Context-Aware Software Documentation

Emad Aghajani
Software Institute, REVEAL
Università della Svizzera italiana (USI), Lugano, Switzerland
emad.aghajani@usi.ch

***Abstract*—Software developers often do not possess the knowledge needed to understand a piece of code at hand, and the lack of code comments and outdated documentation exacerbates the problem. Asking for the help of colleagues, browsing the official documentation, or accessing online resources, such as Stack Overflow, can clearly help in this "code comprehension" activity that, however, still remains highly time-consuming and is not always successful.**

Enhancing this process has been addressed in different studies under the subject of automatic documentation of software artifacts. For example, "recommender systems" have been designed with the goal of retrieving and suggesting relevant pieces of information (*e.g.,* Stack Overflow discussions) for a given piece of code inspected in an IDE. However, these techniques rely on limited contextual information, mainly solely source code.

Our goal is to build a context-aware proactive recommender system supporting the code comprehension process. The system must be able to understand the context, consider the developer's profile, and help her by generating pieces of documentation at whatever granularity is required, *e.g.,* going from summarizing the responsibilities implemented in a subsystem, to explaining how two classes collaborate to implement a functionality, down to documenting a single line of code. Generated documentation will be tailored for the current context (*e.g.,* the task at hand, the developer's background knowledge, the history of interactions). In this paper we present our first steps toward our goal by introducing the ADANA **project, a framework which generates fine-grained code comments for a given piece of code.**

***Index Terms*—software documentation, program comprehension, context-aware, recommender system**

I. Research Statement

Software systems are often developed by several teams of developers responsible for developing and maintaining different subsystems. Thus, it is not unusual for developers to deal with unfamiliar code they have difficulties in comprehending. This is especially true when the code lacks documentation and comments [**?**], thus hindering the code comprehension activity. Even when comments are there, there is no guarantee that they are aligned with the implemented code, since the asynchronous evolution between code and comments has been empirically demonstrated in previous studies [**?**], [**?**], [**?**].

To make up for the lacking knowledge, developers browse through official/unofficial sources of documentation or ask colleagues [**?**]. This process, called *code comprehension*, is time-consuming and can account for up to 70% of the time spent by developers in their daily activities [**?**]. Fostering this process has been the goal of several researchers who developed a number of approaches for the *automatic documentation of software artifacts*. Many of these approaches are in the form of recommender systems, that analyze the code in the Integrated Development Environment (IDE) and try to recommend useful pieces of documentation for the specific task at hand.

Although these techniques have demonstrated their ability in fostering the code comprehension process, they still exhibit a number of major limitations. Indeed, when giving recommendations, they tend to ignore the level of expertise of the developer and the specific task at hand (*e.g.,* does the developer need to comprehend the code in the context of a bug-fixing activity or of the implementation of a new feature?). Also, most of them work at a fixed granularity level, not allowing to document a single line of code of interest (*i.e.,* to explain what it is implemented by that line) or a whole subsystem.

Our goal is to build a context-aware proactive recommender system which is able to automatically document a given system at different levels of granularity. Our ideal recommender system will be able to (1) understand when a developer is looking for missing information, and (2) provide dynamic information considering the context (*e.g.,* the task at hand, the developer's background knowledge).

We believe the use of contextual information could enhance the "usefulness" of existing approaches, and result in increased developers' productivity.

II. State of the Art

The code comprehension process has been extensively studied by researchers, also with the aim of devising strategies to improve it. Some of these strategies are described in the following.

Summarization. Code summarization techniques have been used in the context of automatic software documentation. The conjecture is that concise natural language descriptions of source code fragments enhance the code comprehension process by reducing the amount of code to be read by the developer and, as a result, the time required to understand the code.
Summarization can be done in two ways: extractive or abstractive [**?**]. Extractive summaries [**?**], [**?**] are obtained by selecting a subset of document elements (*e.g.,* a subset of code statements) which represents the most important information in the code. Note that this is not really a piece of documentation that can help in comprehending a complex piece of code, but rather a way to save time to developers by removing "semantically-irrelevant" statements. Abstractive approaches [**?**], [**?**], on the other hand, take the semantics of the text and applies natural-language processing techniques to generate a

summary. In addition, these techniques produce summaries with different levels of granularity and are designed to summarize different types of documents: methods [?], [?], [?], [?], method usages [?], [?], classes [?], [?], [?], cross-cutting concerns in code [?], or other software artifacts [?], [?], [?], [?], [?], [?], [?].

State-of-the-art code summarization techniques mostly rely on information retrieval (IR) techniques such as PageRank [?], [?], LexRank [?], or Maximal Marginal Relevance (MMR) [?]. However, some summarization approaches leverage new techniques, for instance HoliRank [?], [?], an extension of PageRank [?] devised to analyze data in a holistic fashion. Some other approaches rely on techniques such as neural networks [?], topic modeling [?] and method/class stereotypes [?], [?], [?].

Although summarization is one of the common techniques that can be adapted to document different types of software artifacts, it falls short if the information to comprehend the code is simply not inside the original document. In these situations, one needs to seek the information using external sources. This leads us to the mining of crowd knowledge.

Mining Crowd knowledge. Researchers proposed several approaches which mine and process crowd knowledge (semi-)automatically, and distill the most relevant parts. The term "crowd knowledge" refers to any type of information produced by the crowd, in our scope, developers. Prominent examples of such type of knowledge are Stack Overflow discussions, where a developer can find numerous source code associated with natural language descriptions. Relying on this fact, researchers have proposed different techniques to leverage this information, mostly by suggesting a discussion relevant to the source code at hand [?], [?], [?], [?], [?], [?].

Wong *et al.* [?] proposed an approach, called CloCom, which mines a set of given projects (inputs) to generate code comments for a target project. The idea is to reuse existing code comments from input projects in order to generate comments for a target project. The results show that 23.7% of the automatically generated code comments are useful at describing the source code, which suggests an improvement on the earlier approach by the same authors, AutoComment [?], in terms of number of useful comments generated for the same set of target projects in both research.

The mentioned automatic documentation approaches vary mostly on the way information is retrieved, processed and finally presented to developers. The source code a developer is working on is the dominant input for most of them, regardless of other parameters that vary from person to person or task to task, and can be taken into account for more relevant and useful results. Binkley *et al.* [?] stress the importance of "useful" documentation, besides mere "good" documentation, and emphasize the necessity of exploiting non-code factors, such as the expertise level of the developer who the information is being given to. Moreover, Happel *et al.* [?] conducted a survey on some well-known recommender systems and discussed their limitations and extant challenges, indicating that obtaining useful documentation is not a straightforward task and there are several challenges to be addressed. In a recent discussion on the future of software artifact documentation [?], Robillard *et al.* conducted a review of state-of-the-art approaches and outlined the key challenges in three categories: information inference (*i.e.,* mechanisms to model and infer information), document request (*i.e.,* mechanisms to enable developers to express their information needs in a better way) and document generation (*i.e.,* approaches to generate appropriate output) [?].

In this research, we want to address some of mentioned issues, most importantly, the lack of context-awareness and fixed granularity level of documentation. Our objective is to devise a context-aware recommender system which is able to document a given system at whatever granularity is required.

III. Current State of Research

We thus far focused on (1) studying documentation issues, and (2) implementing a fast prototype of our envisioned recommender system for the automatic generation of documentation.

A. Documentation Issues

We investigated documentation issues and their effects on developers. In this context, Arnaoudova *et al.* [?] presents a catalog of 17 *Linguistic Antipatterns* (LAs), representing inconsistencies among the implementation, naming and documentation of source code entities such as methods.

We conducted a large-scale empirical study [?] on 1.6k releases of popular Maven libraries and 14k open-source Java projects using these libraries, to understand the impact of using APIs affected by LAs on these projects as compared to using clean APIs. Our in-depth quantitive and qualitative analyses suggested conflicting evidence. We plan to further investigate LAs issues, as well as other kinds of issues that software developers deal with when using documentation.

B. Documentation Generation

We proceeded toward our vision with the ADANA project, a framework which generates fine-grained code comments for a given piece of code by reusing similar well-documented code snippets. The framework consists of an Android studio plug-in, a set of backend services for analyzing and extracting data from online repositories, and a knowledge base for storing snippets and descriptions. The knowledge base stores 64K pairs of $\langle code, description \rangle$ from GitHub Gist, Stack Overflow discussions and Stack Overflow Documentation. We also devised ASIA (Android SImilarity Assessment), a clone detection approach tailored for Android. Given a code snippet to be documented, ADANA tries to find potential clones of it in the knowledge base and reuse the associated description. Listing 1 reports an example of comment automatically generated by ADANA. Although ADANA correctly documents the code behavior ("*Shows a Clear button after the first character*

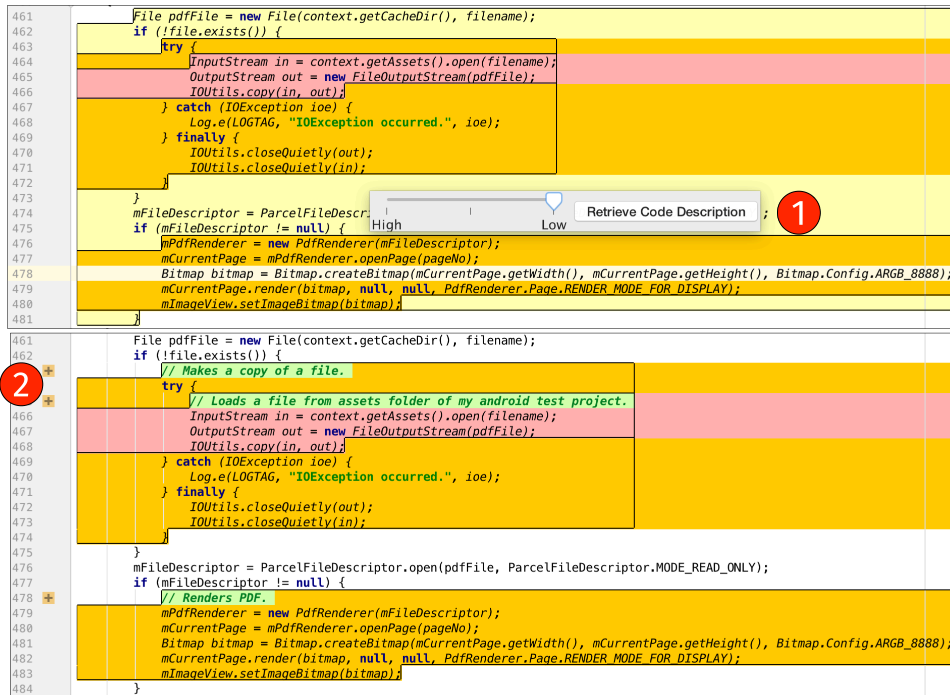

Fig. 1: ADANA Android Studio plug-in GUI

pressed"), it also generates an imprecise comment describing a behavior not implemented in the code snippet ("*hides it when the text is empty*"). We plan to address this limitation in our future work.

```
public void onTextChanged(CharSequence s, int
    start, int before, int count){
  // Shows a Clear button after the first
     character pressed and hides it when the
     text is empty
  if(s.length() > 0){
    clear_button.setVisibility(View.VISIBLE);
    searchText = s.toString();
  }
}
```

Listing 1: Example of an injected comment

Figure 1 depicts the ADANA Android Studio plug-in. The developer first selects a code fragment she is interested in comprehending and invokes the ADANA plug-in. ADANA shows a granularity slider (1) to set the granularity of the comments one is interested in retrieving: If the slider is to the left, ADANA looks for clones of the whole code selection and, in case of successful retrieval, only injects a single comment describing the selected code. Moving the slider to the right, ADANA decomposes the selected code on the basis of the indentation level, as identified by parsing the AST representing the selection. Each of the parts ADANA tries to document is shown in a different color. The maximum value of the granularity slider depends on the maximum indentation level of the selected code. Once the developer picks the granularity, she clicks on the "Retrieve Code Description" button close to the slider, obtaining the descriptions (highlighted in green) retrieved by ADANA for each of the highlighted code portions. By using the code markers added by ADANA (2) , she can either accept it as is, modify and accept it, or reject it. If she accepts (before/after changing it), both code snippet and the associated comment are added to the ADANA knowledge base.

We have evaluated both ADANA and ASIA comprehensively in three studies. The results showed that ADANA is able to detect clones for Android code snippets with a precision of 77,76%, relying on ASIA. Moreover, ADANA can boost the program comprehension process by generating comments which were mostly considered as "useful" by the study participants.

In comparison with similar approaches, *e.g.,* CloCom by Wong *et al.* [**?**], ADANA provides a first basic implementation of the ideal recommender system we would like to develop in the long-term run, *e.g.,* enabling developers to decide upon the granularity level of injected comments. Moreover, the ADANA knowledge base becomes richer as new questions and Gists emerge, whereas the performance of CloCom is highly dependent on the projects which are specified as the input.

However, ADANA has some limitations. For example, it relies on limited contextual information (only the code in the IDE), can only be invoked on-demand (*i.e.,* it is not proactive), and can not document coarse-grained components (*e.g.,* a whole subsystem). We plan to address these limitations as part of our future work (see Section IV).

IV. Conclusion and Future Research Plan

Our idea is to define and experiment techniques serving as the basis for a novel generation of recommender systems acting as an intelligent personal assistant during code-related activities (*e.g.,* fixing a bug, implementing a new feature, *etc.*). To that end, we are investigating documentation issues experienced by software developers. Moreover, we are working on a number of key features in ADANA, including:

1) **Context-aware.** Currently, recommender systems supporting developers during code-related activities exploit the code in the IDE as the main source of information to capture the context and, then, recommend useful pieces of information for it (*e.g.,* related Stack Overflow discussions). However, the working context is much more than a bunch of lines of code shown in the IDE. For example, two developers having diverse expertise level should receive different types of documentation even when working on the same task on the same code component. Ignoring the developers' profile might result in generating documentation that is either "too trivial" or "too complex", thus useless in both cases. We plan to capture the context in which documentation will be generated by considering:
   - **Developer's profile.** Information such as the code fragments developed in the past, the developer's expertise on the different technologies used in the project, and the history of successful and unsuccessful tasks performed in the past form a developer's profile. For example, a bug fixed by a developer and not "reopened" in the future can be considered a successful task performed by the developer, while a fixed bug that has been then reopened and fixed again, can represent an instance of an unsuccessful task. Knowing this can help in assessing the experience level of the developer on a particular subsystem (*i.e.,* the one involved in the bug-fixing activity) and, thus, to tailor the generated documentation to the specific developer's profile.
   - **Task at hand.** Considering the developer's task at hand can provide hints useful to narrow down the type of information she needs. For example, during a bug-fixing activity it is important to understand and thus, to automatically document, the production code as well as the test code. The idea of utilizing the current task information to assist developers has been adopted and shown to be effective [**?**], [**?**] and has been implemented in Mylyn[1], a plug-in for the Eclipse IDE. However, this information must be explicitly indicated by the developer, and it is not automatically inferred by the tool. We plan to exploit data from the issue tracking system to infer the tasks the developer is working on.
2) **Heterogeneous sources of information.** As previously discussed, the automatic documentation of source code can be achieved in different ways (*e.g.,* extractive *vs* abstractive summaries). What clearly makes a difference in the ability to document a given code are the basic information sources exploited. We plan to exploit not only the official project's documentation, but also the project's repositories (*e.g.,* history of changes, information extracted from the issue tracker, *etc.*), as well as the information that can be mined from the Web such as documentation written by other developers for code similar to the one to be automatically documented.
3) **Proactive and On-Demand.** While we want the developer to be able to interact with our system on-demand (*i.e.,* by asking explicit questions), we also want to provide proactive recommendations in case, for example, we can infer that the developer is struggling to understand a code snippet. This can be done by monitoring her behavior in the IDE and observing patterns likely indicating understandability issues (*e.g.,* scrolling up and down several times over a specific method). In this cases, the tool could automatically document the code with hints on what the different lines of code implement.
4) **Self-improving.** We want to explore the usage of feedback mechanisms to allow our techniques to self-improve over time. For example, once a piece of documentation is automatically generated for a given code snippet, the developer can provide feedback indicating whether it was useful or not to comprehend the code. This feedback can be then exploited to infer good and bad "commenting patterns", possibly customized upon the developer's preferences.
5) **Information granularity and presentation.** The recommender system must be able to present information at different granularity levels. For instance, facing a class to comprehend, one developer might be interested only in a high-level description, while others might expect instantiation examples. Thus the system must be able to generate documentation at different granularity levels and provide the developer with a "show me more/less details" mechanism. Also, the way the documentation is presented to the developer is important. In particular, we will investigate different presentation techniques to select the most appropriate one based on the amount and type of data to present.

Moreover, we aim to extensively study the evolution of documentation over time, investigating the issues, common practices, and trends in this area.

Currently, I am few months far from starting the 3rd year of a PhD under the supervision of Prof. Michele Lanza and Prof. Gabriele Bavota. I intend to continue my research on automatic software documentation by (1) working on a context-aware recommender system, and (2) studying the nature of software documentation per se, with a specific focus on documentation issues experienced by software developers.

ACKNOWLEDGEMENTS

I gratefully acknowledge the financial support of the Swiss National Science Foundation for the PROBE project (SNF Project No. 172799), and CHOOSE for sponsoring my trip to the conference.

[1]http://www.eclipse.org/mylyn/