account the context that surrounds the method being summarized (*i.e.,* when and how a method is used). Based on this observation, they build a source code summarization technique that writes English descriptions of Java methods by analyzing how those methods are invoked. They did that by using the PageRank algorithm to discover the most important methods' calls in the analyzed method's context. The idea of using PageRank to rank methods based on their importance comes from the intuition that methods that are called many times or that are called by other "important methods" are likely to be more "important" than methods called rarely. After doing that, they use SWUM, the Software Word Usage Model, which is a technique converting Java statements into keywords and prepositional sentences. SWUM is used to extract keywords about the actions performed by the previously-found most important methods and a custom natural language generation system is then exploited to generate sentences. After having lexicalized descriptions for the method (a quick summary, the return type, its importance rank and others), they look for patterns of those descriptions and then group their phrases in sentences. An example they use in their paper is that, if a summary for a method follows another summary for a different method, then it implies that they are related, and can be connected using the preposition "for". The result is a phrase such as "skips whitespace in character streams for a method that processes xml". In the end, they use an external library, simplenlg [5], to realize complete sentences from those

phrases. While the generated summaries are good, they are still less accurate and concise than human-written ones.

2.2 Retrieval-based approaches

Retrieval-based approaches have frequently been used to generate documentation. The main idea behind them is to mine some source of information (*e.g.,* Stack Overflow snippets and their related discussions) and use it to build a knowledge base. This knowledge base is then used by an application or a software package to answer questions or perform some tasks (*e.g.,* use the Stack Overflow discussion to generate documentation for a code component similar to the snippet in the discussion).

CODES (mining sourCe cOde Descriptions from developErs diScussions), developed by Vassallo *et al.* [15], applies a "social" approach for documentation generation. Their tool is an Eclipse plugin with a knowledge base built from Stack Overflow. It adds a functionality to the IDE that allows the developer to generate the documentation of a selected method by just pressing one button. The reported results are very good in terms of precision of the generated comments (*i.e.,* the generated comments are relevant), but CODES can only describe a small percentage of the project's methods. This is due to the fact that it is unlikely to find, in developers' communications, a description of all possible methods.

Another approach, named ADANA (Automated Documentation of ANdroid Apps) [1], documents code snippets by exploiting a knowledge base built from Stack Overflow and GitHub Gist. ADANA is implemented as an Android Studio plugin which is able to document code snippets by using ASIA (a code clone detector developed by the same authors) which takes the code snippet selected by the developer in the IDE (*i.e.,* the one she wants to document) and tries to find a clone in its knowledge base. If a clone is found, then the tool writes documentation using the code description taken from Stack Overflow or Gist discussions. It is possible to get those code descriptions from Gists since every code snippet is often accompanied by a short description explaining their purpose. Similarly, from Stack Overflow, they crawled the code descriptions from both the Q&A discussions and the SO documentation: an initiative (now shut down) to create reference material for developers, by collecting code examples that show how to deal with common tasks.

2.3 Approaches using Deep Learning

Deep learning (DL) approaches are on the rise in the field of code comment generators. They train a DL model on a large dataset (usually created from GitHub repositories or Stack Overflow discussions). Then, this model is used to generate a comment or a summary for a previously unseen piece of code. Usually, a small subset of the dataset (~10%) is not used in the training of the model, but kept to test it.

Iyer *et al.* [9] used a data-driven approach to generate short summaries of small code snippets written in C# and SQL queries. Their approach uses an end-to-end neural network, called CODE-NN, whose model is trained on a dataset of code snippets with a description built from Stack Overflow. In particular, their dataset consists of 66,015 C# pairs (<code snippet, description>) and 32,337 SQL pairs (<query, description>). Their qualitative analysis shows that CODE-NN performs well on simple code snippets, but its performance degrades for longer, compositional inputs. In addition, there are cases where the output does not match the input code. This might happen when low-frequency tokens are present or when there are very long range dependencies or compositional structures in the input, such as nested queries.

Hu *et al.* [8] propose an approach using DL called DeepCom. It is built upon advances in Neural Machine Translation (NMT) which aims to translate natural language strings from one language (*e.g.,* Italian) to another (*e.g.,* English). They see generating comments as a variant of the NMT problem, where source code written in a programming language needs to be translated to natural language text. They trained and evaluated DeepCom on a Java dataset of almost 10,000 repositories from GitHub and, despite the dataset being smaller than the one used by Iyer *et al.* [9], they achieved a better performance [8].

Alon *et al.* [2] proposed a framework for predicting program properties using neural networks. In particular, they built a neural network that learns code embeddings, meaning continuous vectors for representing snippets of code. This neural network is then used to generate a method's name given its body. Their evaluation shows that their neural network performs faster and better than existing approaches (including the one by Iyer *et al.* [9]).

2.4 Other techniques

We discuss in this subsection works that do not fit in the previous categories of techniques or that exploit a combination of them.

2.4.1 Rule-based

Rule-based approaches are very similar to pattern-based ones. The main difference is that a rule-based approach takes a set of hand crafted rules (if-then statements), which are usually very hard to define for large domains, and then it applies them on the input to gain some information about it and act accordingly. Pattern-based approaches, instead, look for pre-defined patterns (which can be more complex than simple if-then rules) and analyze them to gain some informations.

A rule-based technique was used to analyze lambda expressions in Java programs [3]. They analyzed the way developers comment lambda expressions and discovered that most of them simply use a high-level documentation. This is why they decided to use a rule-based approach in their work instead of a more complicated technique. We can see an example output of their tool, LambdaDoc, taken directly from their publication, in the table below.

LAMBDA EXPRESSIONS AND CORRESPONDING DOCUMENTATION GENERATED BY LAMBDADOC

Lambda expression	Generated documentation
`callInContext( REPO_USER_2, repo2.getId(), MASTER_BRANCH, () -> createNode ( NodePath.ROOT, "repo2Node" ) );`	This lambda expression does not take any parameter and returns the result of the execution of the "create Node" method with two parameters "NodePath ROOT and "repo2Node" ".
`(Integer t, Integer t1) -> Double.compare( splitEvaluation[t], splitEvaluation[t1])`	This lambda expression takes 2 parameters Integer t and Integer t1 and returns the result of the execution of Double's "compared to" method with two parameters element of "split Evaluation" array t and element of "split Evaluation" array t1.
`.peek(batch->count3= count3+batch.size())`	This lambda expression takes 1 parameter batch and returns count3 equal count3 plus the result of the execution of the "size" method on it.
`.beforeResolved(ExecutableComponent.class, ec -> ec.set("c"))`	This lambda expression takes 1 parameter ec and returns the result of the execution of the "set" method on it with parameter "c".
`return stream.flatMap(t -> Stream.of(value, t ))`	This lambda expression takes 1 parameter t and returns the result of the execution of Stream's "of" method with two parameters "value and t".

FIGURE 2.1: A table showing the source code and its generated documentation by the LambdaDoc tool [3].

A different work by Hassan and Hill [7] uses a similar approach to help novice programmers understand code. They automatically generate a very detailed description of statements that helps inexperienced users to understand what a selected statement does. For example, if a method is called within the statement, its *@return* comment is used to give more details about it to the user. Their work is very different than ours, mostly because they focus on single statements instead of methods and the output of their technique is verbose while we aim at generating more concise comments. For example, a statement analyzed by their tool will show the user a description of everything that is written there, from the type of the variable assigned, to what a function called in the same statement does and return.

Wang *et al.* [17] aim at recognizing Object-Related Statement Sequences within a method in order to divide its body into smaller pieces. Each "piece" represents an *action unit*, defined as a code block composed by a sequence of consecutive statements that logically implement a high-level action or algorithmic step. One action unit is usually too small to be a single method,

but requires more than one statement to be implemented. Their research focuses on generating a small comment for every action unit in a method. Both the identification of the action units and their lexicalization are performed through a set of rules that they developed through a data-driven study over a big dataset of Java methods.

Another paper, from Sridhara *et al.* [14] focuses on comment generation for the parameters of a method. They showed that, usually, there are almost no comments about the parameters of a method. They propose a solution that relies on two sets of heuristics: one to identify the main role of a formal parameter in a method and another one to generate succinct and accurate phrases about those formal parameters. The second set of heuristics is based on SWUM and is used to document formal parameters.

2.4.2 Stereotype-based

Another interesting technique has been used by Newman *et al.* [12] and it is stereotype-based. A stereotype is a high-level description of the role of a method. A stereotype designation gives a clear picture of what a method does and its responsibilities within the class. Constructor, accessor and mutator are all stereotypes often used by developers. A constructor is a method for initializing an instance of a class; an accessor is a method used to read a field of an object without modifying its state; a mutator is a method used to modify a field of an object. This approach is not influenced by names (of variables, methods, *etc.*) since it does not use any natural language processing. The comments generated using this technique are accurate, but they do not resemble real, developer-written comments and are very limited in descriptive capabilities: comments for the same stereotype always provide the same information. An example is shown in Figure 2.2.

```
/**
calcWidthParm is a property that
   returns a computed value new_width
   based on data member: m_range and parameter: number.
Calls to- high()     get
          low()      get
 */
```

FIGURE 2.2: A comment generated by the tool of Newman *et al.* [12]

A similar work was performed in the same year by Guarnera *et al.* [6]. They use the stereotype-based approach to automatically comment code, but they have the same drawbacks of the previous work (do not resemble real, developer-written comments and are very limited in descriptive capabilities).

While the previous approaches were focused on documentation generation for Java methods, another work, authored by Moreno *et al.* [11], uses the stereotype-based approach to generate documentation of Java classes. In particular, they generate a summary of the class by identifying its stereotype, reading the classes it implements or extends and analyzing its public methods' behaviour. An example of the generated summary is shown in Figure 2.3.

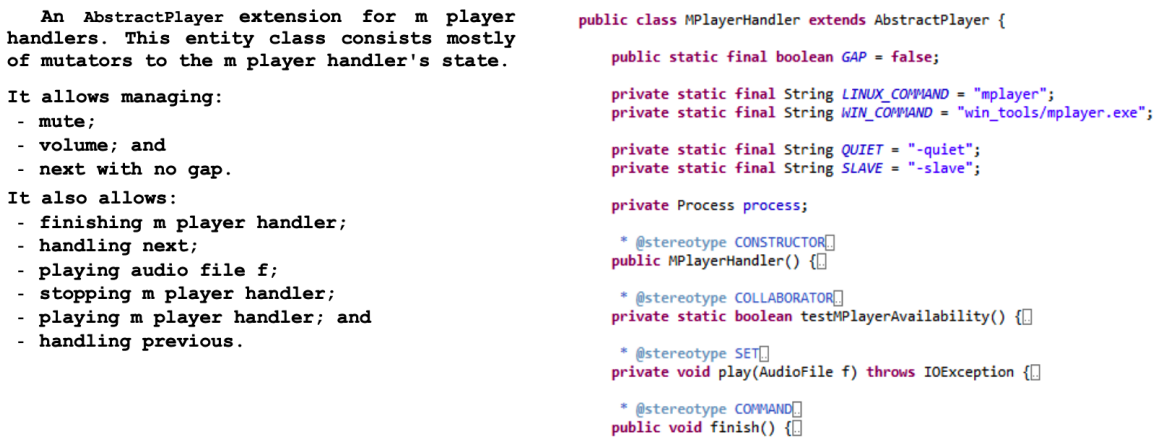

FIGURE 2.3: A class summary generated with the approach of Moreno *et al.* [11] and a fragment of the class code.

2.5 Summing Up

As we have seen by the current state of the art, different approaches have been proposed and implemented to solve the documentation problem. The proposed pattern-based approaches are very limited in terms of granularity (*e.g.,* they only work at method-level, class-level). Additionally, they also try to define patterns for an "entire language". We believe that this is very difficult to achieve so we focused on a specific context (Android applications) where we can also benefit of well-documented APIs. Other proposed approaches are retrieval-based: their weakness is that they are only effective if they are able to find a snippet that is very similar to the document they are trying to document. The last proposed approaches are Deep Learning based, but they still have strong limitations as shown in recent studies [8].

Chapter 3

A Pattern-based Approach to Automatically Generate Code Comments in Android Apps

In this chapter we overview the main steps behind the proposed approach. In order to create a pattern-based approach to document code snippets, we need to firstly define these patterns and, to do so, we need to collect snippets on which these patterns can be defined. The first step is, then, to extract those snippets from Android repositories. We decided to use the F-Droid catalogue[1], since it contains many open-source high-quality Android repositories. We then used a crawler to download ~300 of these repositories. After this, we created a Python script to extract all the methods from every Java files in the repositories and collect code snippets from them. To do so we used different heuristics that we will discuss later. Since it is not realistic to define patterns for all extracted snippets, we decided to cluster the snippets using code similarity and to define patterns only for one representative snippet of each cluster. To support the patterns definition, we implemented a web application using to allow the creation of "abstract" patterns using the AST of the code snippet. Using the application we defined a total of 717 patterns, that can then be used to document previously unseen pieces of code.

3.1 Code snippets mining

The mining of code snippets is the first step of our approach and it is needed to obtain a set of code snippets that will be used to define documentation patterns and, later, to test and evaluate the ability to document previously unseen code. Figure 3.1 depicts all steps that we perform to prepare the patterns definition and the mining of code snippets is the first step of this "preprocessing" pipeline.

[1] https://f-droid.org/

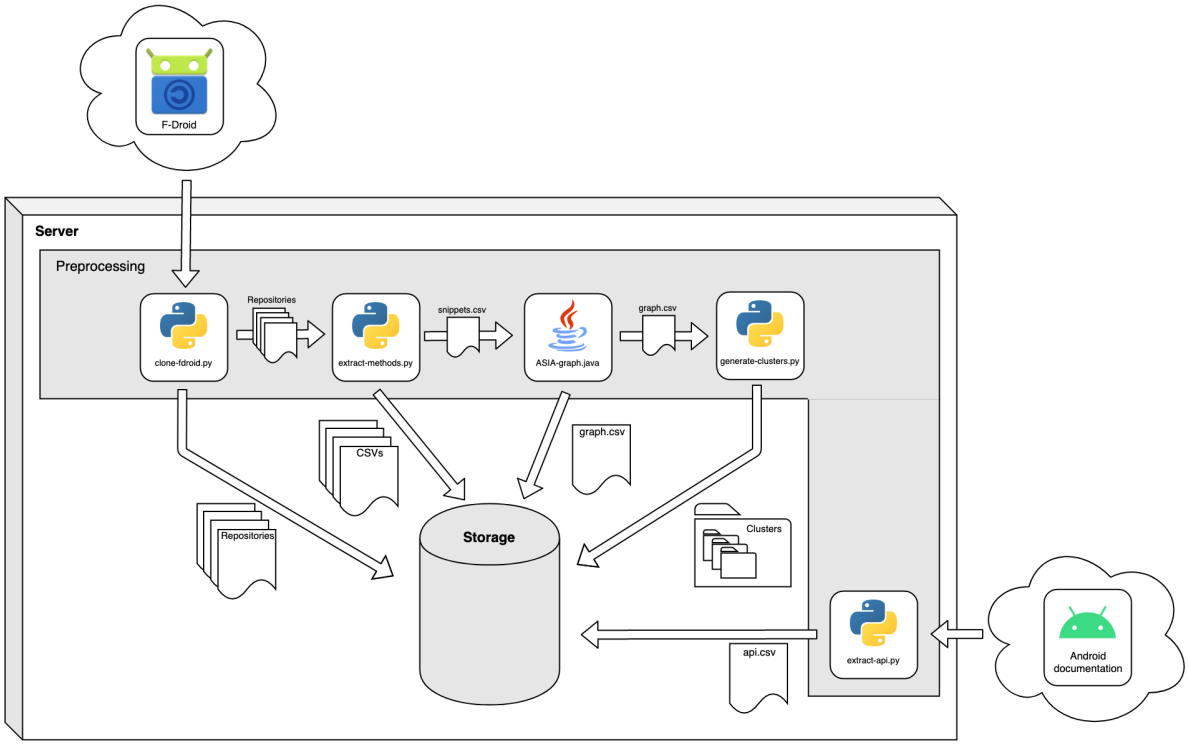

FIGURE 3.1: A diagram showing the main preprocessing steps.

The process of mining snippets is divided into two part: the cloning of Android repositories and the extraction of methods and snippets.

3.1.1 Cloning F-Droid

In order to gather a substantial number of Android repositories, we built a Python script which is able to clone every F-Droid repository using different versioning technologies such as Git, Subversion and Mercurial. While the script will not stop until it has cloned every repository in the collection, we decided to stop it after reaching ~300 repositories. This is due to the fact that the purpose of this thesis is to demonstrate the feasibility of the proposed approach by defining a limited number of patterns rather than investing months in the pattern definition. In case the approach works, a collective effort can then be taken in the research community to generate more patterns. The output of this script will be a folder with every cloned repository which will be then taken as input by the next preprocessing step and saved for later use by the web server. We keep all the source files in each repository, because we will use the Java files later for the patterns' definition.

3.1.2 Methods and snippets extraction

The second step of snippets mining is to extract all the methods from the repositories' Java files and use them to generate the snippets. To do that, we used the Javalang Python library which is able to parse Java files and generate their AST. We used a slightly modified version of the library which adds the position of a node (line number, column number) to its properties. We opened a pull request adding this functionality that has been merged in the original project[2]. Once the methods were extracted and saved in a CSV file, we extracted the code snippets from them. We defined 4 different types of snippets and different heuristics to extract them:

- Full-body snippets: the most simple snippet, it is composed by the whole method body, without further processing.
- Empty-line snippets: often, developers split into particular sections their code to improve its readability. Most of the time, each section is made of cohesive instructions that work together to achieve a single goal. So, we decided to split the method's code by empty lines in order to extract those particular sections of its body, aiming to extract code snippets that represent a single functionality of the method.
- Block snippets: a natural way of splitting a method is by block. Blocks often represent a single functionality in a method and they have their own scope.
- Empty-line in block snippets: this is a combination of the two previous types of snippet: from the extracted blocks, we create further snippets by splitting the block's body on empty lines.

There were some complications while extracting the snippets (*i.e.,* snippets that were cropped when their last line had a newline in it, strange indentation not optimal for reviewing, incoherent start line or end line of the snippet). To solve them, we implemented some helper methods in the *utils.py* file. Most of them are used to fix incoherent results and to extract blocks correctly.

As a last functionality, the *extract-methods.py* script extracts all the method invocations and checks whether they are Android API methods. This is done by relying on the Android API documentation. In particular, we use information from import statements and invocation parameters to verify whether a method invocation refers to an Android API. Also, for each extracted snippet, we store a reference to its method and the path of its Java file. Using this approach, we collected ~930k code snippets.

[2] https://github.com/c2nes/javalang/pull/70

3.2 Clustering code snippets

The second part of the preprocessing, previously shown in Figure 3.1, aims at clustering the collected code snippets based on their similarity. It is important to perform the clustering because defining a pattern for each extracted snippet would be unfeasible in the available time, and also inefficient since many code snippets will be exact duplicates or very similar to other snippets. For these reasons, we decided to cluster the code snippets based on their code similarity and then to define patterns only on the clusters' centroids (the snippet with the highest number of connections within a cluster) which are the most representative snippets of each cluster. This also means that they would be similar to many other snippets and creating patterns for them would mean to also have patterns that could be probably reused on their connected snippets as well.

3.2.1 Computing similarity between methods

Before performing the clustering, we first had to create a similarity graph. To build it, we used ASIA, a code clone detector already mentioned in the State of the art section while discussing ADANA [1]. ASIA can compute the code similarity between two Android code snippets. Since ASIA is a Java library, we created a Java program to build a weighted undirected graph made by the snippets (the nodes) and their similarity (the weighted edges). We defined the minimum similarity to add an edge in the graph to 90%. This may seem a very high threshold. However, after several attempts, we noticed that the threshold was appropriate to ensure that the obtained clusters actually contain similar code snippets. The program takes as input every snippet extracted in the previous step and generates a graph in CSV format where each line contains the *snippet1, snippet2, similarity* fields. *Snippet1* and *snippet2* are two integers representing the id of two connected snippets and *similarity* is their similarity. This step is very time-consuming and the time requirement increases quadratically with the number of snippets (we need to compare each snippet with every other snippet).

3.2.2 Clustering algorithm

Now that we have the graph, we want to extract clusters of code snippets from it. This is because we want to generate patterns only for their centroid so we can be sure to have code snippets that represent code often used in the Android environment and to reduce the number of code snippets to manually review while maximizing the effectiveness of the patterns. To do that, we used the NetworkX[3] Python library which is able, given the graph previously built, to automatically extract its connected components. For every connected component, we found its centroid and we saved the whole connected component in a directory hierarchy. The directory name which contains the connected component is named after the centroid's snippet id and it contains many text files named after each node's (centroid included) snippet id. Each text file contains the code of the corresponding snippet. In this way we can easily browse the hierarchy and see if the results were good enough for our usage. Using a high similarity threshold (90%) has, however, its downsides: the number of graph nodes without any connection is very high and many snippets were discarded because they did not belong to any cluster having no connections with other snippets in the similarity graph.

[3] https://networkx.github.io/

3.3 Mining the Documentation of Android APIs

An additional step of the preprocessing is the Android API documentation extraction. It is employed to help the user during the pattern definition process (as described later), by showing her the documentation of Android methods found within the snippet. The extraction script connects to the Android API documentation website and, by parsing its content, saves all the data we need about packages (name, description and its dedicated page's URL). Once a package (*i.e., androidx.activity*) is saved, we navigate to its URL and start parsing all of its classes and interfaces saving the same fields as the packages in addition to its type (if it is a class or an interface) and the package name which contains the class or interface. In the end, we can finally navigate to each Android class and interface documentation web page to parse and save their methods, which can then be used to facilitate the patterns definition.

3.4 A Web-based Application to Support the Definition of Patterns

As previously mentioned, we decided to implement a web-based solution to support the definition of the patterns. The result is a web application which allows a (set of) user(s) to define patterns for a particular code snippet. Her task will be supported by the web application with features like pattern preview, an easy way to quickly read Android methods' documentation and some tools to easily create sub-patterns. In the next sections, we are going to explore the functionalities of this web application: we will start with a list of its functional requirements. Then, we will move to its architecture and demonstrate how a user can use it to define patterns (we will also explain the syntax of patterns) and a practical example of pattern definition. Finally, we explain how already defined patterns can be reused to document previously unseen pieces of code.

3.4.1 Functional Requirements

We will now see what functionalities we expect from the web application, along with a short description.

Simplify the interaction with the code snippet

The primary aim of the tool is to provide an easy-to-use interface to define documentation patterns for code snippets. Thus, the tool should be able to visualize code snippets and make it easy to interact with them. In particular, we expect the web application to show the code snippet together with the class from which it was taken. This is because the context of a snippet can greatly help the user to identify what that code snippet does.

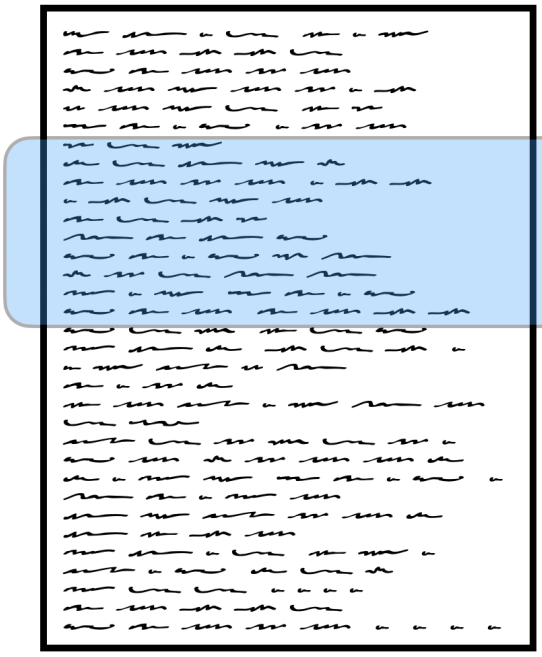

FIGURE 3.2: How the web application should show the code snippet to the user. The highlighted part is the code snippet, while the rest of the page is the file from which it comes from.

The user should be able to interact with the code snippets in multiple ways:

- Scroll the page in order to navigate the snippet and/or the file containing the snippet;
- Create a new snippet from the existing one, by reusing parts of it;
- Browse the Abstract Syntax Tree (AST) of the code snippet in the pattern definition.

The first interaction (scrolling) should be performed using a scroll bar or any other common way a user would scroll through a page. The second interaction (creation of sub-snippets) should be performed by selecting the subset of code that the user would like to have in the new sub-snippet and then pressing a button or clicking the right item of a context menu (shown through a right click). Lastly, the third interaction (visualize the AST of the snippet) should be performed by clicking anywhere in the code snippet and a popup should show to the user the AST of that part of the code that can be used in the patterns definition. The popup should also allow the user to quickly use the AST nodes by clicking on them. Clicking anywhere else should dismiss the popup.

Define patterns

The web application should allow the user to easily define patterns for the current code snippet. For this reason, a text area should be provided where the user can write the pattern. It would be useful to also have keyword highlighting and auto-completion. Additionally, the user should be able to see a preview of the pattern he is defining applied to the current code snippet (i.e., the documentation that would be generated by applying the pattern on the code snippet). In this way, he should be able to verify that his pattern is matched like he expects. Since the same

keyword in a pattern might match multiple parts of the code, the user should be able to choose the part of the code he wants to match directly from the preview. Lastly, the user should be able to rapidly see if the pattern's requirements are satisfied within the current snippet.

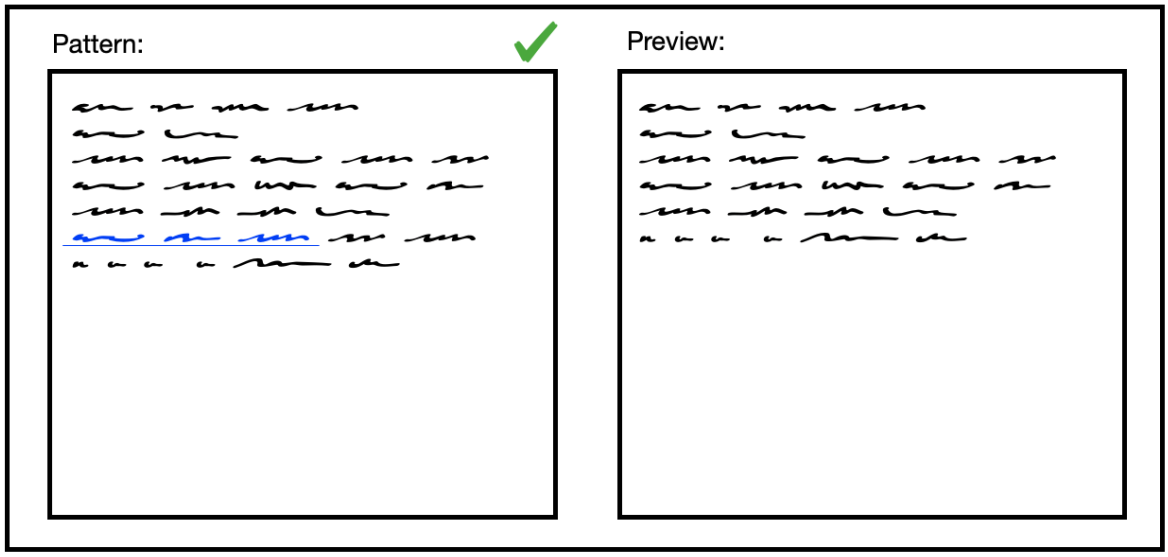

FIGURE 3.3: A visualization of how the pattern definition should appear. We can see a keyword (in blue and underscored) and a tick that signals the user about the validity of the pattern.

Reuse patterns

The web application should allow the user to easily reuse existing patterns in her new pattern. It should also allow to modify the natural language text generated from the reused pattern in order to better match its new pattern and to choose the keywords she wants to reuse. In order to reuse the patterns, the user should also be able to view them and see to which lines they are matched.

Browse Android documentation

The web application should allow the user to quickly browse the Android documentation, since most of the methods called in the code snippets will be from the Android API.

3.4.2 Architecture

In this section we describe the architecture of our system and we explain more in detail every component of the architecture.

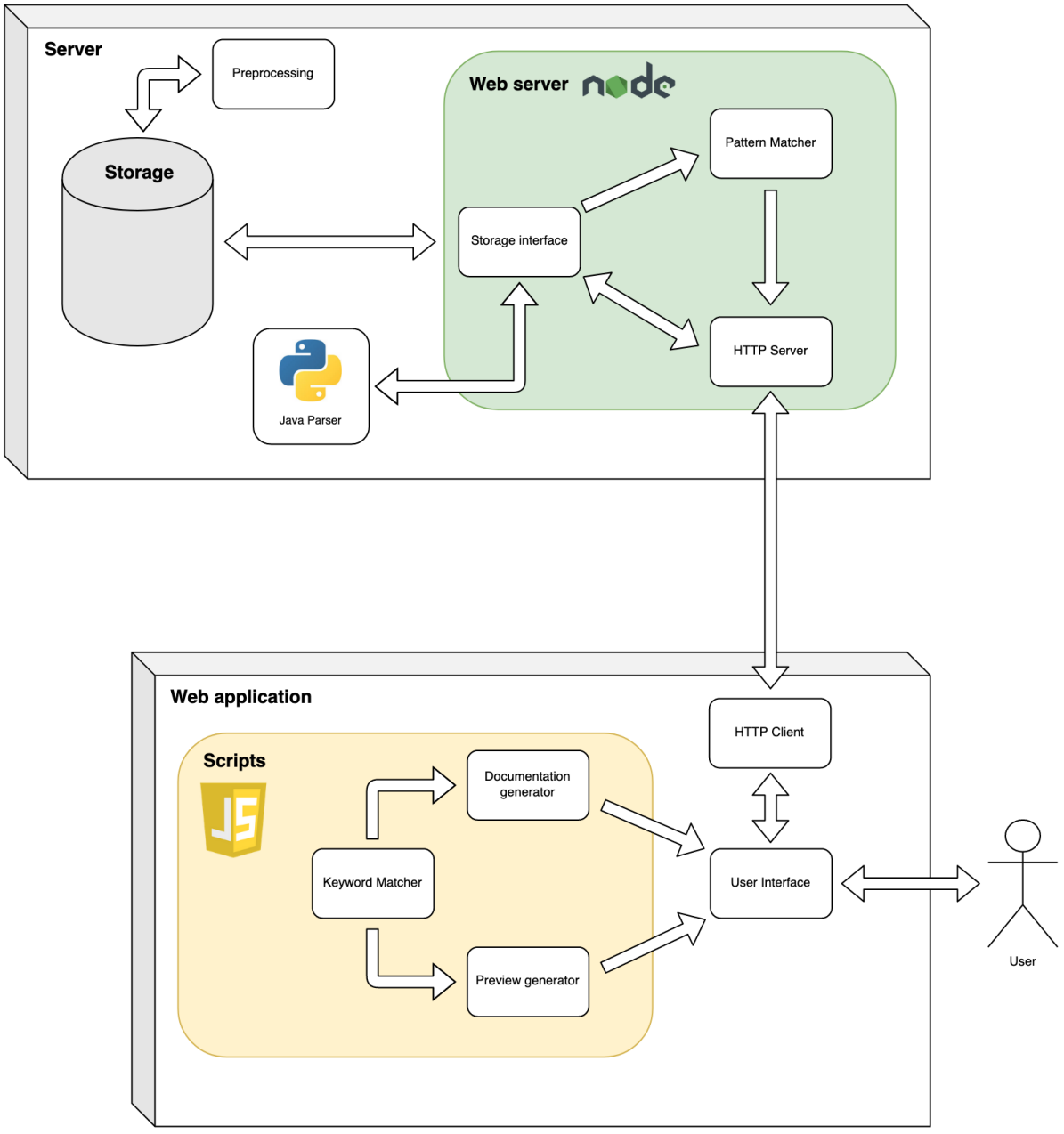

FIGURE 3.4: The full system architecture diagram.

Figure 3.4 depicts the architecture of our web application, that will be detailed in the next two sections: backend and frontend.

3.4.3 Backend

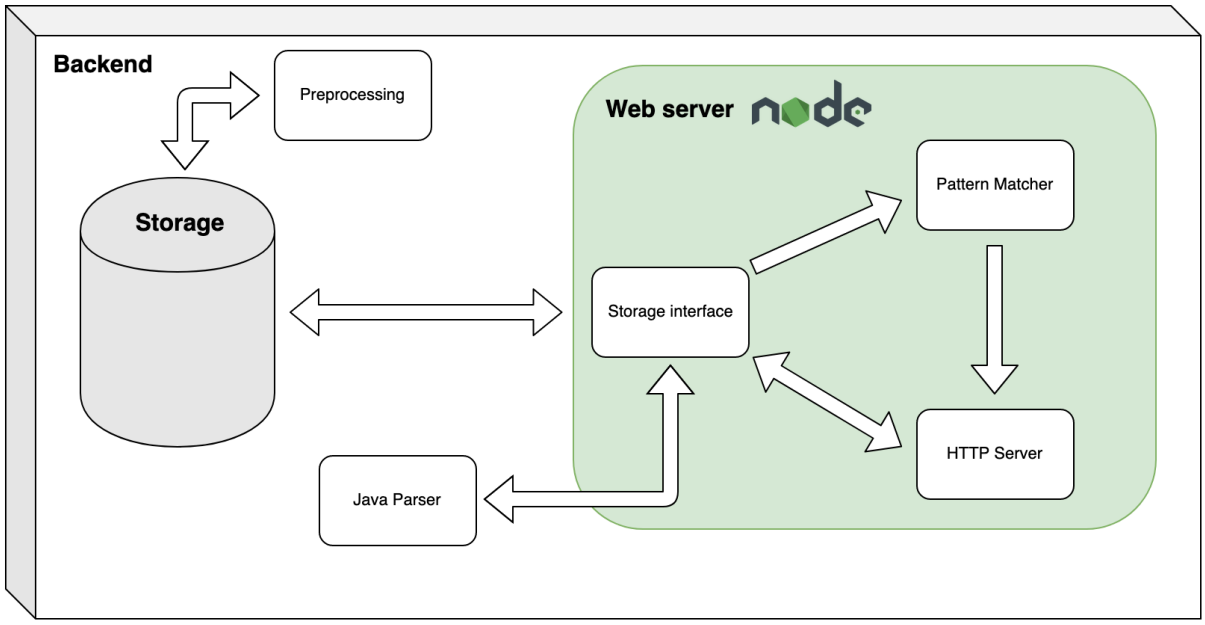

FIGURE 3.5: A diagram showing the backend architecture.

As we can see from the diagram, part of the backend is the preprocessing of the data which we have already seen in the previous section, with the Storage representing the file system. We store the data in JSON and CSV format. The only files which are modified by the web server are dedicated to hold the custom snippets and the defined patterns. They are updated by the server synchronously and backups are periodically performed on them. We will now see each component more in detail.

Storage Interface The storage interface is a component which handles every file system's operation made by the backend. This includes the writing of a new snippet or pattern as well as reading all the preprocessed files when the server is first launched to save their data in the memory to be used by the backend. Then, it offers an interface to easily access such data. For example, when a snippet should be sent to the frontend, one of the functionalities of the Storage Interface is to find the snippet and send it back. It also keeps track of snippets which were already served to avoid multiple users writing patterns for the same snippet. Finally, another important functionality of this component is to invoke and retrieve the result from the Java parser when a user attempts to create a new snippet from raw code.

Java Parser The Java parser used by the backend is a Python script which invokes the modified Javalang parser used during the preprocessing on a code snippet which is passed through a command line parameter. Then it outputs the result of the parsing into the standard output. As mentioned before, this component is invoked when the user wants to add a new snippet and we need its AST before being able to use it in our web application. An important note is that the code snippet, to be parsable, must be a syntactically correct (i.e., compilable) Java class. This means that all of the code snippets submitted by the user will not be parsable since they will all be part of a Java method and, for this reason, will never be a full Java class. To fix this problem, we wrap the user-submitted code snippet in a fake Java class with a single *main* method. Being part of a class and a method, the code snippet will be compilable and return its

AST. Most of the time, however, the user will still have to adjust mismatching parentheses and missing semicolons or other minor syntactical errors.

HTTP Server The HTTP server is a very simple web server (powered by Express) with only three routes:

- */snippet* This route can be used (with a GET request) to receive data about a random snippet which is not being reviewed by another user. Other than simply serving the snippet, the route will make sure to add all the data about used Android APIs (if available) and all the existing patterns that match the snippet, found using the Pattern Matcher (explained in the Pattern Matching section). Additionally, it is possible to ask this route for a specific snippet, using the query parameter *snippetId* with the identifier of the wanted snippet as value (*i.e., /snippet?snippetId=1*).

- */snippet/submit* This route can be used (with a POST request) to submit a new pattern to the system. The body of the request must be a JSON object which contains the pattern data. This route should never be invoked manually, but only by the Web Application. After a pattern has been submitted using this route, it will be available for matching and the server will attempt to match it to every subsequent code snippet that is requested.

- */snippet/new* This route can be used (using a POST request) to create a new code snippet. The expected JSON data contains the code of the new snippet and the type (it can be a snippet from an existing method or a completely new snippet). Once submitted, if there are no errors, the snippet will be saved and it will be available for normal usage. This route will also reply to the client with the new snippet's identifier so that it can navigate to it or use it immediately after being created.

All of the routes will return an error message with status 500 to the client if the Data Storage component is still loading the data.

Frontend

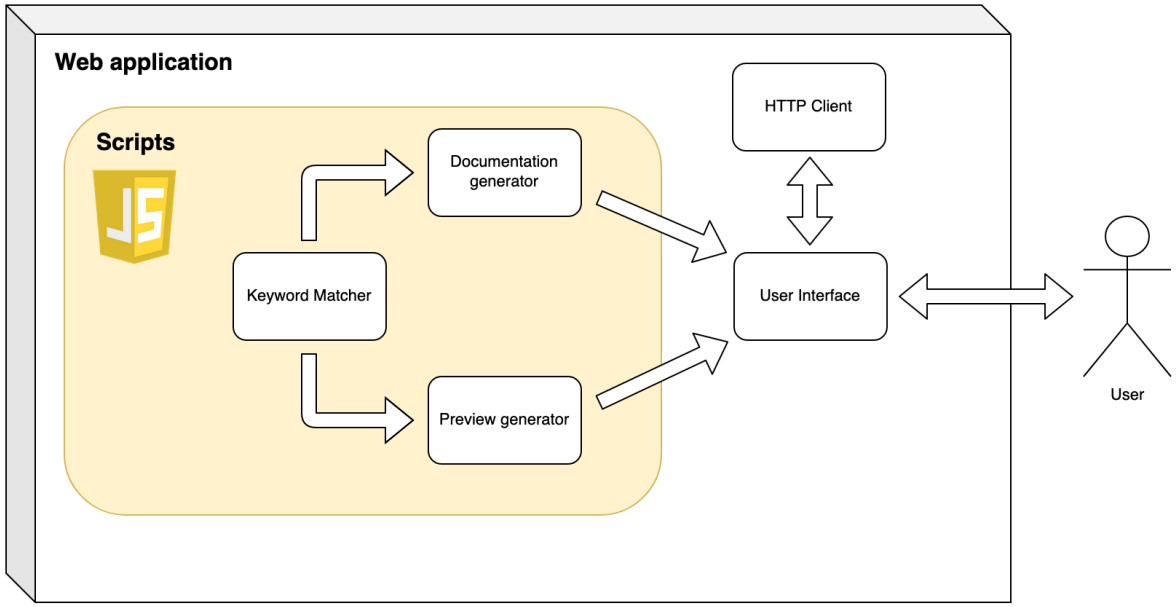

FIGURE 3.6: A diagram showing the frontend web application architecture.

The Web Application frontend is a website built with HTML, CSS and Javascript. Most of its component are described more in detail in another section (User Interface, Keyword Matcher and Documentation Generator). However, we give a quick overview as well as some additional information about the Keyword Matcher.

Interface The user interface is used to interact with the underlying architecture. It can be employed to access the various functionalities of the web application while maintaining a clear design and hiding most of the complexity from the user. Since the user interface has multiple functionalities, we describe it in more details in a dedicated section.

Documentation Generator The documentation generator is the component that, given a code snippet, attempts to generate its documentation. It does that by finding the best patterns that can be applied to a snippet and applying them with the help of the keyword matcher. How this component picks the best pattern and how the keyword matcher works will be explained more in detail in a dedicated section.

HTTP Client The HTTP client is a simple component which performs AJAX requests and handles the response. For example, when submitting a new pattern it will POST it to the server and then, on successful response, it will navigate to the new snippet. Another example is, when the page is loaded, it asks the server for a snippet and, on response, it calls the User Interface to render it to the user.

Preview Generator The preview generator handles modifications made by the user to the pattern he is defining, in order to show a preview of how the documentation will look like when that pattern is applied to the given code snippet. To do that, it uses the Keyword Matcher to resolve all the AST paths (we will see them more in detail in the next section) and use them to check the requirements of the pattern and replace all the AST nodes used in the pattern with their actual value. If multiple patterns are matched, the user will be able to modify the preview to see the result of the pattern on different lines. Additionally, if some AST nodes

are dependent on a sub-pattern, previewing a different match for any of the same-pattern AST paths will also modify all the other AST paths for the same sub-pattern in order to reflect the correct application of the sub-pattern. If one of the keywords cannot be matched, the user will see a red "No matches!" message in the preview.

Keyword Matcher In a following section, we will see the mechanism to resolve AST nodes and match a pattern to a snippet. The difference between the Pattern Matcher (found in the backend architecture) and the Keyword Matcher is that, while the first attempts to match a whole pattern to a snippet, ignoring the result of the matching and only checking whether a path matches or not, the Keyword Matcher is a subset of it, which only has to resolve single AST paths to show the result to the user.

3.5 Patterns Definition

The pattern definition is the most important part of the approach, along with pattern matching. For this reason, we tried to make the pattern definition process as simple as possible, by keeping the interface clear and by reusing symbols that are often used for similar functionalities in other computer science environment. We will now see more in detail what is a pattern and the functionalities of every keyword and how they should be used while defining a pattern.

3.5.1 What is a pattern?

A pattern is a natural language string that describes what a code snippet does. In addition to the natural language, there is a way to insert parts of the code snippet (*i.e.,* method names, variable names) using AST nodes. Moreover, a pattern can (and should) have some defined requirements that must be respected by a code snippet before the pattern can be matched to it. In other words, if the requirements are not satisfied by a code snippet, the pattern cannot be used to generate documentation for that specific snippet.

3.5.2 AST nodes

As mentioned in previous sections, the code snippet is associated with its Abstract Syntax Tree, represented in a JSON format. An AST node is a JSON path that matches a single element of the AST. For example, suppose our snippet's AST is shown by the web application in the following JSON format as can be seen in Figure 3.7.

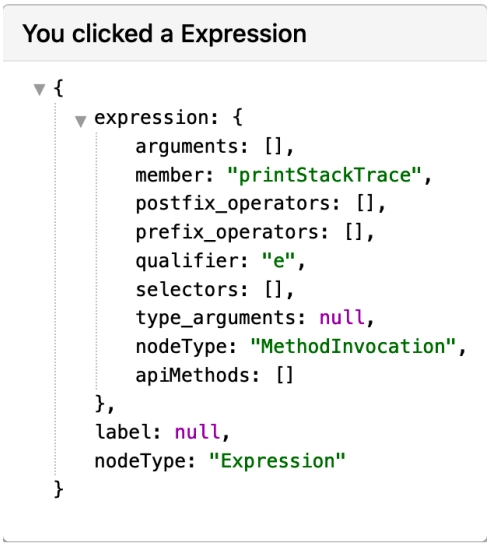

FIGURE 3.7: AST of the *e.printStackTrace()* code.

Figure 3.7 shows the AST of the *e.printStackTrace();* code line in JSON format. Let's now assume that we want the generated documentation for this line to be "Print the stack trace of the exception e". Instead of writing that exact sentence, we might want to get the variable name *e* to be taken directly from the code. In this way, the pattern will be compatible with a much larger amount of snippets such as *error.printStackTrace();* or *ex.printStackTrace();*. To do this, we use AST nodes. Using them is very simple: we just need to follow a path from the beginning of the AST to the node we want to use. In our example, the path to reach the variable *e* is *"expression.qualifier"*. Using AST nodes in a pattern definition requires the node to be enclosed in the symbols <...>. So, in our example, the pattern will be: *"Print the stack trace of the exception <expression.qualifier> ."*. It is possible that the same path matches more nodes, in that case the user can choose which one she wants to use, with the default one being the first match.

Advanced syntax

Other than simply following the AST path to use an AST node, it is possible to use other keywords in the path to obtain more complex result:

- **?** if the user types a question mark (instead of a node) in an AST path, it will match any single node that is in that position. For example, if one types *expression.?.member*, this path will match to every child of *expression* that has a child with the *member* property (i.e., *expression.expressionl.member*).

- **+** similar to the question mark, a plus will attempt to match any number of nodes until we reach a node which has the next property (if any). So, *expression.+.member* will match, for example, *expression.expressionl.member*, but it will also match *expression.expressionl.expression.expressionl.member*. It is important, however, to avoid using the plus and the question mark as the last node in the AST path: there would be a lot of matches that won't tell us anything about the code we are documenting.

- **:** the colon symbol, preceded by a pattern id, is used to restrict a path to only the lines matched by another pattern. There can be multiple colon symbols and they should be

ordered (*i.e.,* 1:2: is different than 2:1:). So, using *1:PATH* means that the path will be matched only amongst lines that are first matched by the pattern with id 1. This symbol is used automatically when the user reuses a pattern and is rarely used directly by the user.

- ^ the last keyword, mostly used to interact with getters and setters. This keyword must be added at the end of an AST path and it only works with string results. This keyword is used to remove the first part of the matched result and to restrict the result to only strings that starts with a given string. An example will clarify it better: suppose we have *expression.member* that matches with "findViewById", "getText", "setText". If we want to match only the getter, we can use the "^" keyword: *expression.member^get*. This AST path will match only "getText" and it will return "text" as result, instead of "getText". In this way we can generalize very well a generic getter: *i.e., box.getText()* can return "Get the box text" with the pattern *Get the <expression.qualifier> <expression.member^get>*.

3.5.3 Requirements

Requirements are a fundamental part of the pattern: they prevent a pattern from matching lines that do not satisfy them. This is useful because, for example, if we made a pattern for a method called *getArea()* that describes the functionality of that method with the text "Return the area.", we surely do not want that the pattern is matched with the method *setCounter()*. In the example, we might want to have something that says to the system that the pattern is to be matched only to the *getArea()* method: this is what requirements do. In our system, we have two keywords that are used to define requirements: *!require* and *!satisfies*.

!require

The *!require* keyword is the most used amongst the two. It allows the user to compare an AST node with a String or to compare two AST nodes. If the value of the two parts in the current snippet is the same, the requirement is satisfied for that snippet. If the AST nodes has more matches, the requirements is satisfied if at least one of the matches respects the condition. The syntax for writing requirements is the following:

$$!require\ X == Y > N$$

where X and Y are two AST nodes or Strings and N is the minimum number of matches that satisfy the requirement. For example, the requirement *!require expression.member == findViewById > 2* is satisfied if the code snippet contains 3 or more *findViewById* at the path *expression.member*. It is possible to omit N and simplify the syntax to *!require X == Y*. In that case, N is 0 and a single match is enough to satisfy the requirement. Multiple requirements can be defined, each one of them must be on its own line.

!satisfies

The *!satisfies* keyword is used to signal the system that, in order to reuse the pattern we are defining, the snippet must also be able to reuse other patterns whose identifiers are specified with this keyword. To do that, we can add a line in the pattern definition that follow this syntax:

!satisfies x,y,z,...

An example is *!satisfies 1*: this line means that the snippet must first satisfy the pattern with id 1 before being able to reuse our pattern. As seen from the syntax, we can specify more than one pattern using a comma *i.e., !satisfies 1, 2, 3.* The id of a pattern can be found in the tabs under the preview area or extracted from the syntax of pattern reuse. We will see this in more detail in the next section where we demonstrate the pattern definition through an example usage scenario.

3.6 Usage Scenario

3.6.1 Interface description

The web application we created to support the definition of patterns is a very simple web page with a clean design that allows the user to quickly jump to the pattern definition. Figure 3.8 presents a screenshot of the Web Application.

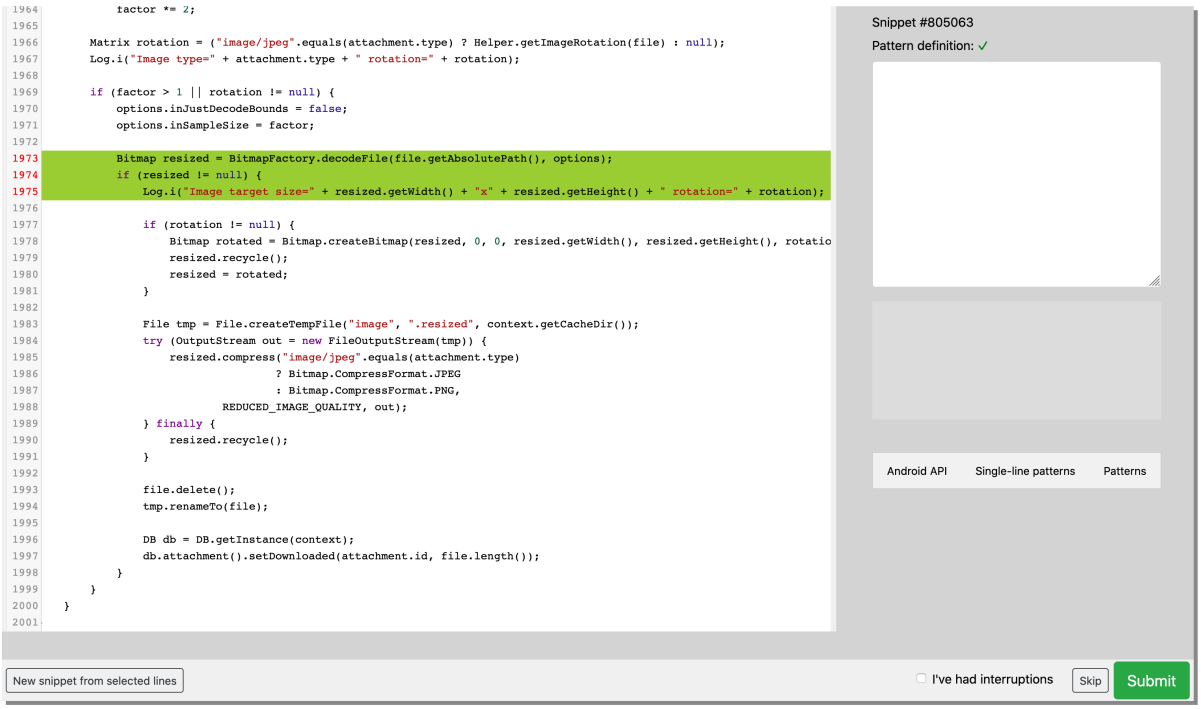

FIGURE 3.8: The web application user interface.

The main interface is divided into 3 parts: the code for which a pattern must be defined (left side), the pattern definition UI (right side) and the option bar (bottom). Using the web application, the user can submit a pattern for the code snippet highlighted in the code and then submit it by pressing the "Submit" button. We will now analyze every part more in detail, starting from the code.

The code snippet

The code snippet is highlighted in green, while the class containing it has a white background. This is because we choose to show the user not only the code snippet for which a pattern must be defined, but the whole Java code that contains it as well. The reason behind this choice is that, sometimes, it might be difficult to understand what a piece of code does without reading its context. So, we decided to show the whole Java class to the user, while, at the same time, highlighting the lines that contain the code snippet. Also, some line numbers use a bright red font color instead of the "standard" gray one. That signals the user that an existing pattern, previously defined, matches the specific line. Such a feature is meant to maximize pattern reuse so, when an existing pattern is matched, we strongly advise the user to reuse it (if it is appropriate) while defining the new pattern. This allows to build hierarchies of patterns, with more complex ones containing (and depending on) simpler ones. To do that, a user can click the red line number and a popup appears (see Figure 3.9).

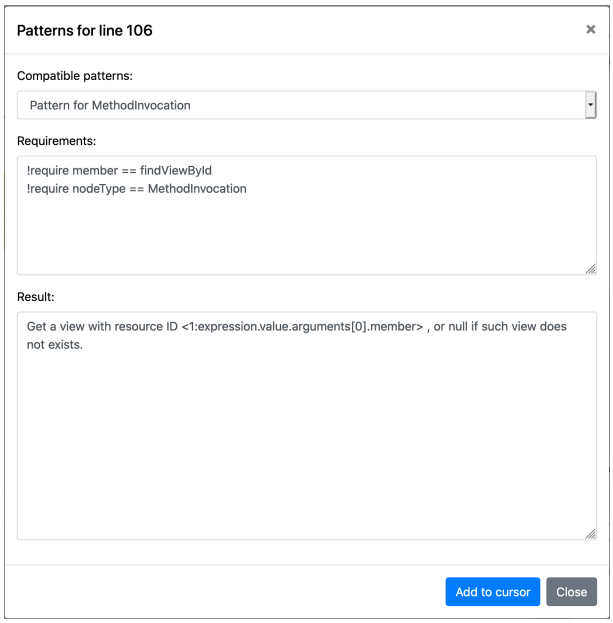

FIGURE 3.9: The popup that the user will see when clicking on a red line number to reuse a pattern.

The user, in the popup, has access to information about the matched pattern, along with a list of matched patterns (if there are more than one). The elements of the popup are, from the top to the bottom:

- A title, indicating which line of code has been clicked.
- A list of all the patterns found within that line. The user can pick the pattern he wants to see or reuse by clicking on the control. In the case of single-line patterns, the user can see what part of the line of code has a matched pattern (in the figure, a Method Invocation). For multi-line patterns, instead, the user will see the range of covered lines. When the popup is opened, one pattern from the list is automatically selected and shown to the user. Selecting a new pattern will modify the next two text area controls.

- A text area containing the requirements for that pattern. We had seen a more detailed description and format of them in the previous chapter. If a pattern is matched, however, it means that, for that specific line of code, its requirements are satisfied.
- A text area containing the pattern itself, ready to be reused. In this area the user can see the exact text that will be added to its pattern if he choose to reuse it.
- Two buttons: one to add the pattern to the pattern definition area and one to close the popup without other actions.

In case there are multi-line patterns matched on the current snippet, the user will see rectangles around the covered code lines. Every rectangle uses a different color for each pattern. If the same pattern is matched twice, both rectangles will be of the same color. To access the multi-line pattern, however, the user will still have to click on one of the line numbers within the rectangle.

Finally, the UI rendering the code also aids the user with the pattern definition by showing the AST of a line of code. A user can access this information by clicking on a highlighted line of code. At this point, a popup will appear like the one shown in Figure 3.10.

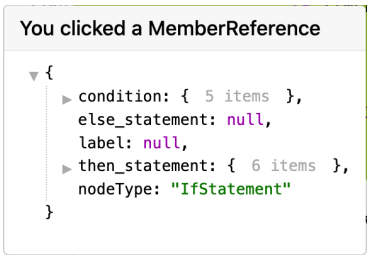

FIGURE 3.10: The popup showing an AST that the user will see when clicking on an highlighted line of code.

As we can see, the AST adopts a JSON format, which makes it easier to navigate and visualize. At the top, we can see the type of node we clicked, but the AST will start from the first node of the line. By navigating through it, it will be possible to reach any part of the line of code. To make the patterns more reusable and dynamic, the user can click on the name of an AST node (*i.e.,* nodeType) to add it to the pattern definition area. This is of great help during a pattern definition and also has a role during pattern matching (described in the next chapter). The code container is implemented using CodeMirror[4].

[4] https://codemirror.net/

The pattern definition UI

The pattern definition UI contains all the tools that the user can use to define a pattern.

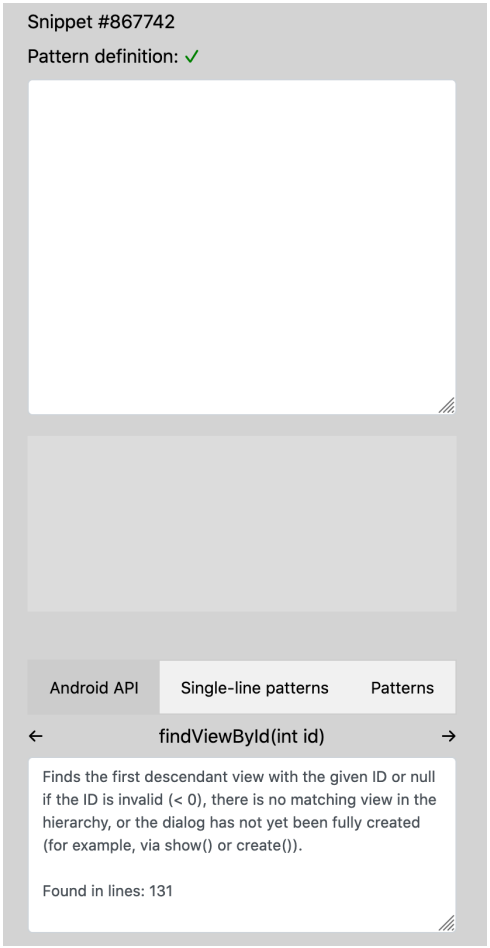

FIGURE 3.11: The pattern definition part of the web application.

We describe it by explaining every component in Figure 3.11 from the top to the bottom:

- The id of the snippet, which can be useful in certain cases. One can use it to access that particular snippet using the URL parameters *?snippetId=id* while navigating to the web application. If one reached this snippet from another one, she will have a link to the old snippet to the left of this snippet id (detectable also as a URL parameter *?from=id*).
- The "Pattern definition:" text, clicking on it will open a popup that shows the documentation automatically generated by our tool reusing the available patterns.
- Near the "Pattern definition:" text, there is a green tick (or a red cross) indicating if all the requirements for the specified patterns are currently satisfied.
- A text area where the pattern will be written by the user.
- A preview area (in gray). While the user writes its pattern, a preview of how that pattern applies to the current code snippets will appear. It automatically updates on changes and

the user is able to choose the line of codes to which apply part of the pattern by clicking on the matched AST nodes (*i.e.,* an AST node can occur both at line 109 and 111, the system will pick line 109, but the user can click on the node to make it match the next line, 111).

- Three different tabs which show details about the current snippet. In the first tab, we can see details about the Android APIs found in the snippet (name, parameters, description extracted from the official Android documentation, lines where that API is found). The second and third tab contain, respectively, details about single-line and multi-line patterns found in the snippet (requirements, actual pattern) without the need of manually finding them amongst the red line numbers.

The bottom bar

The bottom bar mainly contains buttons used to submit and skip the current snippet or to create a sub-snippet.

FIGURE 3.12: The bottom bar part of the web application.

This time, we will describe it from left to right:

- The new snippet button is used to create a sub-snippet. Indeed it could happen that, while writing patterns, the user sees some code snippets, different from the one under analysis, that can benefit from the definition of a specific pattern aimed at documenting it. By selecting that code, the user can then click on the New Snippet button and generate a new snippet. At this point, a popup will open with two different options for the user (see Figure 3.13).

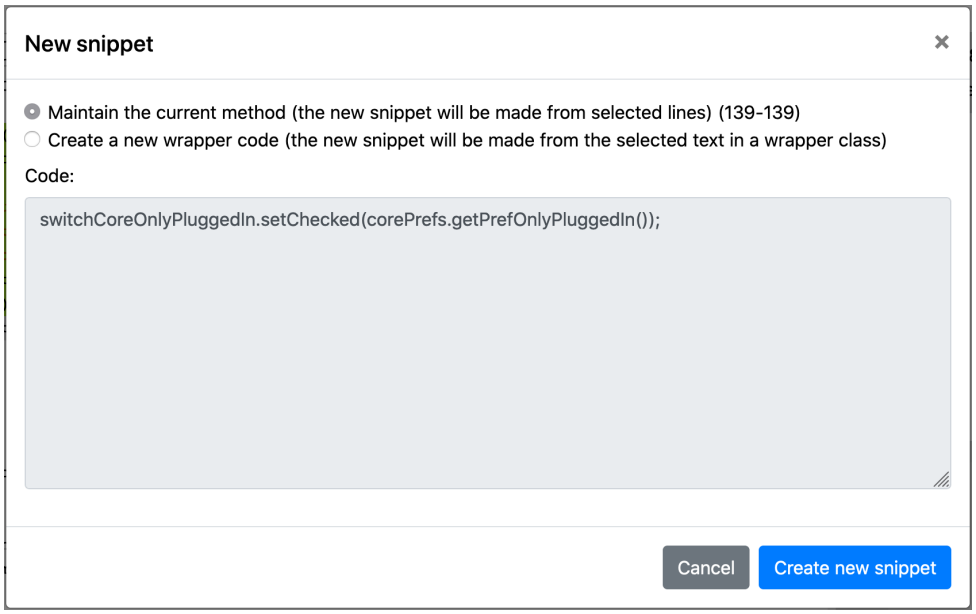

FIGURE 3.13: The new snippet popup.

The popup allows the user to choose between two types of new snippets:

- Keep the current method: it allows the user to generate a new snippet from the selected lines while keeping the same file. So the new snippet will be in the same file as the current one, but it will only cover the selected subset of lines. After the creation of the snippet, it will be proposed to the user.
- Create a new wrapper code: the user should do this only for in-line code, that does not need the whole line (*i.e.,* a function called as a method parameter). While choosing this option, the user is able to modify the selected code. This possibility, however, should only be used to make minor adjustments in order to make the code parsable (usually, it is needed to adjust some parantheses or add a semicolon). This is because, otherwise, the user could write any arbitrary piece of code in there, which defeats the purpose of using real snippets extracted from open-source repositories. The code will be then wrapped in a clean Java class, inside its *main* method, saved as a snippet and proposed to the user.

After choosing one of the two options, the user will be redirected to a new snippet. She can always return to the previous one by clicking on the link that will appear on the top of the page, near the snippet id, as described in the previous part.

- The "I've had interruptions" checkbox: before submitting the pattern, the user should check this box if she had any interruption while defining it. This will be saved in the database and can be used while analyzing the data about the submitted patterns (*i.e.,* mean time to define a pattern, with and without interruptions).
- The Skip button: it allows the user to skip the current snippet and get a new one. Anything written in the pattern definition area will be ignored and the current snippet will not be proposed to the user again. After the user submits a pattern, the text of this button becomes "Next", but its functionalities stay the same.
- The Submit button: this button sends the defined pattern to the server. It won't redirect the user to another snippet, since he might still want to interact with the current snippet after submitting. The user will receive a confirmation that its snippet has been successfully saved. At this point, the defined pattern will be available and attempted to match with every code snippet. The current snippet, however, will not immediately show any changes. If the user wants to see the new pattern applied to the current snippet, he can return to the same snippet using the URL parameter *?snippetId=id*.

3.6.2 Example of a pattern definition

Now that we are familiar with the web application's interface, we can proceed to an example of a simple pattern definition. The code for which we will define the sample pattern contains a call to a common Android API and it stores the result into a variable. We describe this process from the beginning.

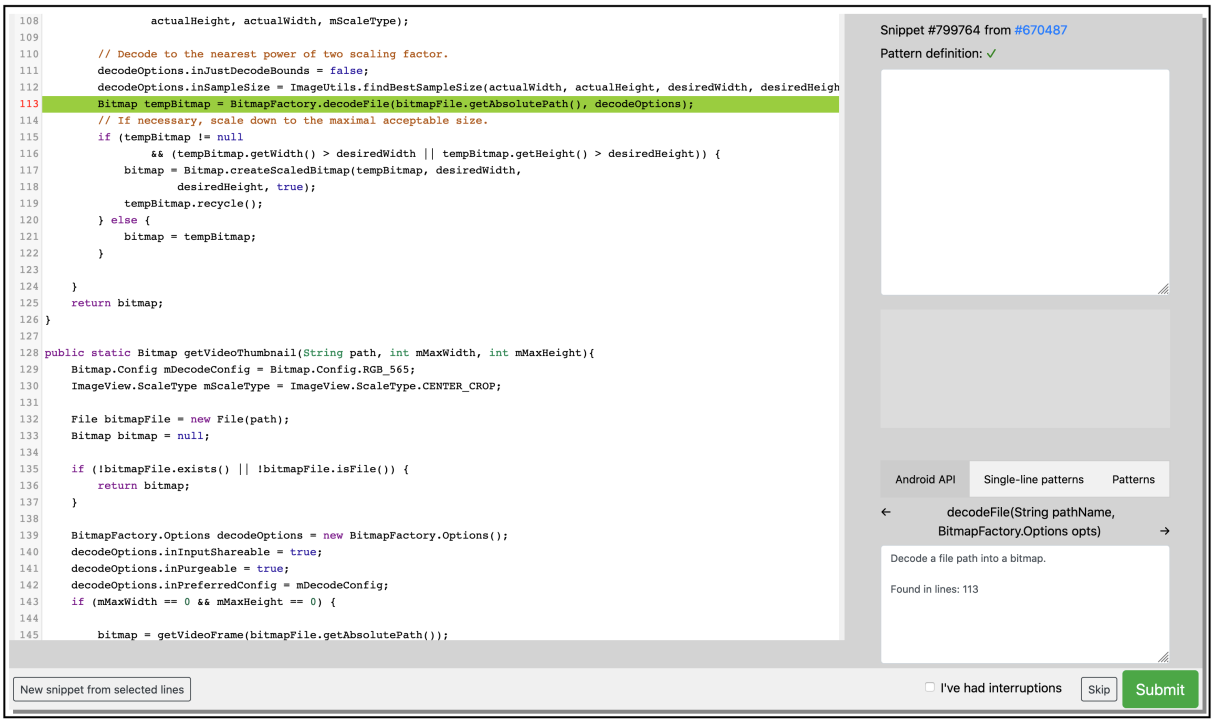

FIGURE 3.14: When the page is first loaded, this is what the user sees.

When the web application is loaded, the user sees the UI shown in Figure 3.14. First of all, one must understand what the code snippet does. To do that, the user proceeds to read our code snippet which, in this example, will be the line of code highlighted in green. As we can see, her snippet contains the declaration of a variable called *tempBitmap* of type *Bitmap*. Additionally, this variable is initialized with the return value of the function *BitmapFactory.decodeFile(...)*.

If the user does not know what a particular Android function does (and it would be unrealistic to assume otherwise), she can read the description of that function using the "Android API" tab. This tab is located in the bottom right corner of the page and it contains the descriptions of all the Android API methods recognized in the code snippet, so, in this case, it contains the description of the *BitmapFactory.decodeFile(...)* method. Now that she knows exactly what this code snippet does, she can proceed with the pattern definition by planning what a good description for this code snippet might be. For example, the user has to decide whether she wants to generate a verbose or a short description for the code, whether she wants to define sub-patterns for the parts of code composing the green snippet or directly a pattern to document the whole line (knowing that sub-patterns could be reused in future in the patterns definition), and considering the existence of patterns that already match the highlighted snippet.

Let's assume that the user wants to be very precise and verbose in the pattern definition. Also, she can define a sub-pattern for this snippet and so, after having defined it, she will have a pattern to reuse. Actually, there are two sub-patterns that can be defined here, but, to keep

this example short, she already has the definition of one of them. The two sub-patterns that a user can define are:

- A pattern for the same code snippet, but without the variable declaration (so the *BitmapFactory.decodeFile(...)* method). This will be the sub-pattern the user will define.
- A pattern for the *bitmapFile.getAbsolutePath()* method. This sub-pattern is already defined and the user will reuse it.

To define a sub-pattern for that method, she needs to select it and then click the "New snippet from selected lines" button. Since she does not want to define a pattern for the whole line, but just for a part of it, she will select the option to create a new wrapper code rather than maintaining the current method (see Figure 3.15).

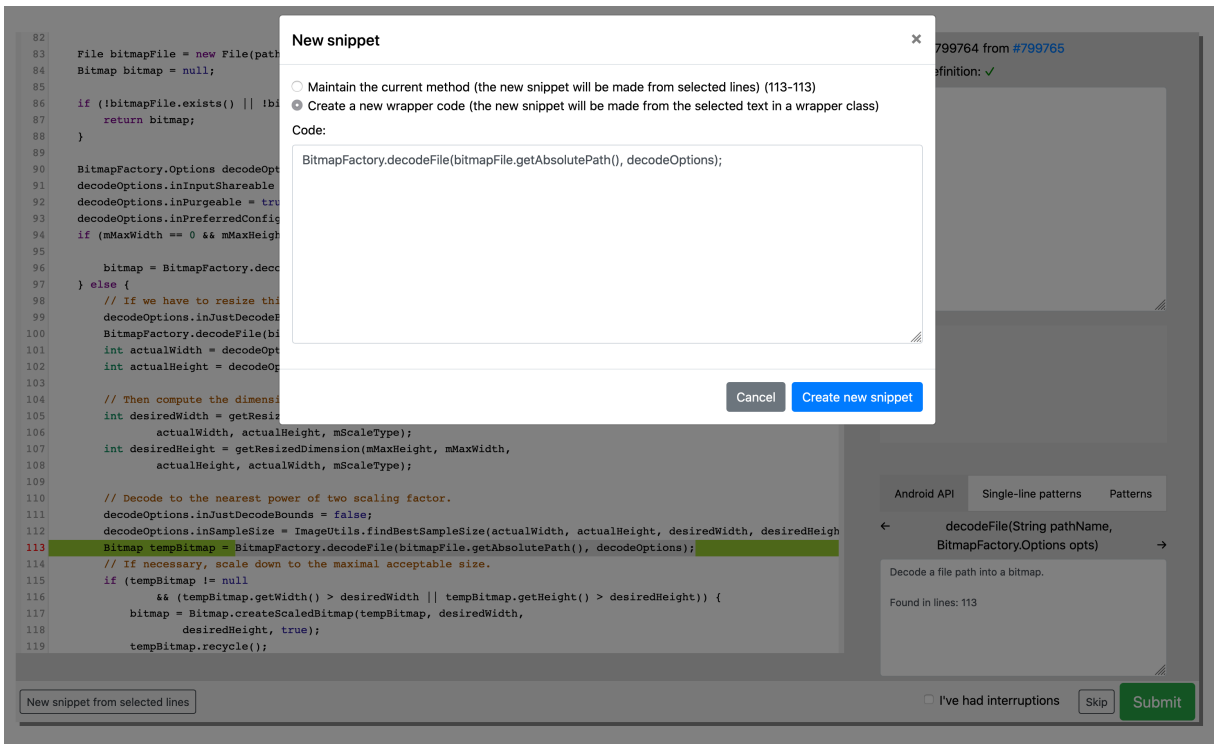

FIGURE 3.15: Creation of a new snippet from a part of the line of code.

After creating the new snippet, she will see the view in Figure 3.16.

```
1 public class test {
2 public static void main(String[] args) {
3 BitmapFactory.decodeFile(bitmapFile.getAbsolutePath(), decodeOptions);
4 }
5 }
```

FIGURE 3.16: Result of the creation of a new snippet using the wrapper code.

She can now start writing her pattern. For this piece of code a suitable documentation might be "Decode the bitmapFile's absolute path into a Bitmap using the decode options decodeOptions". In order to write this piece of documentation and make it flexible for other variable names, the user can click anywhere on the line of code and explore its AST (see Figure 3.17).

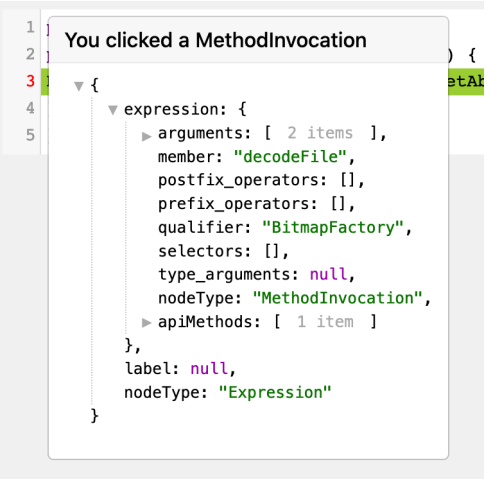

FIGURE 3.17: Clicking on a line of code reveals its AST.

She will start by defining the requirements: for this she will use the *member*, *qualifier* and *nodeType* nodes. She wants to make sure that our pattern will be matched only when the method is the *BitmapFactory.decodeFile(...)* method. For this she starts by defining three requirements (see Figure 3.18).

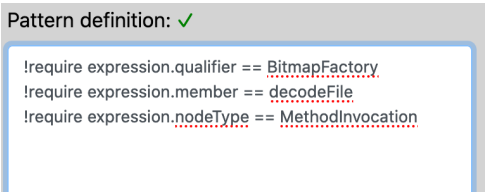

FIGURE 3.18: Example of requirements definition.

With these requirements the user will ensure that this pattern will only be applied to that particular method. Before writing the actual pattern, the user should check if there is a reusable pattern. By clicking on the red line number, she can see that a pattern for getter methods already exists (see Figure 3.19).

FIGURE 3.19: A reusable pattern for getter methods.

By clicking on it, she can copy the pattern into the pattern definition area and reuse it. After adding it, the preview of the pattern changes (see Figure 3.20).

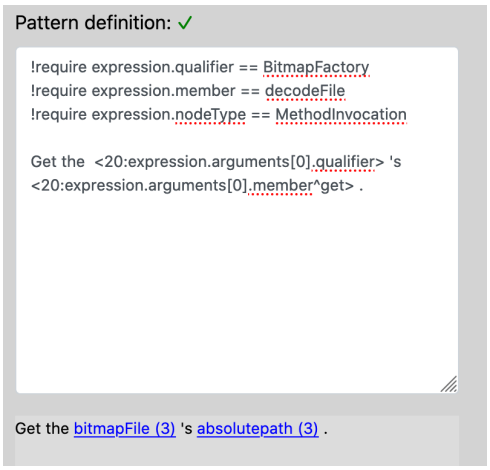

FIGURE 3.20: Preview of the reused pattern.

The user can now modify the natural language description in order to reach the pattern she planned to have, but, to do this, she also needs an additional node to get the *decodeOptions*. To find it, she clicks on the line of code and navigates to the second argument of the function and clicks on its variable name. Finally, the user has her first pattern declared, and she can see what the documentation automatically generated by it is (bottom part of Figure 3.21).

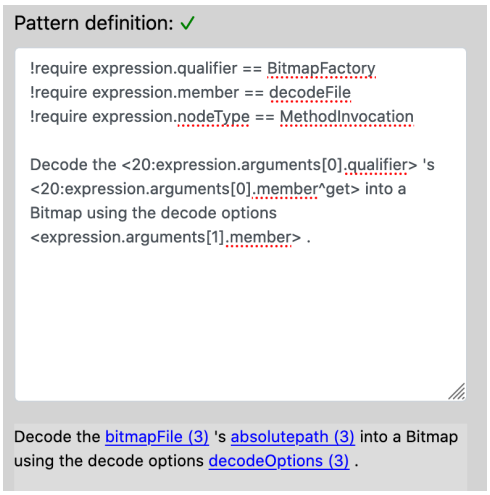

FIGURE 3.21: Preview of the first pattern.

At this point, the user submits the pattern and she can go back to the previous code snippet. Now, since she has declared another pattern, it can be found in her snippet's line, clicking on the red number will allow her to reuse the just-defined pattern.

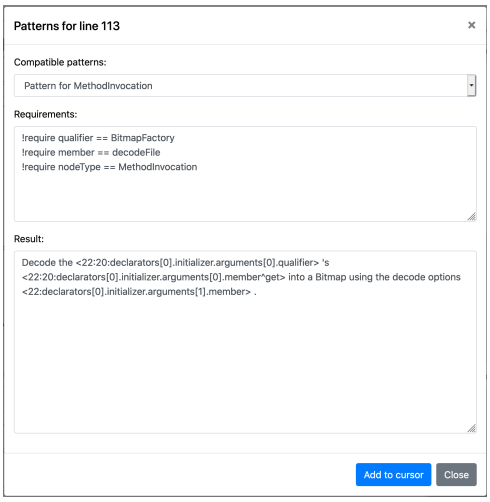

FIGURE 3.22: The window to reuse the pattern we just defined.

Since now the user is documenting a variable declaration, she might want to add details about it to her pattern. So, she may want something like: "Decode the bitmapFile's absolute path into a Bitmap using the decode options decodeOptions and save it into a Bitmap variable named tempBitmap". She has defined her pattern. It can be seen, along with its preview, in Figure 3.23.

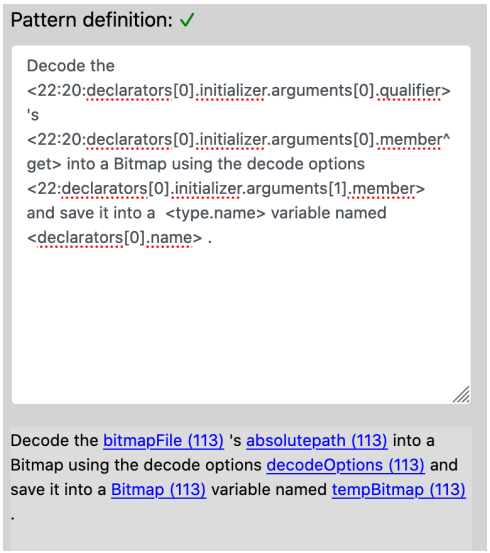

FIGURE 3.23: The finished pattern for our example.

3.7 Pattern Matching

After a substantial number of patterns have been defined, we can reuse them to document previously unseen code snippets or we can reuse them to define even more patterns. Both of those actions require a specific mechanism which is pattern matching (i.e., the ability to identify code snippets that can be documented by using the defined patterns). The difference between the two is that when reusing patterns the user can choose amongst all the matched patterns, while automatic code documentation requires the system to pick the best pattern(s) to use to generate the documentation. We will start this section by explaining how pattern matching works and, then, we will move to how the system generates the documentation.

Starting from a code snippet, the system attempts to match all existing patterns to it. The procedure is different if we are attempting to match a single-line pattern or a multi-line pattern.

3.7.1 Single-line pattern matching

The system splits the line based on the hierarchy of its AST nodes, and it attempts to match the pattern on every subset of the line (*e.g.,* the line is composed of three AST nodes A, B and C in this order, so that A contains B and C and B contains C; the system will attempt to match the pattern first on C, then on BC and then on the whole line ABC). The system iterates over all the lines of the code snippet and checks if the pattern's requirements are satisfied within that line or part of the line. To do that, it resolves the requirements' operands AST paths and gathers all the matches. Then, it compares all the matches of the left operand with the ones of the right operand to count how many times the condition was satisfied. Finally, it compares the count against the number of times the condition must be satisfied (1 by default). At this point, if all requirements are satisfied, the system attempts to check if all the node paths written in the pattern are resolvable within the line of code. If that is the case for every node path, the pattern is matched on that line.

3.7.2 Multi-line pattern matching

Multi-line pattern matching is a bit different: other than checking if a pattern matches, we are also interested in the intervals (start line, end line) where that pattern matches. Additionally, we are interested in *minimizing* the interval that the pattern covers, otherwise it would be matched on the whole snippet while it may refer to just two lines of it and, moreover, there might be multiple matches of the same multi-line pattern in a code snippet. To find the minimum interval and all the possible matches, we use a divide and conquer approach.

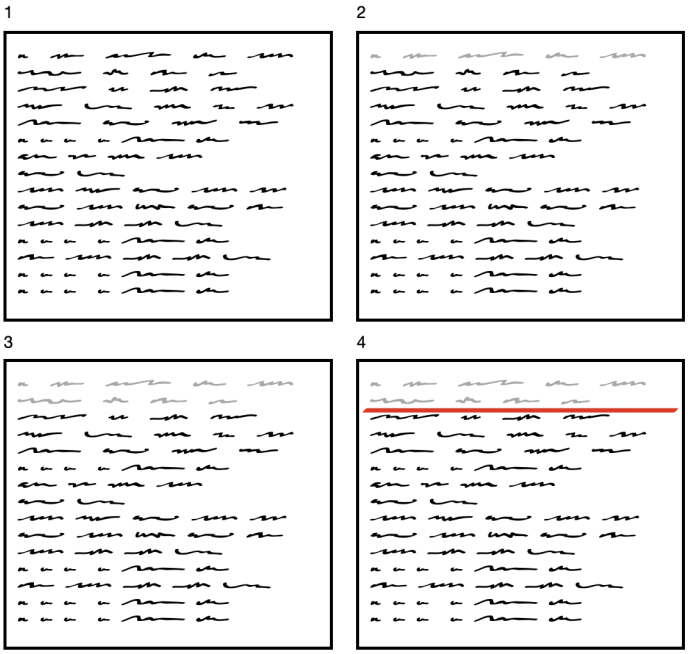

FIGURE 3.24: First steps of the interval matching algorithm.

Starting from the first line of the whole code snippet (number one in the Figure 3.24), we remove one line at the time and we attempt to match the pattern again (number two and three of Figure 3.24). When removing a line will prevent the pattern from applying to the snippet, we add that line back and save the line number (number four of Figure 3.24). Then, we repeat this process but, this time, we start from the last line of the code snippet instead of the first, until we reach the state represented in Figure 3.25.

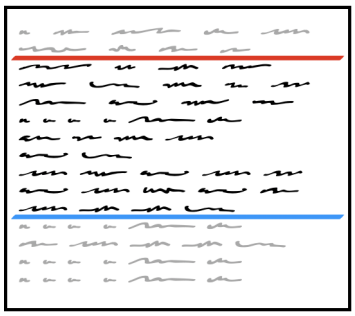

FIGURE 3.25: Last step of the interval matching algorithm before its recursion.

At this point we save the found interval (represented by the red and blue line in the figure) and we run the same algorithm again on two code snippets created using the found interval. The first code snippet goes from the first line of the current snippet to the line before the end interval line number (represented by the blue line). The second snippet, similarly, goes from the last line to the line after the start interval line number (represented by the red line). For a better understanding, the next figure shows a visualization of the two sub-snippets on which the algorithm will be called again.

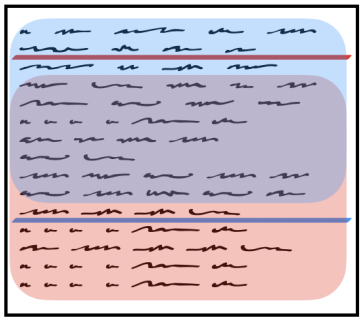

FIGURE 3.26: Recursion of the interval matching algorithm on two sub-snippet (red and blue rectangles).

We defined two stop conditions for the recursion. One is when the code snippet length (number of lines) becomes 1, the second is when the multi-line pattern cannot be matched on the whole sub-snippet. After the recursion finishes, we aggregate the results and we then have all the line number intervals where a multi-line pattern matches in a code snippet.

3.7.3 Pattern matching order

One last point about pattern matching that we should make is that patterns can *depend* on other patterns (*i.e.,* when a sub-pattern is used within a pattern or when the *!satisfies* keyword is used). This means that, while doing pattern matching on a particular code snippet, we should also check for any pattern dependency. Since there cannot be circular dependencies amongst patterns (once a pattern is created it cannot be modified and a new pattern can only depend on already existing patterns), our approach to the problem is to attempt to match every pattern to the snippet and save the result of the matching. We use a queue: if a pattern can be checked for matching because all of its dependencies have already been checked (or it has no dependencies), it will be checked and saved. If a pattern cannot be checked due to one of its dependencies not having been checked yet, it will be put to the end of the queue. In this way, after some iterations, all the pattern will be matched and we can get our result. This algorithm should always work on our system (we have no circular dependencies), but, if for any reason we will have inconsistencies in our data, we defined a maximum number of iterations that can be made before stopping and discarding the remaining, not yet matched, patterns.

Chapter 4

Empirical Study Design

The goal of this study is to evaluate POET in terms of its ability to generate code comments for previously unseen code snippets. The context of the study is represented by defined patterns, train snippets, test snippets, mined applications and extracted methods. We compare our approach with two state-of-the-art techniques (retrieval-based, deep learning based). The two techniques we choose to compare POET with are ADANA [1], a retrieval-based approach which has been already mentioned in the State of the art chapter, and a Deep Learning implementation based on the Transformer architecture developed by Vaswani et al. [16] that we deemed representative of the current Deep Learning state of the art.

4.1 Research Questions

We aim at answering two research questions:

- RQ1: What is the coverage ensured by POET when documenting previously unseen code as compared to other competitive techniques?
- RQ2: How meaningful are the comments generated by POET as compared to other competitive techniques?

The objective of the first question is to evaluate the coverage, which is the percentage of code snippets in the test set for which it is possible to generate documentation. For this question, we are not interested in the *correctness* of the generated documentation, but only in whether a documentation can be generated or not. So, for the first question, we observe the results of POET from a quantitative point of view. We will, then, do the same for the other two state-of-the-art approaches previously mentioned and, to answer the research question, we will compare the results.

The objective of the second question is, instead, to assess the quality of the generated comments, which is how well a generated comment represents the code used to generate it (i.e., the one to document). So, for this question, we will only perform a qualitative check, not taking into account the number of generated comments, but focusing on their quality.

4.2 Context Selection

The selection of the context for this study can be divided into four main steps:

1. Applications mining
2. Methods and snippets extraction
3. Training-Test set split
4. Patterns definition

We have already described the mining, extraction and definition steps in previous chapters, but we will summarize them here with the addition of quantitative data about each step.

4.2.1 Applications mining

As previously mentioned, we created a script that was able to mine the F-Droid applications catalogue. We used that script to mine the applications, because we F-Droid collects Android repositories of real apps, most of which are published in the Google Play market. While the script was able to clone every F-Droid repository, we chose to stop it after having reached a number of 300 repositories. This is due to the fact that consequent preprocessing steps' computation time increases with the number of repositories cloned (see the preprocessing chapter). Additionally, the selected number of repositories is definitely enough to perform a preliminary evaluation of POET.

4.2.2 Methods and snippets extraction

The methods and snippets extraction was also made using a script, which parsed every Java file found in the previously extracted repositories. We have already seen the different types of snippets that we defined in the Methods and Snippets extraction section (3.1.2). In Table 4.1, we can see some statistics about the data that we were able to get after completing this step.

Full method	222,204
Block	250,582
Empty line	178,565
Empty line in block	279,629
Total	**930,980**

TABLE 4.1: Number of snippets generated, divided by type.

As we can see, the number of snippets is very high, especially considering that they come from just 300 repositories. We can say that, on average, a single repository contributes to the generation of ~3,000 code snippets. Table 4.2 depicts more data about extracted methods and snippets, considering that the total number of Java files from the repositories that were able to generate at least one snippet is 24,167.

	Methods	Snippets
Total Number	221,050	930,980
Average length (LOC)	9.21	7.28
Average for file	9.14	38.52
Average for repository	736.83	3,103.26

TABLE 4.2: Table showing general data about extracted methods and snippets.

4.2.3 Training-Test set split

As we have seen in the preprocessing chapter, after having extracted all the snippets, we proceeded to create clusters based on code similarity. In total, we built 18,717 clusters. The code snippets which could not be put into a cluster (i.e., singleton clusters) were discarded. To generate the train and test sets, we then decided to analyze the centroid of each cluster and create groups based on used Android APIs. For example, all the centroids which invoked a function from the Android API *android.graphics.BitmapFactory* were put in the same list. If a code snippet invoked two or more APIs, it was put in both list. In Table 4.3, we can see the API classes that were used the most in our centroids.

API class name	**Number of centroids**
android.util.Log	1,324
android.view.View	204
android.net.Uri	204
android.widget.Toast	190
android.preference.PreferenceManager	87
android.graphics.BitmapFactory	55
android.os.Environment	53
Others	665
Total	**2,782**

TABLE 4.3: Table showing the most used Android APIs in our centroids and the total number of used APIs in our centroids.

Not surprising, the most used API class is the one dedicated to logging. This process was made in order to allow us to pick some of those APIs and attempt to generate patterns and documentation for them. We felt unrealistic, due to time limitations, for us to be able to write patterns for every centroid, so we chose to pick a small subset of them based on which APIs they use. In this way we could write patterns for a cohesive set of code snippets. The choice of which API we wanted to analyze was made following two main factors: the number of centroids, in order to have big enough train and test sets, and their usage. This choice was made after seeing the most used class: the usage of the *android.util.Log* class in Android application is often limited to just a few methods, very similar and with the same function (*i.e.,* write to the debug log, write to the warning log, write to the error log, *etc.*). For the same reason, we discarded

android.net.Uri and *android.widget.Toast*. At the end, we decided to use the centroids invoking *android.view.View* APIs and *android.graphics.BitmapFactory*. The reasons for this choice are:

- *android.view.View*: it is the second most used API and it is used everywhere in the Android environment. Additionally, it can be used in many different ways together with other Android widgets such as TextView.
- *android.graphics.BitmapFactory*: like View it also is amongst the most used APIs and it is widely used to show images. The choice of this class was mostly driven by the idea of having a class with a more precise usage than View.

Having chosen the set of centroids for which we wanted to create patterns, what we had to do at that point was to split them into two sets: the train set and the test set. Before actually performing the splitting, we shuffled the data in order to obtain randomized results. Then, we split the data, keeping 80% of it for the train set and dedicating the last 20% of it for the test set. The resulting number can be seen in Table 4.4.

	view.View	**graphics.BitmapFactory**	**Total snippets**
Train set snippets	163	44	207
Test set snippets	41	11	52
Total snippets	**204**	**55**	**259**

TABLE 4.4: Resulting snippet number for the train and test set.

4.2.4 Patterns definition

After the train set was built, we loaded it into the web application so that it started to serve the train set's snippets for us to write patterns upon. We first defined a substantial number of small single-line patterns using the "new snippet" feature of the web application in order to be able to reuse them for other patterns. Then, we kept defining patterns until we had processed every training set's snippet. Table 4.5 shows the number of defined patterns, divided by type.

Single-line patterns	661
Multi-line patterns	56
Total	**717**

TABLE 4.5: The number of defined patterns by type.

As we have seen in previous chapters, the web application can record the time spent on a pattern definition and whether the user had interruptions during the pattern definition process or not. Using that data, we were able to calculate how much time was spent on pattern definition, the average time required to write a pattern and the number of patterns defined with and without interruptions. This data is available in Table 4.6. The table references single-line patterns as SLP and multi-line patterns as MLP.

Type	Number	Total time	Average time
SLP without interruptions	649	24:07	00:02
MLP without interruptions	54	03:23	00:03
SLP with interruptions	12	04:03	00:20
MLP with interruptions	2	00:36	00:18
Total	**717**	**32:09**	**00:02**

TABLE 4.6: Effort spent in pattern definition. The time format is *hh:mm*.

From the data, we can state that most of the patterns were defined without interruptions and that, on average, defining a pattern usually takes less than four minutes. We deemed this number of pattern sufficient for a first attempt at documentation generation. A higher number of patterns, however, should lead to better results when the documentation is generated. As a last note, due to the small sample size, we believe the average time to define a pattern with interruptions to be different from the one reported.

4.3 Data Collection and Analysis

We will now see more in detail how the two chosen state-of-the-art techniques work and how we were able to use them to obtain the documentation for our test set's snippets. Finally, we will see how we analyzed the results to answer the two research questions posed at the beginning of this chapter.

4.3.1 Competitive Techniques

As mentioned previously, we will now see how we used ADANA, a retrieval-based approach for documentation generation, and a Deep Learning approach based on transformers to generate documentation for our test set. We will also describe them more in detail.

ADANA

ADANA, as already described in the State of the art chapter, is an Android Studio plugin which is able to automatically generate documentation for a selected code snippet. To do that, the creators of the tool, built a knowledge base from Stack Overflow and Github Gist made of pairs *<code snippet, description>*. The plugin then exploits this knowledge base with the help of ASIA, a code clone detector developed by the same authors. Using the plugin, the user can select a code snippet and the plugin will attempt to find a code clone of the selected snippet in the knowledge base. If a clone is found, the associated description extracted from the Stack Overflow and Github Gist discussions will be automatically written in the form of a code comment above the selected snippet.

Deep Learning

The Deep Learning approach, as stated previously, is transformer-based, and, as any other Deep Learning implementation, it requires a huge training dataset to work properly. We picked the

CodeSearchNet Challenge[1] Java dataset which, after some data cleaning, like duplicates and empty strings removal, had more than 300,000 code snippets along with their description. After the data cleaning, we transformed it in the data format that FastAI, the AI framework we are going to use to train the model, requires. To do that, since what we want to achieve is to transform a code snippet into the code's documentation string, we will create a Sequence-to-Sequence Databunch (that is the format FastAI uses for managing and loading data for training and evaluation). Then, we trained Byte Pair Encoding (BPE) models in order to allow FastAI to automatically tokenize our data. The BPE models are trained on a fraction of the train set. FastAI's tokenizers is also able to do some additional processing such as lower casing all words, removing repetitions and more. After this preprocessing, we are able to use our Transformer model. Before training, we used the Learning Rate finder provided by FastAI in order to find the best fit for the training hyperparameters, using the BLEU score to evaluate the various fits. Then, finally, we were able to train our model. We chose ten as the number of epochs, but used an early stopping callback with a value of patience of three and minimum delta of 0.01. This means that the training would stop if three consequent epochs does not improve the valid loss by a value greater than 0.01.

4.3.2 Generating Comments

The documentation generation for both approaches was very straightforward.

We were able to evaluate ADANA by using its latest release, available online[2]. We downloaded, installed it and opened every file from which the test set's code snippets were extracted. At this point, what we did was to manually select the code snippets in the files and run the ADANA plugin on them. We then saved the results.

Running the Deep Learning approach was a bit less trivial: we used a Jupyter notebook[3] as a starting point and, with enough modifications, we were able to get the resulting documentation for the snippets of our test set. In particular, we removed some filtering on the data, increased the sample size, the number of epochs and replaced the test set used by the notebook with ours. One curious difficulty that we had to overcome was that the approach was not working unless we increased the number of the test set's elements. To solve this issue, we simply multiplied our test set's elements by duplicating it.

[1] https://github.blog/2019-09-26-introducing-the-codesearchnet-challenge/

[2] https://github.com/emadpres/ADANA

[3] https://nathancooper.io/i-am-a-nerd/deep_learning/software_engineering/2020/03/07/How_to_Create_an_Automatic_Code_Comment_Generator_using_Deep_Learning.html

4.3.3 Data Analysis

Now, we can explain how the results obtained from each technique were analyzed in order to answer the two research questions.

Quantitative results

The first research question is about the coverage of the approaches, which means we reviewed, for each approach, how many code snippets were documented. By construction, we know that the Deep Learning approach always returns an output, thus its coverage will be 100%. ADANA, instead, can return an empty documentation to the user if a suitable code clone is not found within its knowledge base. For POET, we consider as documented any code snippet to which at least one pattern was applied and a code comment was successfully generated. So, after generating the comments, we manually count, for each approach, how many code snippets were successfully documented from each approach and write down the findings. The results of this process are reported in the Results Discussion chapter.

Qualitative results

The second research question focuses on the quality of the generated comments. Since, as shown later, the results of the quantitative analysis showed that ADANA was only able to generate comments for 7 snippets (due to the lack of code clones in the dataset), we decided to focus the comparison in this part only on the Deep Learning approach.

We have manually reviewed the output of the two approaches by giving every generated documentation a mark from one to five based on how much a comment is related to the code from which it was generated. To assign a mark we applied the following guidelines:

- **1**: A mark of one indicates that the generated comment is unreadable or completely unrelated to the documented code snippet.
- **2**: A mark of two indicates that the generated comment was somewhat readable and related to the documented code snippet, but missed the point of the code or described only a small part of it.
- **3**: A mark of three indicates that the generated documentation describes with enough accuracy at least half of the code, but there are still some readability issues or part of the documentation is still unrelated to the code.
- **4**: A mark of four indicates that the generated documentation accurately describes the majority of the code. There are still minor issues with the documentation, but it is understandable and related to the code.
- **5**: A mark of five indicates that the generated documentation accurately describes the whole code snippet, has no readability issues and it is completely related to the code snippet.

The qualitative results from this analysis are reported in the Results Discussion chapter.

4.4 Replication Package

The replication package is available online at: https://github.com/Gargarensis/MSCThesis. It contains the whole system ready to be used. To do that, clone the repository and execute the preprocessing scripts in the order shown Chapter 3. We suggest to stop the mining of Android repositories when you have reached around 300 repositories, otherwise the time required for the preprocessing can become very long. Then, to run the server, one can navigate to the *webserver* directory and run the command *npm run start*. In order to work, the web server requires Node.js and NPM. After the server finishes to load, one can navigate to the URL *localhost:3000/webapp/index.html* and start to use the web application, unless some modifications were made to the port or if the web server is hosted somewhere else. In order to run POET on any snippet of the test set, one can use the query parameter *?snippetId=ID* to navigate to a specific snippet of the test set using its identifier and generate the documentation. Otherwise, simply navigating to the web application home page will serve a test set snippet, unless one modified its settings to behave differently.

The replication package also contains a Python notebook which can be run with Google Colab or Jupyter in order to replicate the deep learning implementation, along with our used train and test sets and our defined patterns.

Chapter 5

Results Discussion

In this chapter we discuss and compare the results obtained using the three approaches mentioned in the Empirical Study Design chapter.

5.1 Quantitative Analysis

In this section we report the quantitative findings of the three experimented approaches. In particular, we compute the coverage for each of the approaches and compared them. In addition, we manually reviewed every generated documentation and assigned a mark to it, from one to five, based on its quality.

To compare the coverage of the three approaches, we can take a look at Table 5.1 that shows, for each approach, how many snippets were documented and how many snippets were left without documentation. For POET, we considered documented a code snippet if at least one pattern was applied to it and, thus, at least one comment was generated.

Approach	Documented	Not documented
POET	52	0
ADANA	7	45
Deep Learning	52	0

TABLE 5.1: Coverage of the three experimented approaches.

From these results, we can say that POET is able to ensure a full coverage when documenting previously unseen code and it is at the same level of the current state-of-the-art Deep Learning approach while having a definitely better coverage than the retrieval-based approach. This last remark may be due to the fact that ADANA is based on a limited database created two years ago likely to be outdated due to the rapid evolution of the Android ecosystem. This could be the cause for the limited coverage (14%). Given the very low number of data points for this approach (only 7) we do not consider it in the manual analysis of the generated comments.

After those observations about the coverage of the approaches, we can analyze the results of the qualitative evaluation of the generated documentation, comparing POET with the Deep Learning based approach. While doing this evaluation, we followed the guidelines described in the Empirical Study Design section to assign a mark to every generated documentation. The distributions of marks for both approaches are depicted in Figure 5.1.

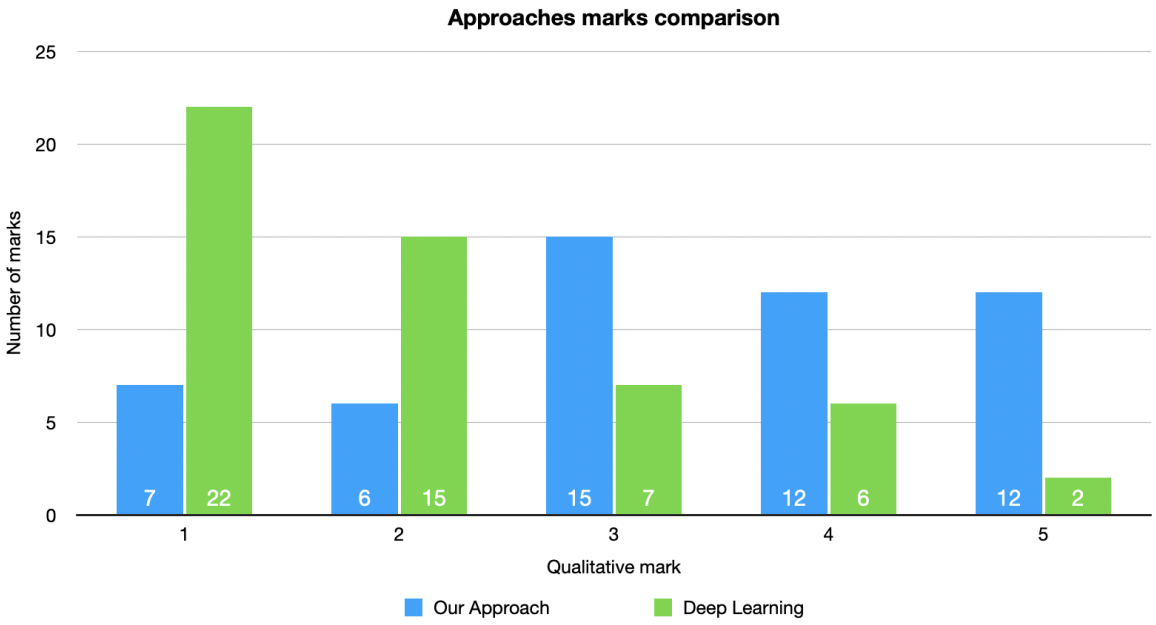

FIGURE 5.1: A diagram showing the evaluation differences between POET and the Deep Learning approach.

Reading Figure 5.1, we can make the following observations:

- Most of the Deep Learning results are not readable or not related to the code.
- Most (75%) of POET results are at least somewhat related to the code (3+).
- Only 25% of the documentation generated by POET was rated 1 or 2 as opposed to the 71% of the Deep Learning approach.
- Almost half (46%) of the documentation generated from POET was rated 4 or 5.
- The average mark for POET is 3.3, while the average mark for the Deep Learning approach is 2.05.

Observing the results, we can say that POET generates in several cases comments that are at least related to the code and almost half of the times they are meaningful. Basically, POET showed its superiority with respect to the deep learning based technique but, it is important to highlight, that the deep learning based approach is fully automated, while POET requires a substantial effort for the patterns definition.

5.2 Qualitative Analysis

For a deeper qualitative analysis, in this section we selected a sample of good and bad documentations generated by POET. We also describe why we regard them as "good" or "bad" and, in the latter case, how their issues can be solved by improving the approach.

5.2.1 Try and Catch Documentation

```
1
2
3   /*  Attempts to set the text of the view app_name to  appNames . In case of error, prints its stack trace.  */
4   // Get a view inside view with resource ID " app_name " and save it in the variable  txtAppName of type  TextView .
5   TextView txtAppName = (TextView)view.findViewById(R.id.app_name);
6   try{
7       // Set the  txtAppName  text to  appNames  .
8       txtAppName.setText(appNames[position]);
9   }catch(Exception e){
10      // Print the e stack trace to the standard error stream.
11      e.printStackTrace();
12  }
13
14
```

FIGURE 5.2: A code snippet documented by POET containing a Try and a Catch block.

The code snippet shown in Figure 5.2 has been documented using POET. All the comments shown in the code snippet were generated by POET. It is possible to distinguish between multi-line and single-line patterns applied to this code by looking at the type of comment used: inline comments (*// ...*) are used for single-line patterns while multi-line comments (*/* ... */*) are used for multi-line patterns. This is the convention used for all snippets described in this section.

During our qualitative evaluation, we assigned to the documentation generated in Figure 5.2 a mark of 4. As we can see from Figure 5.2, every pattern applied to the code snippet was able to generate a comment which reflects exactly what the code does. In particular, the multi-line pattern does a very good job at describing what the whole snippet does. However, there is a detail that prevented us to assign a grade of 5 which is that both the multi-line comment at line 3 and the single-line comment at line 7 fails to acknowledge that the text of *txtAppName* is being set to a particular element of the array *appNames* and not to *appNames* itself. To solve this issue there are two possibilities:

- **More patterns**: the system did apply the most suitable patterns for this code, which means that a similar pattern with an array element instead of a variable was not available. To solve this problem and obtain a better documentation, we can define those patterns.
- **Improve the approach**: as we will see in the Future Work chapter, we propose some different ideas to improve POET. In particular, to solve this problem, we can add the possibility to reuse patterns automatically instead of writing a new pattern which combines more patterns. In this case, we should be able to define a pattern for *appNames[position]* which should translate the code to the documentation "An element of appNames at index: position". At this point we should modify the *setText* pattern to allow it to use any pattern matched in its argument instead of just the variable name. Improving the approach in this way will combine the patterns to obtain: "Set the txtAppName text to an element of appNames at index: position".

Nonetheless, we found the documentation quite accurate and well written. Additionally, every line of the code snippet was commented.

5.2.2 If and Else Documentation

```
1
2           // Check if transparentMenusEnabled is true.
3           /*  If transparentMenusEnabled is true, set the   view  alpha to  0.73f . Otherwise, set the view  alpha to  1.0f .   */
4           if (transparentMenusEnabled) {
5               // Set the  view  alpha to  0.73f .
6               view.setAlpha(0.73f);
7           } else {
8               // Set the  view  alpha to  1.0f .
9               view.setAlpha(1.0f);
10          }
11      }
12  }
13
```

FIGURE 5.3: A code snippet documented by POET containing an If statement with an Else block.

A mark of 5 was assigned to the documentation of Figure 5.3 due to its completeness, coverage and accuracy. The comments are well written and explain exactly what the code snippet does. We do not think there is room for improvement on this particular snippet. This is also due to the simplicity of the involved code snippet.

5.2.3 Long Code Snippet Documentation

```
1
2           // Creates a new OnLongClickListener in the productLongClickListener variable.
3           final OnLongClickListener productLongClickListener = new OnLongClickListener() {
4               public boolean onLongClick(final View v) {
5                   // Save the  into the variable  builder .
6                   final DialogBuilder builder = DialogBuilder.get(DirectionsActivity.this);
7                   // Set the  builder  title to  directions_products_prompt  .
8                   builder.setTitle(R.string.directions_products_prompt);
9                   // Set the  builder  items to  directions_products  .
10                  builder.setItems(R.array.directions_products, new DialogInterface.OnClickListener() {
11                      public void onClick(final DialogInterface dialog, final int which) {
12                          // Iterate over each element (  view  ) of the   's  viewProductToggles .
13                          /*  Iterate over each element of viewProductToggles and change each element checked based on the value of which .   */
14                          for (final ToggleImageButton view : viewProductToggles) {
15                              // Check if  which   is equal to 0 .
16                              /* If which   is equal to 0 , change the  view  checked .   */
17                              if (which == 0)
18                                  // Set the  view  checked to  v  .
19                                  view.setChecked(view.equals(v));
20                              // Check if  which   is equal to 1 .
21                              /* If which   is equal to 1 , change the  view  checked .   */
22                              if (which == 1)
23                                  // Set the  view  checked to  v  .
24                                  view.setChecked(!view.equals(v));
25                              // Check if  which   is equal to 2 .
26                              /*  If which  is not equal to 2 , set the  view  checked to  true .   */
27                              if (which == 2)
28                                  // Set the  view  checked to  true .
29                                  view.setChecked(true);
30                              // Check if  which   is equal to 3 .
31                              /*  If which  is not equal to 3 , set the  view  checked to  "SUTBP" .   */
32                              if (which == 3)
33                                  // Set the  view  checked to  "SUTBP" .
34                                  view.setChecked("SUTBP".contains((String) view.getTag()));
35                          }
36                      }
37                  });
38                  // Show the  .
39                  builder.show();
40                  return true;
41              }
42          };
43          // Iterate over each element (  view  ) of the   's  viewProductToggles .
44          for (final View view : viewProductToggles)
45              // Set the  view  onlongclicklistener to  productLongClickListener  .
46              view.setOnLongClickListener(productLongClickListener);"
47
```

FIGURE 5.4: A long code snippet documented by POET.

Figure 5.4 shows the documentation of a larger code snippet to which we assigned a mark of 3. From the figure, we can see that many comments are poorly written (like the one at line 5, 31, 33 and 38). Nonetheless the rest of the documentation describes pretty well what the respective lines of code do. In particular, the multi-line pattern at line 13 describes very well a big part of the code snippet, including a the loop and multiple conditional statements. An issue with this snippet is that, unlike the previous two snippets, there is no high-level comment which summarize what the whole snippet does. A possible solution to use POET to generate such comment in this particular case would be to improve the documentation generator in a way that it would be able to recognize listeners initialization and to concatenate existing patterns to create a cohesive documentation for the listeners. This would not, however, solve completely the problem: even with such features, the generator would still be missing a link between the first part of the code snippet (the *OnLongClickListener* declaration) and the second part (the loop that assigns the listener to each view). To solve this problem, the correct thing to do (since we are proposing a pattern-based approach) is simply to define a new pattern that, right now, is clearly missing.

5.2.4 Full Method Documentation

```
1
2   public void onItemViewClick(final View view, final int position) {
3       // Check if    position  is not equal to   ListView  INVALID_POSITION .
4       if (position != ListView.INVALID_POSITION) {
5           // Save the mAdapter 's  checkeditemcount into the variable  count .
6           int count = mAdapter.getCheckedItemCount();
7           // Get the view 's  id .
8           switch (view.getId()) {
9               case android.R.id.icon:
10                  // Check if  count   is equal to 0 .
11                  if (count == 0) {
12                      // Start the  supportactionmodeifneeded .
13                      mMultiChoiceHelper.startSupportActionModeIfNeeded();
14                      // Sets the checked state of the  position  position to  true .
15                      mMultiChoiceHelper.setItemChecked(position,true, true);
16                  } else {
17                      // Sets the checked state of the  position  position to  isItemChecked .
18                      mMultiChoiceHelper.setItemChecked(position,
19                              // Invoke the  isItemChecked function with Array of 1 arguments.
20                              !mAdapter.isItemChecked(position), true);
21                  }
22                  break;
23
24              case R.id.button_popup:
25                  // Cause the new Runnable with type Runnable  to be added to the view message queue.
26                  /*  Adds a Runnable to the view message queue that  shows the popupmenu at the position  position .    */
27                  view.post(new Runnable() {
28                      @Override
29                      public void run() {
30                          // Show the popupmenu .
31                          showPopupMenu(view, position);
32                      }
33                  });
34                  break;
35          }
36      }
37  }
38
39
```

FIGURE 5.5: A code snippet of a method documented by POET.

Figure 5.5 shows another code snippet, whose documentation was given a mark of 3. The reason for this is because there are not multi-line snippets that can summarize the snippet and other minor issues. For example, the comment at line 12 is not really readable due to the *supportactionmodeifneeded* not being split by word. This is a recurring problem in POET but, luckily, it can be solved by improving the documentation generator to allow it to split such words into their single components. Additionally, at line 7, the system ignored the fact that the method it documented was inside a switch statement. A good generated comment, instead, is the one at line 26. It does not only describe what the *view.post* method does, but also what the Runnable does.

5.2.5 If Statement Documentation

```
1
2               // Set the  colorEdit  text to  getHexString  .
3               colorEdit.setText(Utils.getHexString(getStartColor(initialColor), isAlphaSliderEnabled));
4               // Set the  colorPickerView  coloredit to  colorEdit  .
5               colorPickerView.setColorEdit(colorEdit);
6           }
7           // Check if isPreviewEnabled is true.
8           /* If isPreviewEnabled is true, set this object visibility to  GONE .  */
9           if (isPreviewEnabled) {
10              // Inflate a view from the XML resource  color_preview
11              /*  Get a view with resource ID " context " and change its  visibility .  */
12              colorPreview = (LinearLayout) View.inflate(context, R.layout.color_preview, null);
13              // Set the  colorPreview  visibility to  GONE  .
14              colorPreview.setVisibility(View.GONE);
15              // Adds a  view to  pickerContainer .
16              pickerContainer.addView(colorPreview);
17
18
19
```

FIGURE 5.6: A code snippet of an If statement documented by POET.

Figure 5.6 shows a code snippet that was likely split by new lines: we can notice this detail due to the fact that we can see the end of the block that contains the first two lines of code. Nonetheless, the generated documentation for this code snippet was given a mark of 3. This is because, except the comment at line 2, every generated comment is well written and related to the code. In particular, we can see a summary of the if statement at line 8 and of the operation on the *colorPreview* view at line 11. Apart from the problem exposed before, there are two other problems with this documentation and both of them are related to the line 8 comment. The first problem is that, while it describes one of the operations done inside the if block, it is missing the last one, which is the addition of the view *colorPreview* to *pickerContainer*. The first problem is, then, that the multi-line snippet does not document the whole block but just a part of it. Another, smaller problem is that we are not changing *this object* visibility as the comment states, but the visibility of the *colorPreview* view. The only way to solve those 2 problems, however, would be to define more patterns. Instead, to solve the problem of the documentation at line 2, apart from defining more patterns, we could improve the approach with the pattern reusability modifications explained in the first snippet of this section and further elaborated in the Future Work section.

5.2.6 Part of Method Documentation

```
1
2   public final void openAppDrawer(View view, int x, int y) {
3       if (!(x > 0 && y > 0) && view != null) {
4           // Creates a new int in the pos variable.
5           int[] pos = new int[2];
6           // Invoke the  getLocationInWindow function with Array of 1 arguments.
7           view.getLocationInWindow(pos);
8           // Set the  's cx  to  's pos .
9           cx = pos[0];
10          // Set the  's cy  to  's pos .
11          cy = pos[1];
12
13
```

FIGURE 5.7: A code snippet containing the first part of a method documented by POET.

Figure 5.7 shows the documentation generated by POET that received the lowest score. We assigned it a grade of 1. There are many problems found in the documentation of this code snippet, so we compiled a list of all of them along with a way to solve them:

- **Missing documentation for line 3**: the solution for this problem would be either to define a pattern for that statement or to improve the documentation generator to improve the patterns reusability in the case of boolean formulas. We discuss this improvement more in depth in the Future Work chapter.
- **Wrong documentation for line 5**: the documentation at line 4 does not take into account that we are initializing an array of integers and not a simple integer. To solve this problem, we should define more patterns to cover this and similar cases.
- **Unexpected pattern for line 7**: the documentation generated at line 6 uses the pattern for a generic method invocation while we are sure to have an already defined a pattern for getter methods. The problem, in this case, is that the pattern for getter methods has a requirement that states that the number of parameters of the function must be equal to zero. This means that the system could not match that particular pattern in this situation. This is not wrong, because the correct documentation here would be "Get the location in the window at the position pos" instead of simply "Get the location in the window". However, we are stuck with a bad comment with the only solution being to define a new pattern for this code.
- **Bad documentation for line 9 and 11**: there are two problems with the generated documentation for line 9 and 11. The first problem is that both of them are missing a qualifier and the pattern that is applied expects a qualifier (see the *'s*, it expects something like *object 's cx*). This problem can be solved by improving the pattern definition to support optional keywords or to check for empty fields in the AST nodes. The first solution is discussed more in detail in the Future Work chapter. The second problem is the same problem found in the documentation for line 5: the applied pattern does not take into account the fact that *pos* is an array. To solve this problem we should just define a new pattern for this situation.

5.2.7 End of Method Documentation

```
1
2                                   // Save the entry 's  value into the variable  emoji .
3                                   Sticker emoji = entry.getValue();
4                                   // Set the  spannable  span to  a new  ImageSpan .
5                                   spannable.setSpan(new ImageSpan(context, BitmapFactory.decodeStream(emoji.res.getAssets().open(emoji.assetUri.getPath()))),
6                                           // Start the   .
7                                           matcher.start(), matcher.end(),
8                                           Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
9                               }
10                          }
11                  }
12                  return hasChanges;
13          }
14
15
```

FIGURE 5.8: A code snippet containing the end of a method documented by POET.

Another example of sub-optimal documentation can be seen in Figure 5.8. This documentation received a 1 as mark. This is due to multiple issues:

- Even though the documentation generated for line 5 is technically correct, we consider it a bad comment due to the fact that it does not provide any relevant information about line 5 at all. We believe that the fact that we are initializing the *ImageSpan* with a Bitmap from a stream should be included in the documentation in order for it to be relevant. To fix this problem, we should either define new patterns for such cases or we should improve the pattern definition with a better pattern reusability. This problem was already discussed in a previously analyzed snippet.

- The comment at line 6 should not be there. It is there because POET attempts to generate a comment for every line of code where a line is distinguished from the previous when separated by a newline. The problem is that the documented line (line 7) is, following the Java syntax, part of line 5 and, thus, should not be documented on its own. To solve this problem we should either modify the documentation generator to remove such comments or to avoid writing them in the first place. It is important to notice, however, that, when generating the documentation for line 5 or any other similar line (which spans multiple lines), the AST is always the same and so the generated pattern for that line will be correct (it will consider the lines as a single line). The same problem would have presented while documenting line 8, but, since there is only a variable in that line, no pattern was found and thus no comments were generated for that line.

- The return at line 12 is ignored. This is because the documentation generator does not use Java keywords while generating the documentation. Also, no patterns were matched on that line because none were defined for a line where the only available code is a single variable.

5.2.8 Repeated Code Documentation

```
1
2   // Get a view inside view with resource ID " range0_picker_number " and save it in the variable  pickerValue0 of type  NumberPicker .
3   NumberPicker pickerValue0 = (NumberPicker) view.findViewById(
4               R.id.range0_picker_number);
5   // Get a view inside view with resource ID " range0_picker_unit " and save it in the variable  pickerUnit0 of type  NumberPicker .
6   NumberPicker pickerUnit0 = (NumberPicker) view.findViewById(
7               R.id.range0_picker_unit);
8   // Get a view inside view with resource ID " range1_picker_number " and save it in the variable  pickerValue1 of type  NumberPicker .
9   NumberPicker pickerValue1 = (NumberPicker) view.findViewById(
10              R.id.range1_picker_number);
11  // Get a view inside view with resource ID " range1_picker_unit " and save it in the variable  pickerUnit1 of type  NumberPicker .
12  NumberPicker pickerUnit1 = (NumberPicker) view.findViewById(
13              R.id.range1_picker_unit);
14  if (pickerUnit0 != null && pickerUnit1 != null && pickerValue0 != null
15              && pickerValue1 != null) {
16          // Set the    uppickers to  view  .
17          setupPickers(view, pickerValue0, pickerUnit0, pickerValue1, pickerUnit1);
18  }
19
```

FIGURE 5.9: A code snippet with many duplicated lines documented by POET.

The last of the code snippets we are reviewing is the one in Figure 5.9, which we graded with a mark of 2. The reason for this choice is that, even if all the variable declarations are correctly commented, we cannot really extract any information from it. This is because the last two lines (14 and 17) are the ones that define what this code snippet does. The first was not documented, the second, instead, received a comment which clearly comes from the wrong pattern. Line 14 was not documented due to the fact that no patterns could match it, an improvement for the documentation generation for similar cases has already been discussed in previous snippets. The problem of line 17, instead, can be seen from two different point of view: either the bad documentation comes from a lack of suitable patterns or the approach could be improved with a way to reuse existing documentation taken from the context of the snippet. More clearly, instead of writing a pattern that generates the text “Setup the pickers”, we could reach the definition of the *setupPickers* function and reuse its documentation. If the function has no documentation, we can attempt to generate one and reuse that one. However, this last improvement would require additional components not yet implemented in the system.

5.2.9 Other documentation

To conclude this section, we show other snippets that received a good score and we make a couple of observations. Figure 5.10 shows four different documented code snippets, each one marked with a number to divide it from the others. Every snippet shown in the Figure has been graded with a 5.

```
1 // Check if view is of type TextView .
  /*  If view is of type TextView , save the view 's  tag into the variable  com .  */
  if(view instanceof TextView){
      // Get the view 's  tag and save it into the variable  com .
      com = (Comment)view.getTag();
  }else{


2 // Check if  getDialog   is equal to null .
  /* If the return value of getDialog is equal to  null , get a view with resource ID " mdtp_done_background ", and set its visibility to   GONE .   */
  if(getDialog() == null) {
      // Get a view inside view with resource ID " mdtp_done_background ", and set its visibility to   GONE .
      view.findViewById(R.id.mdtp_done_background).setVisibility(View.GONE);
  }


3 // Set the  view  tag to  photo  .
  view.setTag(photo);


4 // Get a view inside view with resource ID " built " and save it in the variable  built of type  TextView .
  TextView built = view.findViewById(R.id.built);"
```

FIGURE 5.10: Four different code snippets documented by POET which received a grade of 5.

One thing we can notice is that all of the snippets are very short. We believe short snippets to be the easiest to document for the simple reason that less patterns are required to document them and so it is much easier for the right patterns to have been defined. For much longer snippets, instead, we usually assigned a lower grade due too the fact that there was not a compatible multi-line pattern that could summarize the whole snippet or, at least, different parts of it. This is not, however, an exact rule: there have been exceptions on both sides (short snippets with a bad or without documentation and long snippets with a very good documentation), but it seems that, especially for very long snippets, it holds true for the majority of the cases.

5.3 Summing Up

We can now summarize our findings about the results obtained from POET.

The first thing we analyzed was the coverage of POET and we discovered that it was able to document every snippet of the test set. Then we observed the quality of the generated documentation and analyzed the results. In particular, we discovered that 75% of our documented snippets were rated at least 3 and the average mark was 3.3 as opposed to the 2.05 of another state-of-the-art approach. Finally, we read and analyzed different documented code snippets in order to find some of the most common problems and strength of POET. In particular, we noticed that POET is able to generate very good documentation for short snippets, but, most of the times, it lacks the patterns to describe at an higher level very long snippets. Other problems and possible improvements are discussed in the Future Work section.

Chapter 6

Threats to Validity

This chapter discusses the threats that could affect the validity of the reported results.

6.1 Construct Validity

Threats to *construct validity* concern the relationship between theory and observation. While analyzing the results of POET, both quantitatively and qualitatively, we considered equally important each snippet of the test set, regardless of its length, context, or complexity. This may result in very simple code snippets that are always well-documented and more complex code that is, instead, documented in unsatisfactory ways. Our experimental design does not consider the code snippet complexity as a main factor in the analysis.

6.2 Internal Validity

Threats to *internal validity* concern factors internal to our study that could have influenced our results. A possible source of bias in our findings may have been introduced by subjectivity during the manual evaluation. This is due to the fact that the qualitative evaluation of each generated documentation has been carried out only by the author of this thesis. However, the scoring criteria have been discussed with the supervisors, in order to reduce the subjectivity bias. Also, the performed evaluations are available in the replication package.

Concerning the competitive techniques, the knowledge base in ADANA may not contain recent code snippets of the two APIs we considered in our study (i.e., *android.graphics.BitmapFactory* and *android.view.View*), thus penalizing this approach in the experimentation. Similarly, the deep learning approach has not being directly trained on Android-related snippets. Thus, it is possible that the obtained results were sub-optimal and not really representative of its capabilities.

6.3 External Validity

Threats to *external validity* concern the generalization of our findings, that is bounded by the limited number of patterns defined and by the size of the test set, as well as the categories that we chose to document (snippets which use the Android APIs *android.graphics.BitmapFactory* and *android.view.View*). It is possible that the quality and the coverage level observed for the snippets of our test set might not generalize to other "families" of snippets.

6.4 Conclusion Validity

Threats to *conclusions validity* concern the relationship between the experimentation and the outcome. It is important to stress that in our study we assessed the coverage and quality of the generated documentation, but we cannot claim that POET supports in code comprehension activities. This would require a controlled experiments with developers performing code comprehension tasks with/without the support provided by POET.

Chapter 7

Conclusions and Future Work

In this thesis we presented POET, a pattern-based approach to automatically generate a documentation for a given code snippet using patterns previously defined through a web application. The approach is currently tailored to work on Java Android applications, but it can be easily extended to work with any other environment or programming language. For example, assuming we would like to extend POET to support the C++ programming language, this would require:

1. Modifications to the preprocessing scripts. In particular, we should mine C++ repositories instead of Android ones and modify or extend the methods and snippets extraction scripts to follow the C++ AST structure. The AST must be saved in JSON format.

2. Minor modifications to the backend, to support C++ AST nodes instead of Android ones. The parser to create a new snippet should also be a C++ parser (the same used to extract the methods and the snippets) instead of a Java one.

3. Replacement of the Android API documentation miner with the C++ one.

While the results in the performed evaluation are encouraging, we believe that the approach can continuously improve its effectiveness by defining more patterns. Additionally, while defining patterns to test POET, we identified different improvements that can be implemented in the system to improve the reusability of patterns and lower their creation time. Such improvements are described in the Future Work section. Our long-term vision is that a number of developers will be able to increase the effectiveness of POET by defining patterns and sharing them with other users. Gamefication could be a way to involve a greater number of developers, especially if the pattern definition process will be simplified in a future implementation.

7.1 Future Work

After having implemented and tested POET we noticed a number of features that can improve the effectiveness or ease of use of the approach. In this section, we list them along with a brief description.

- **Custom AST**: instead of using the AST as it is generated by the parser, it might be useful to modify it to include more data to be used during the pattern definition.
- **Advanced matching**: while defining patterns it might be useful to have more tools at hand. For example, the keyword matcher could be expanded to support optionals with alternatives, which would allow the user to not define the same pattern twice, but, instead, check for a match and use a default value instead. An example of this scenario can be when we are trying to define a pattern for a getter function call, let it be called *getRandomItem()*. In this case, the documentation of this function would change based on if the function has a qualifier or not (*getRandomItem()* can be documented as "Return a random item from this object." while *list.getRandomItem()* should be documented as "Return a random item from list"). Instead of defining the same pattern twice, we could use a special operator which is able to return the first matched element amongst a sequence of AST paths or Strings (*i.e.,* "Return a random item from <expression.qualifier || "this object">").
- **Documentation builder**: one of the most annoying problems while writing patterns was attempting to define every possible pattern for an "if" statement. This problem can be solved by implementing a documentation builder. Right now, we need to define a pattern for every possible combination of "if" conditions (*i.e., x > 1*, *x > 1 && x < 10*, *x > 1 || x < 10*, and a lot more considering that an "if" can have an arbitrary number of boolean checks which can also use variables, literals and function calls). A documentation builder would fix this problem by reading and parsing any "if", "for", "switch" and other statements and automatically apply sub-patterns found inside them to generate a cohesive documentation. For example, given *if (x > 1 || x < 10) ...* the documentation builder will read the "if" and will create a documentation by putting together the two patterns matched for *x > 1* and *x < 10* to return as output something like: "Checks if x is greater than 1 or x is less than 10".
- **Increase pattern reuse**: A problem, similar to the previous one, that we faced while defining patterns, was that strictly reusing the AST nodes was very limiting to pattern reuse and required many patterns to be defined when just a simple one could have been enough. This problem can be seen in function calls: instead of defining many patterns for every type of parameters that can be passed to a function (which can also be very complex and long), there should be the option to say to the system to write the documentation generated by a sub-pattern. For example, adding this feature would allow to write a single pattern for any setter function which takes a single argument as input: *title.setText(...)* would be documented as "Set the title text to <pattern matched to the parameter ...> ." instead of defining many patterns for covering the cases in which the parameter is a variable, a literal or a chain of function calls.
- **Order and size of multi line patterns**: when matching a multi line pattern, it can happen that we get very strange matches that might span a lot of lines and they match seemingly unrelated lines of code. This is because, when defining multi line patterns, right now a user is not able to specify the order in which the written keywords should be

matched, so if we are defining a pattern for a variable declaration that is then used in a function call, the system can match first the function call and then the variable declaration, which is wrong. Additionally, the system is not limited by the size of the snippet, so it can match very far lines, even in two different scopes which would not be ideal.

- **Complexity of pattern definitions**: a great improvement to make would be reducing the complexity of a pattern definition. Right now, we have noticed that there are many little things that it is easy to oversight while defining a pattern. To solve this problem, there should be a default value for such nodes that is not required to be written by the user in the requirements but, instead, it is added by the system. For example, prefix and postfix operators ($i{+}{+}$, ${+}{+}i$) are rarely used, but, if you define a pattern without specifying that they should not be present, when they will be in the code it is possible that the pattern you defined will be matched, even if the documentation generated by it does not represent the piece of code due to the presence of one of the operators.

Bibliography

[1] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza. Automated documentation of android apps. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, Jan. 2019.

[3] A. Alqaimi, P. Thongtanunam, and C. Treude. Automatically generating documentation for lambda expressions in java. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 310–320, Piscataway, NJ, USA, 2019. IEEE Press.

[4] R. Buse and W. Weimer. Automatic documentation inference for exceptions. pages 273–282, 01 2008.

[5] A. Gatt and E. Reiter. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, pages 90–93, Athens, Greece, Mar. 2009. Association for Computational Linguistics.

[6] D. Guarnera, M. L. Collard, N. Dragan, J. I. Maletic, C. Newman, and M. Decker. Automatically redocumenting source code with method and class stereotypes. In *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, pages 3–4, Sep. 2018.

[7] M. Hassan and E. Hill. Toward automatic summarization of arbitrary java statements for novice programmers. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 539–543, Sep. 2018.

[8] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210, 2018.

[9] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, Aug. 2016. Association for Computational Linguistics.

[10] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, Feb 2016.

[11] 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, May 2013.

[12] C. Newman, N. Dragan, M. L. Collard, J. Maletic, M. Decker, D. Guarnera, and N. Abid. Automatically generating natural language documentation for methods. In *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, pages 1–2, Sep. 2018.

[13] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. pages 43–52, 01 2010.

[14] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80, June 2011.

[15] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 106–109, New York, NY, USA, 2014. ACM.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[17] 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, Feb 2017.

[18] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, Oct 2018.