Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.Y. 2022/2023

Graduation session April 2023

Dear Diary

Helping novices creating better code documentation

Supervisors:

Luigi De Russis

Juan Pablo Saenz Moreno

Candidate:

Gabriele Sara

Acknowledgements

First of all, I would like to thank Prof. Luigi De Russis and Juan Pablo Saenz Moreno who gave me this opportunity, entrusted me with this work and supported me all the way through. I would have never been able to finish this project without your help.

A special thanks go to my family that has always been there for me even through the hardest times. A special thought goes to my mom who always believed in me and supported me when I was going crazy between exams and the thesis. I thank my dad for always being there when I needed it and trusted me to make it. I also really appreciate my sister Elisabetta who made me cheer up, joked with me, always had a way to calm me down and make me smile, and helped me whenever and however she could: if it wasn't for you I wouldn't be graduating now. I cherish my brother Marco who has been thoughtful and caring to me and has just started his nurse University career: I'm proud of you.

I appreciate my uncle Costantino and my aunt Maria Franca for supporting me in every way they could. I specially thank my godmother Paola, who has always greatly cared about me and has always believed in me. I wanted to thank my godfather Cristoforo and aunt Marisa who hosted me at the beginning of this journey and for always helping me whenever I needed them. I also thank my uncle Giuseppe and my aunt Elda for morally supporting me and for motivating me through these five and a half years.

I want to take a moment to express my deep affection towards my Sardinian friends: Luisa, Stefano, Claudia, Andrea, and (Simone) Orrù. I particularly thank Stefano for all the laughs, the game nights, and all the fantastic moments passed together in Turin. A really special thanks needs to go to Luisa. She has always been there for me, helped me, motivated me, made me laugh, and supported me through it all. Thank you for these ten years of friendship. The way I am now is also thanks to you.

I thank Luca, that has always been there for me whenever I needed it and cheered me up anytime I would feel down. You and Stefano gifted me with unforgettable memories and I thank you both a lot for that.

I thank Cosimo that has been by my side throughout all these years. He really

helped me passing some difficult times and supported me all the way through. I sincerely appreciate you and our friendship.

I would like to thank Roxy for always believing in me, for all the talks we had during our long walks, for her suggestions, for her vitality and her joy.

I really want to thank everyone of the 4th floor of A2 (Borsellino Residence). Raffaele, Veronica, Alessia, Manuel, Agata, Michele, Elisa, Federico, Valentina, Pierluigi, and Federica, I have many unforgettable memories with you. Thank you for all the laughs, the discussions, the support, the late night games, talks and drinking, and all those times we went dancing at Bananamia ;). I really appreciate you guys.

I especially want to thank Raffaele, my "thesis mate", who borne me complaining a little too much through these last few months. You always gave me solutions whenever I had problems. You taught me a lot (including how to cook). I really cherish you and our friendship. I wish you the best of luck for what lies ahead.

Another special thanks goes to Veronica. You were always by my side and always cheered me up. Whenever I needed you, you were there for me. You gave me different perspectives on things, and I really value that. Thank you for your honesty, your patience, and every moment that we passed together.

Moreover, I want to thank Alessia, my Sardinian friend of the floor. You're the one that I can trust to be there whatever I want to do. You made these months lighter thanks to our jokes. You're the one that, when I was going out of my mind because of this thesis, tried to understand my work just to try helping me. I seriously appreciate you.

I also want to thank anyone that participated at the user test: Michele, Cosimo, Viel, Lisa, Alberto, and Tommaso. I seriously appreciate your help. I couldn't finish this thesis without you.

In the end, I want to thank anyone that has been part of this journey lasted almost six years. These years have been unforgettable thanks to all of you.

Finally, I want to thank myself for succeeding on completing this chapter of my life even though, many times, it seemed impossible to go on.

Table of Contents

List of Figures

Chapter 1

Introduction

When working on a new project, novices will often neglect to create related pieces of documentation. Moreover, this kind of documentation is almost entirely composed by comments on the code, which has poor utility when using it for future reference. The documentation that has been created, if created at all, lacks a series of related and useful information, such as: which resources have been consulted, errors encountered and the relative solutions implemented, programming choices, lessons learned, and suggestions for future work. In addition, the documentation does not properly account for the evolution of the project and so the achievements reached.

Despite the presence of some researches that aim at better understanding the code and easily integrating code examples found on the internet [1, 2, 3, 4], little work has been done in implementing a seamless way to create documentation that can also be enriched by the developer by adding comments, thoughts, suggestions. The possibility of adding personal insights can also improve the awareness of the implemented code as suggested by Lehtinen et al [5]. Furthermore, the current documentation practices do not consider contextual data, essential for the implementation of the project, such as: dependencies, command line scripts executed, file and project structure.

For these reasons, in this thesis I present Dear Diary, a tool for the seamless creation of documentation directly in the Integrated Development Environment (IDE) through which developers can keep track of the evolution of the project and add insights, with their own words, on how they reached their development achievements. In particular, the tool allows developers to create “snapshots” - documentation artifacts that store pieces of code, files, or entire projects, and collect contextual and useful information like dependencies, the file tree, and the command line scripts executed. These snapshots are grouped together into larger structures called “diaries”. On top of that, the developer can add comments and links on any of the registered data contained in a "snapshot", further enriching the piece of documentation created.

1.1 Goal

The aim of this thesis is to design and develop a tool, directly implemented in the IDE, that can be used by novices to seamlessly create pieces of documentation enriched with contextual data and comments, links, and thoughts of the developer.

Our strategy is to encourage beginners to record their development process and to make it simple for them to do so in order for the resulting documentation to be useful in supporting them in the future. Our technique seeks to document the entire development process, in contrast to many research studies that concentrate on automatically creating documentation based on accessible code. In particular, as will be discussed in greater detail in the chapters that follow, our strategy aims to capture and document the project's status as it develops rather than at a single point in time.

A first literature analysis was conducted to look for similar and/or related projects in order to extrapolate the core functionalities the tool needed.

Using the literature research as reference, the functionalities the tool could support have been discussed and evaluated. We envisioned the tool utilization around the creation of documentation artifacts called "snapshots" that would contain project's context data.

After selecting the core features that firstly needed to be implemented, we went through a design phase.

The tool then was designed around the core concept snapshot, which would save information about the project and automatically gather contextual data, including: the file tree, the project dependencies, and the command-line scripts executed.

After implementing the tool as an extension for Visual Studio Code, an user study was conducted to verify whether the tool was effective in the creation of documentation on the fly to better support novices in the development of projects and in keeping track of the evolution of the code.

1.2 Thesis organization

The thesis is structured as follows:

- Chapter 2 focuses on the literature analysis conducted on related projects to extrapolate the main functionality supported by the tool.

- Chapter 3 describes the design choices made. In particular, core functionalities of the tool are selected.

- Chapter 4 presents the tool structure together with its main functionalities. Technical choices and limitations are also discussed.

- Chapter 5 focuses on the user study conducted, its methodology, and the type of data that was acquired.
- Chapter 6 focuses on the results of the study conducted and presented in chapter 5.
- Chapter 7 presents a final discussion on the results found, the future potentiality of the tool, and its changes and improvements.

Chapter 2

Background and related works

2.1 Preliminary analysis

It has long been maintained that good documentation is essential for helping developers write code more quickly and consistently with design decisions, and that documenting design decisions in particular is essential for preserving code readability [6]. Conventional documentation methods, however, are ineffective. They need a lot of work to be created, and there is a disconnection between the producers and the consumers [7, 8]. It takes time to provide appropriate documentation [9], and the developer could feel overburdened when he weighs that against the prospect that no one would ever read his effort. Because of this, IDEs ought to give developers more assistance when writing project documentation. This procedure should not only assist the programmer in cutting down on the time spent documenting, but it should also automatically capture helpful data that the programmer may not have deemed important in order to be consulted afterwards [10].

Mehrpour et al. [11] suggests a sort of documentation where design decisions are specified as design rules and translated into constraints that can be actively tested against the code when looking at the area of tools that try to improve and ease documentation generation. A link between the documentation and the code is created when a design rule is applied to the code. An Abstract Syntax Tree (AST) query set serves as the representation for each design rule. Upon code modification, developers receive quick feedback on the design rules that have been followed and those that have been broken, alerting developers who missed design decisions to their presence.

Head et al. [12] proposes Tutorons, which are language-specific procedures that

generate context-relevant, on-demand code explanations. It recognizes single-line code snippets on web sites, parses them, and creates micro explanations in the form of natural language explanations and code examples automatically. It's designed to work with CSS selectors, regular expressions, and the Unix function *wget*. The programmer, in particular, may read descriptions for all explainable sections of code and provide new micro-explanations for non-documented regions of code by installing a browser extension.

Henley et al. [13] introduces the idea of a tool for recognizing and correcting novices' misunderstandings about code behavior. It consists of an inquisitive code editor in the form of a plugin for the Atom IDE that prompts beginners with inquiries about the behavior of their code while programming. The suggested tool might provide explanations regarding the program's real behavior and redirect the beginner to relevant documentation sources based on the answers supplied by the novices. The tool's preliminary implementation calls for the questions, correct answers, and feedback to be set manually by another programmer on a specific code fragment.

Horvath et al. [2] presents Catseye, an annotation tool for quick code note-taking. In contrast to conventional methods of externalizing code-related information, like commenting, Catseye has the advantages that the annotations maintain the original context of the code while not actually altering the underlying source code, they can support richer interactions like lightweight versioning, and they can be used as navigational aids. They discovered that developers were able to successfully employ annotations to help their code sensemaking when performing a debugging job during their examination of developers' note-taking procedures using their tool.

Lehtinen et al. [5] presents a way to examining students' code understanding by having them answer questions about their own programs. They suggested a method for creating queries about student-written computer code automatically. In the context of automated evaluation systems, they also suggest a use case for these types of queries: after a student's program passes unit tests, the system asks the student questions about the code. They contended that by serving as prompts for self-explanation, these inquiries can improve evaluation processes, deepen student learning, and give insight into how well students understand the programs they are creating.

Oney et al. [1] introduces a study that examines whether coders can use examples more productively if they are treated as "first-class" items in the code editor as opposed to just text strings. They investigated this by developing and testing Codelets: a piece of sample code that is displayed in-line with the user's own code and includes an interactive helper widget to aid in comprehension and integration. The Codelet survives the entire life of the example and is still available after setup and integration are finished. Programmers were able to finish tasks requiring samples an average of 43% quicker when using Codelets than when using a normal

Web browser, according to a comparison laboratory research with 20 participants.

Hesse et al. [14] presents DecDoc utility based on an incremental documentation approach. With the help of the tool, developers can work on a thorough documentation of design choices that pertain to assets like prerequisite specifications, design diagrams, and code. This helps to make associated information clear and represent it during the process, which helps to better the decision-making process for design choices. The DecDoc utility is presented in relation to the demands from the decision-making process.

Chen et al. [15] presents EdCode, a platform that lets learners access online teaching support through their IDE in a manner similar to in-person assistance. Additionally, it enables teachers to organize and post students' responses for an entire class by choosing only the pertinent portion of the code cited, thereby preventing copying. Teachers can frame responses by referring students' code. Through a number of usability tests, it was assessed EdCode and determined the advantages and drawbacks of using it in computer classes. Students viewed EdCode assistance to be of a similar quality to support received during in-person office hours, and both students and instructors found it useful to post and watch other students' answers.

Our approach aims to make it easy for beginners to document their development process and to encourage them to do so that the resulting documentation can effectively support them in the future. While many research works focus on automatically generating documentation based on available code [16, 17, 18], our approach aims to document the whole development process. In particular, as will be covered in more depth in the following parts, our method seeks to capture and document the project's status not at a single point in time but rather as it develops. Similar to this, Dear Diary aims to not be code-centric: in addition to remarking on code snippets, our method enables beginners to note on files, command-line processing outputs, and project dependencies.

2.2 First features evaluation

When checking in literature if an application with the same goals as the one we envisioned has previously been developed or if other similar projects have been worked on, we found that no single tool fulfilled our initial requirements in their entirety. Hence, we analyzed the pieces of work in literature to elaborate the functionalities our tool needed to support in order to better support novices in the creation of useful documentation.

From the work of Mehrpour et al [11], it was taken into consideration the implementation of a way to keep flags in a file whenever a related documentation artifact was previously created, keep the file name from which the artifact has

been created, and letting the user search specific words or keywords present in the documentation artifacts produced.

The study conducted by Lehtinen et al [5] suggested us that the tool needed to implement a way for novices to add their insights of the code, possibly throughout comments, which was already one of our requirements.

By consulting the work of Chen et al [15] we envisioned the utilization of hyperlinks in order to link words in comments and notes to specific pieces of code. This would be useful to better understand the context of a documentation artifact.

Finally, from the studies of Oney et al [1] and Henley et al [13] the importance of having links to external code documentation was intuited. For this reason, we discussed the implementation of an external references section in which programmers could insert external references links (e.g., developers forums or examples taken from the Internet) that helped them better understand how to properly implement a functionality in their code.

In the end, the possible utility of having a way to create documentation artifacts directly from the IDE was recognized when consulting the work of Mehrpour et al [11]. Just like their tool helped reducing the overhead of documenting design rules in a separate media by implementing a relative window directly into the IDE, our tool would help novices creating arguably better and useful documentation directly from the IDE.

Chapter 3

Design

3.1 Features evaluation

Using our literature research as a reference, the main functionalities that the tool needed to support have been discussed, selected, and designed.

First, the tool's focus was the creation of documentation artifacts called "snapshots". Each snapshot was designed to contain project-related data such as the file tree, the project dependencies, and the command line scripts executed, that would be automatically collected and stored in the snapshot data structure.

Once the core functionality that the tool needed to support was established, the other functionalities, connected to this main documentation artifact, have been evaluated, starting from those envisioned while making the research in literature discussed in chapter 2. Those being:

- Keeping flags in a file to signal that a connected snapshot have been previously created.
- Save the file name from which the snapshot has been generated in the snapshot itself.
- Let users search for specific words and keywords present in the snapshots and/or diaries.
- Allow users to add their insights on the code through comments contained in the snapshots.
- Utilize hyperlinks in order to link words in comments to specific pieces of code.
- Having a specific section for users to add links to external code documentation and/or code examples.

- The creation of snapshots and diaries needs to be seamless and easily accessible from the IDE.

Notably, because the ability for novices to add their own insights on the code through comments was considered an essential functionality for the tool, the idea was expanded. In particular, it was designed for the user to add comments in each data instance in the snapshot. This means that comments can be added to singular snapshot and/or related files in the file tree, dependencies, command line scripts executed. Moreover, the ability to create multiple comments related to a file has been thought to be particularly useful. This idea came about considering the fact that programmers can add comments for single functions and/or single lines of code contained in the same file in order to describe their behavior, design decisions around them, or simply signal a particular tricky piece of code. Furthermore, when considering that novices often relies on source code examples, forums, and tutorials online to get over execution and development errors [19, 20], adding comments become an even more powerful feature because it lets programmers link their documentation to the resources consulted.

Moreover, an interface for the visualization of differences between a selected snapshot and the previous one has been hypothesized. This would allow the programmer to better understand the additions and deletions made from one development step to the other.

Then, two additional sections have been hypothesized to be useful. The first one would allow the user to add issues present at the time of taking the snapshot. This can comprehend code errors, setup errors, program misbehavior, etc. The second section could be a goal one. Specifically, in this section the programmer could add the ultimate goal that wants to achieve through his code e.g. a specific functionality or a particular design for a section of the application.

Finally, in order to have all the documentation created to be easily accessible and readable, the ability of exporting all the snapshots as HTML files was considered. This would allow the consultation of the created documentation outside the IDE environment making it accessible to users who work in the same project but in different IDEs.

Once the tool functionalities have been laid down, the idea of the "snapshot" documentation artifact has been expanded and evolved. Three different types of snapshots were designed, each representing distinct types of projects documentation artifacts that would contain different data in addition to the general project-related data automatically collected by the tool:

- Code: the snapshot would include a piece of code (e.g., a function or a line of code).

- File: the snapshot would include the content of an entire file.

- Project: the snapshot would include the information of how the project has evolved at the moment of taking the snapshot.

Then, the concept of "diary" was formed: it was meant to enclose a collection of snapshots of the same type. This would allow the tracking of singular pieces of code as the project evolves over time. As an example, a code type diary could contain the snapshots taken in order to keep track of the main achievements reached during the development of the main project function.

Lastly, the core functionalities that the tool needed to support have been selected. They have been chosen based on importance and usefulness. As a result, the list of functionalities that needed to be implemented first have been synthesized accordingly:

- Creation of diaries and snapshots of three possible types (code, file, project).
- Automatically gather and save project-related data to the snapshots (file tree, command line scripts, dependencies).
- Ability to add comments to each individual piece of data in a snapshot (i.e., snapshot itself, a file in the file tree, a dependency, and a command line script executed).
- Show the data related to a snapshot directly in the IDE.

3.2 Features rationale

Instead of emphasizing the implementation products that emerge, our tool concentrates on the development process: the goal of Dear Diary is to encourage beginning coders to record their entire development process, not just the code they write.

In our tool, the novice's explanations are not always conveyed as textual explanations and are not always connected to the project's end code. Instead, they can also be linked to the files, command-line script execution results, and project dependencies. In this way, a beginner could add a note explaining the function of a specific file, justifying the execution of a particular command line directive, or outlining the snapshot context. The goal of this design consideration is to alter how beginners perceive the documentation process. Dear Diary is designed for beginners so that the documentation they create will be helpful for them to keep track of their development choices, replicate successes, and correct mistakes as necessary.

Since developers begin creating documentation files straight from the IDE, Dear Diary is able to autonomously gather data about the development and operation environment together with the project dependencies. By providing this

technological data, we hope to increase the projects' reproducibility and correctly identify any problems with the code's implementation, incompatibilities, and lack of necessary development and execution requirements. It is typical for beginners to ask for assistance when they run into a problem with a related answer requiring the installation or configuration of a dependency. In these situations, additional information regarding the development and execution environment is needed in order to replicate the issue and discover a fix. Consequently, this design choice of capturing the dependencies aims at making it possible to recreate specific project's versions and reproduce non-working snapshots, aside from keeping track of the development and deployment setups, since the successful implementation does not only concern the code but the development and execution dependencies (especially in web development projects).

Errors will inevitably occur when using a particular programming language or development environment for the first time. Programmers typically look for solutions in developers' online communities, instructional videos, or formal literature. They attempt numerous code samples and study numerous sources before they finally figure out the problem. It is not a good idea to submit broken versions of software projects, but in the context of our beginner users, keeping track of broken code, incorrect runs, error messages, and particularly how to fix them, is desirable. Regarding this, Dear Diary enables coders to take screenshots of broken versions of the project, add comments to broken versions of the code files, and describe the results of the erroneous command-line commands. It would not be capable of outlining the sources consulted to resolve the problems without recording the incorrect executions. Moreover, documenting faults and include self-explanations in solutions might aid beginners in remembering their tactics and replicating them across other implementations.

3.3 Workflow prototyping

Once the main features that the tool needed to support have been selected, the workflow was laid down.

The user can create a new Diary and so take the first related snapshot. He would select the type of diary wanted, then he would insert the diary name and the optional snapshot name. Once this is done, the tool would collect the project related data (file tree, command line scripts launched, project dependencies), save everything in the snapshot data structure and save this snapshot data in the diary data. Then, the just created diary (with its first snapshot) would be visible in to the proper view.

From this view, the diaries view, the user can add another snapshot to a previously created diary, check the snapshots saved in a diary, and open a snapshot.

Opening a snapshot would show the piece of code or the file saved (in the cases of a code or file snapshot) or bring the project to the version it was in at the moment of taking the snapshot (in the case of a project snapshot). The related data would be shown in the respective views.

By clicking one of the data instances, or the associated comment button, the user would be able to add comments to the piece of documentation selected. The interface that would allow to write down the comments would be in another view. The comment, once saved, would be consultable at any moment by clicking on the data instance again. In this way the comment view would update to show the previously created comment and he would be able to edit it. Moreover, by clicking on the snapshot or by clicking the associated comment button, the view would shoe the interface that would allow the user to insert the comment for the snapshot itself and in addition it would show all the comments related to the snapshot. This means that the comments associated to files in the file tree, command line scripts, and dependencies related with the snapshot would be visible altogether in the comment view.

Chapter 4

Implementation

4.1 Tool structure

For the implementation of the tool, it was fist decided for which IDE the extension would be developed. Visual Studio Code has then been chosen as the designated IDE due to its popularity and ease of use.

Next, we envisioned the tool interface. We went through a series of discussions in order to produce some hand-drawn sketches. We also followed some Visual Studio Code design guidelines [21] in order to be as close as possible to the final product, so that in the next development sessions we only needed to focus on implementing the selected functionalities.

Since each project's partial version (or snapshot) consists of files, command-line scripts, and project dependencies, the tool structure has been designed around these pieces of data. The design elements are organized around five views as a result, each having a specific purpose: a view to start the creation of diaries, a view for the visualization of diaries and snapshots, and other three views for showing files, command line scripts, and dependencies. A sixth view has also been designed in order to comply with the requirement of making available to the user the insertion of comments.

The extension has been implemented in two main sections: the left side bar and the right side bar (or, optionally, the bottom panel), as shown in Figure 4.1.

The left side bar is divided into five views, entitled as follows:

- New Diary
- Diaries
- Files
- Command line scripts

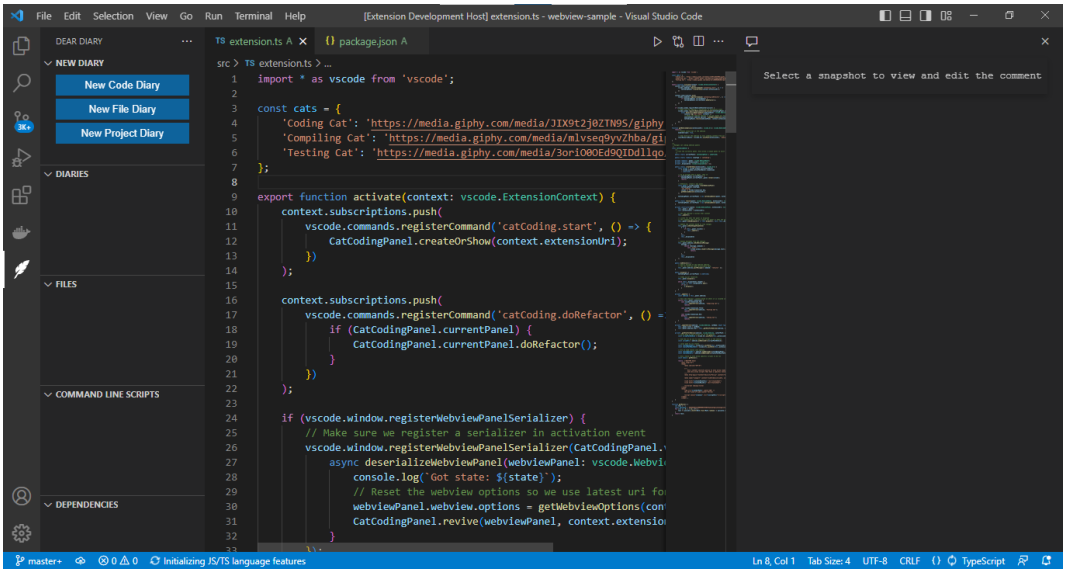

Figure 4.1: Tool structure

- Dependencies

In the first view, the "New Diary" one, there are buttons for the creation of the three different types of diaries (code, file, project). By clicking one of them, the process for the creation of the respective new diary, and its related first snapshot, starts.

In the "Diaries" view, all the previously created diaries are shown. When one of them is clicked on, the sub-list of associated snapshots expands, allowing the user to open any snapshot in order for it to be consulted.

The "Files" view let the user see the file tree of the project she's working on. This includes all the folders and any type of file present in the project folder. The hidden folders are not visible.

The "Command line scripts" view is used to show the scripts associated with the opened snapshot. It allows to check the scripts and consult their results just by clicking on the entry in the view.

Finally, the "Dependencies" view lets you check out the dependencies associated to the project from which the opened snapshot has been created.

In the bottom panel, the "Comment" view is present. It can be optionally dragged to the right, in order for it to populate the right side bar, in case the user prefers having it placed that way. The view is used for adding, checking, deleting and editing comments connected to a piece of data related to a snapshot, and it can be automatically opened whenever clicking on the comment button associated to each data entry in each view. When firstly opened, the view lets the

user acknowledge its purpose by displaying the string "Here you can view and edit comments". When a piece of data is selected, the view refreshes and the user can act on it.

4.2 Main functionalities

Resuming what was previously presented in the previous chapter, at the end of the designing step four main functionalities have been selected, those being:

- Creation of diaries and snapshots of three possible types (code, file, project).
- Automatically gather and save project related data to the snapshots (file tree, command line scripts, dependencies).
- Ability to add comments to each individual piece of data in a snapshot (snapshot itself, file in the file tree, dependency, command line script executed).
- Show the data related to a snapshot directly in the IDE.

The first feature has been developed by using the three different buttons in the "New Diary" view of the left side panel. Each button lets the user start the creation of a new chosen type diary. Despite the type of diary chosen, the tool will ask the programmer to insert a new title for the diary and then an optional title for the snapshot [4.2]. Once this pieces of information are given, the tool will collect the project related data that will be associated to the snapshot. In particular, it will generate the file tree, it will retrieve the command line scripts previously launched and the relative outputs, and get the project dependencies.

Then, depending on the type of diary chosen, the tool will gather the appropriate data for each type. In particular, if the code type diary has been selected, the highlighted section of code will be retrieved and saved. In the case of a file type diary, the tool will save the entire content of the opened file. Once all this data is gathered, the snapshot and diary data structures are created and saved locally in Visual Studio Code. As soon as this is accomplished, the "Diaries" view updates itself showing the just created Diary. Each type of Diary can be distinguished by its icon: a code diary is characterized by the "</>" symbol, a file diary has an associated file icon, and a project diary has a folder icon attached to it [4.3].

By hovering over a Diary, a plus button shows up. This button can be used to create another snapshot associated to the selected Diary. User just need to press it and the process of creating another snapshot starts. This time, the tool will ask the user just the optional name of the snapshot, then it will collect the necessary data, create the snapshot data structure and update the diary data structure with the new snapshot.

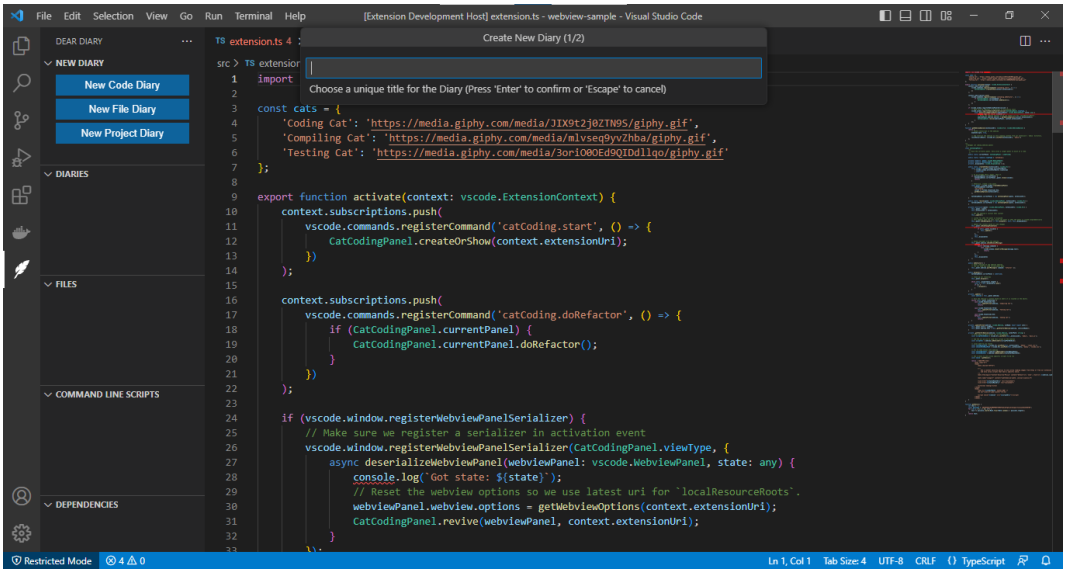

Figure 4.2: New Diary creation process started: a Diary title needs to be inserted

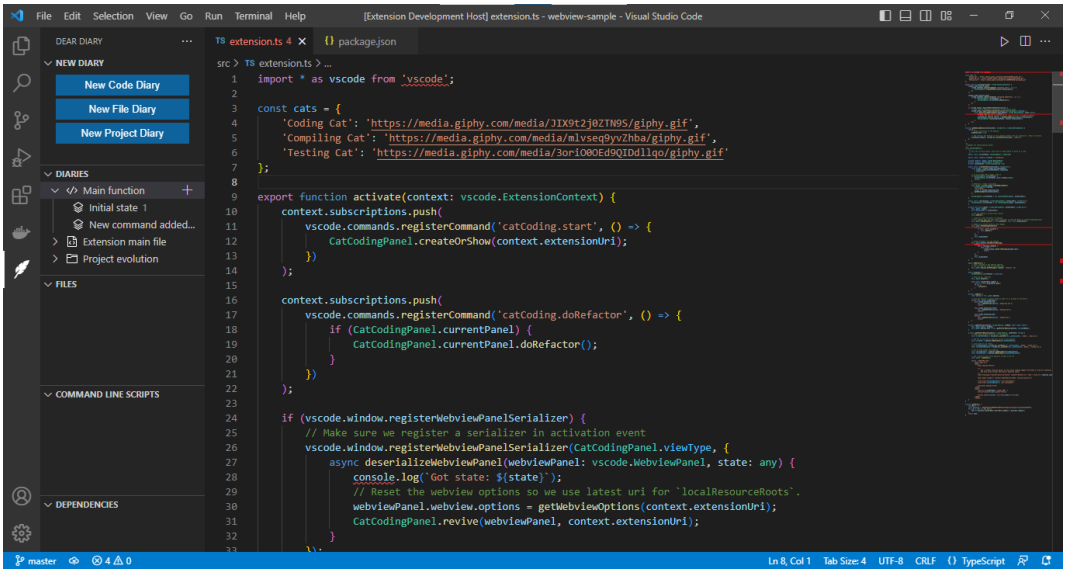

Figure 4.3: The "Diaries" view shows all the previously created diaries and the associated snapshots

By clicking on a Diary, the corresponding snapshots will be displayed underneath. Selecting one, the correlated data is shown in the various views: the "Files" view will show the project file tree, the "Command line scripts" view will show the scripts launched, the "Dependencies" view will show the project dependencies [4.4]. In

particular, when opening a code or file snapshot, the file from which the snapshot has been taken, in addition to all the folders containing the files, are highlighted in the "Files" view.

Moreover, when a script in the "Command line scripts" view is selected, the tool will open a new text file showing the command launched and the associated result [4.5].

Additionally, depending on the type of snapshot opened, a different behavior applies. In particular, consulting a code or file snapshot, the tool will open a new file showing the piece of code or the file content captured at the moment of taking the snapshot as shown in Figure 4.4. On the over hand, by opening a project snapshot, the entire project will be brought back to the version of the project at the moment of the snapshot being taken.

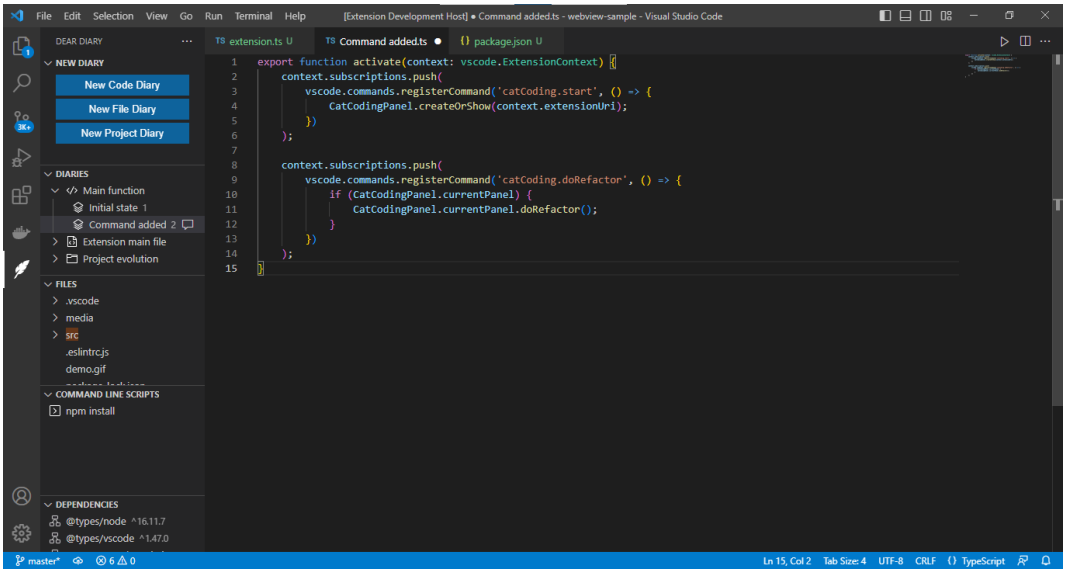

Figure 4.4: Opened snapshot

Once a snapshot is selected and the views are updated with its associated pieces of data, the user can insert his own insights by adding comments to each piece of data. This can be achieved by pressing on the comment button associated to each snapshot, file, command line script, and dependency. In this case, the "Comment" view, present in the bottom panel or optionally in the right side bar, updates itself to show the user interface through which the user can insert the comment. In particular, the source of the comment is described: the diary title, the snapshot title and the title of the associated piece of data (e.g. for a file the name of the file is shown). Then, in case a comment was previously been added, it is displayed just underneath the comment source pieces of information. Lastly, a button to edit the comment is shown. The edit button allows to edit a previously inserted comment

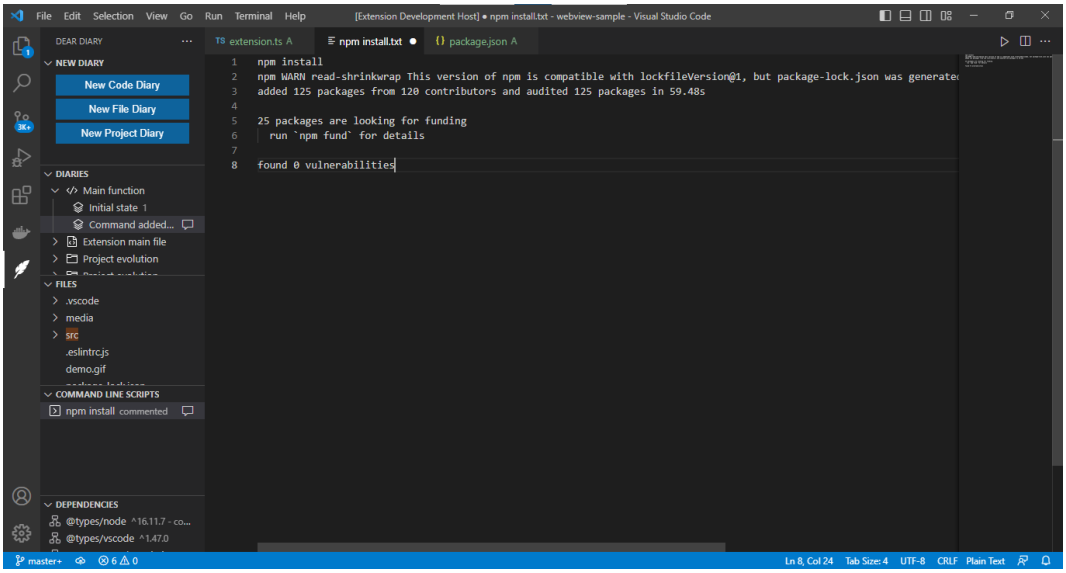

Figure 4.5: Consult command line script saved in a snapshot

but also to add one in case no previous comment has been created. By pressing the button a text box appears with the comment (in case the user is commenting for the first time, the box will be empty). This is pictured in Figure 4.6. Then, by pressing the save button the user saves the inserted text. A cancel button is also present in order to cancel the comment edit.

To signal the presence of a comment associated with a piece of data, next to its entry in the proper view and next to the relative snapshot name, a "commented" tag is added.

The tool also allows the programmer to see all the comments associated to a snapshot by opening the comment section associated to the snapshot itself. In particular, the interface shows the comment of the snapshot and all its related comments underneath, specifying their origin, as shown in Figure 4.7.

4.3 Technical choices and limitations

While progressing in the implementation of the tool's functionalities, our initial design decisions have been changed because of the restrictions of the developing environment.

First of all, extensions for Visual Studio Code are developed through the Typescript programming language so, naturally, the majority of the tool has been written in Typescript.

The development started by following the Visual Studio Code's "Your first

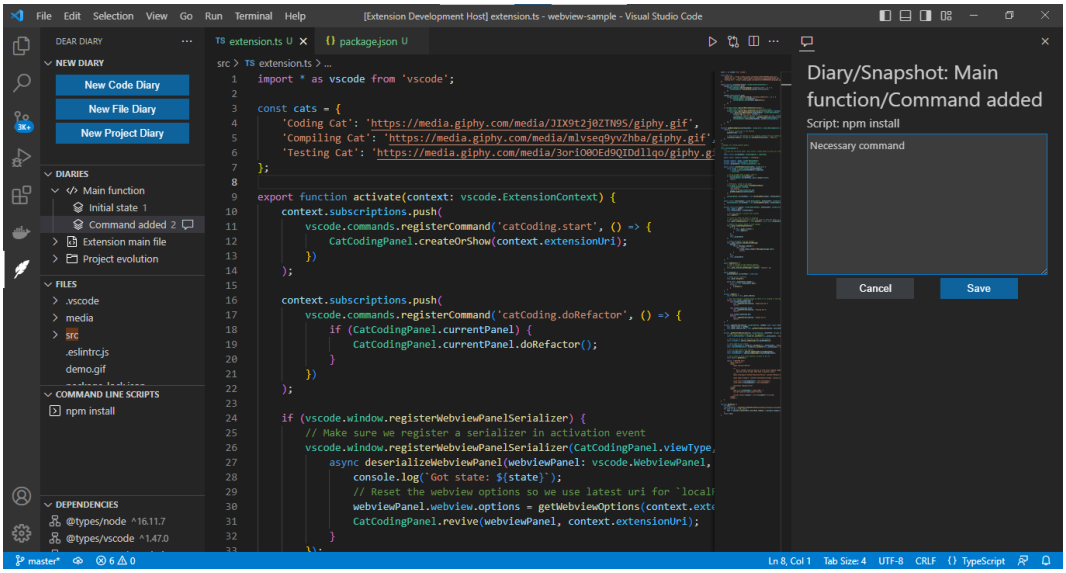

Figure 4.6: Add or edit comments

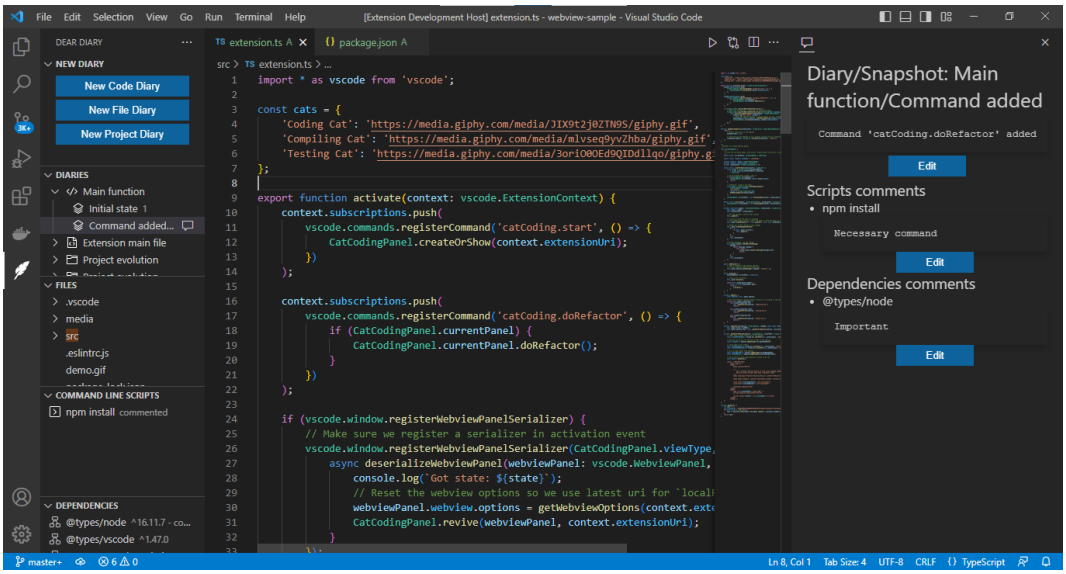

Figure 4.7: The comment view associated to a snapshot shows all the comments included in a snapshot

extension" tutorial [22]. In this way, the environment to develop the extension has been set-up.

We used the official documentation of Visual Studio Code and its official APIs [23] to getting started with the development.

Extensions are built through the use of asynchronous functions called "commands". The application will work around the set-up and the calling of this commands 4.8.

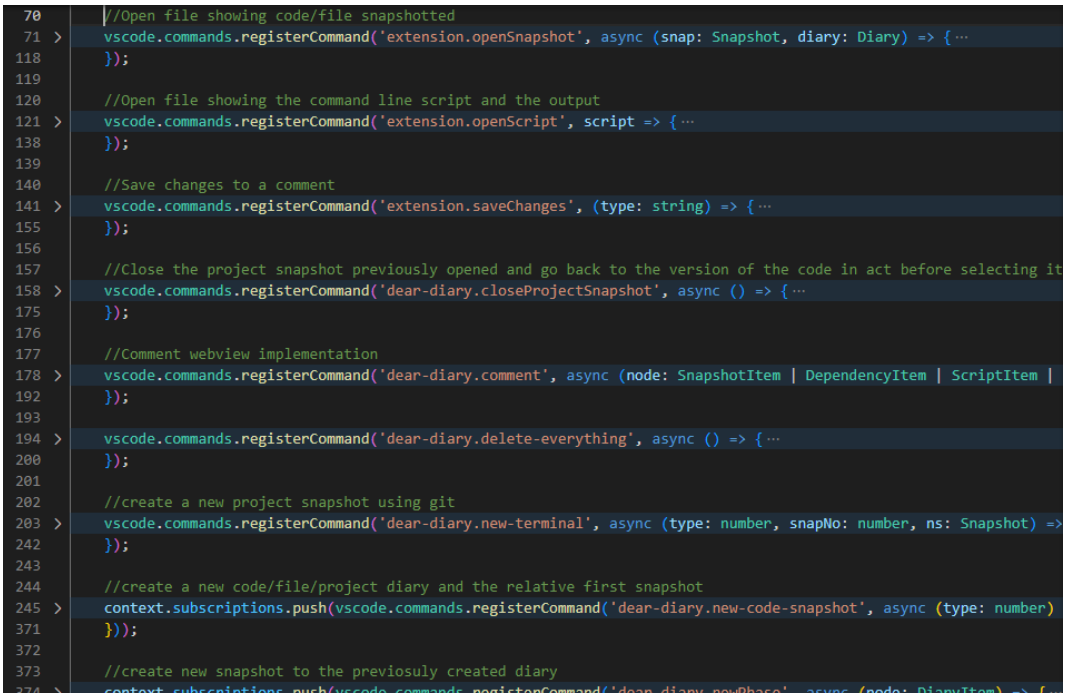

Figure 4.8: Some of the commands implemented in the tool

We began the development of our extension with displaying the first five main section:

- The one with buttons that would have allowed to create the diaries
- The section to display all the diaries and related snapshots
- The sector to display the file tree
- The segment where the list of command line scripts executed would be shown
- The module where all the project dependencies would be listed

We decided that aforementioned sections, called "views", would be placed in the left side bar in order of importance.

Then, we firstly focused on the "New diary" view. In order to start the diary creation process of each type of diary, we opted to use three different buttons. Because normal views in Visual Studio Code contain Tree Views, we had to design a Webview View in order to display the interface with the buttons we designed.

The webview API [24] allows developers to implement fully customizable views within Visual Studio Code. Webviews are also used to create complex user interfaces whenever Visual Studio Code's native APIs are found to be inadequate.

Webviews are built through the usage of a HTML string for its base appearance, and then its functionalities are managed by a separate JavaScript function.

The process of creating diaries and snapshots of type code and file was then implemented. The data structures were created, and so the operation of saving the data acquired.

In particular, the extension utilizes five main data classes 4.9:

- Diary
- Snapshot
- Resource
- FSInstance
- ResCommented

The Diary class is used to store diaries data, such as: its type, its title, and the array of associated snapshots.

The Snapshot class contains the data associated to snapshots, those being: its title, the piece of code snapshotted, in case of a code or file snapshot, or the code of the git commit associated to the snapshot, the number of comments related to the snapshot, the comment associated to the snapshot itself, and all the three arrays containing the lists of command line scripts, the dependencies, and the file tree.

Resource is the data class used to store dependencies and command line scripts data. It stores two strings, which represent the library and the library's version number in case of a dependency, or the command and its outupt in case of a command-line script.

The FSInstance data class is used to store the project's file tree. Each instance of this class represent a folder or a file. For this reason the class contains the type of file system entry, its name, the list of comments associated to it (in case of a file), the array of FSInstance instances (in case of a folder) which contain the content of the folder, and a flag that signals if, in case of a file, the file is the one that has been snapshoted (in case of code or file snapshots).

The ResCommented data class is used to store all the pieces of data related to a snapshot that have a comment associated to it.

The extension manages an array of diaries containing the user's documentation data. When launched, the tool gathers all the diaries previously created from the global state of Visual Studio Code and save it in the array controlled by the tool. Then, whenever a new diary or snapshot is created, the array is updated with the

new data and then the global state of Visual Studio Code, that contains a copy of the array, is updated too.

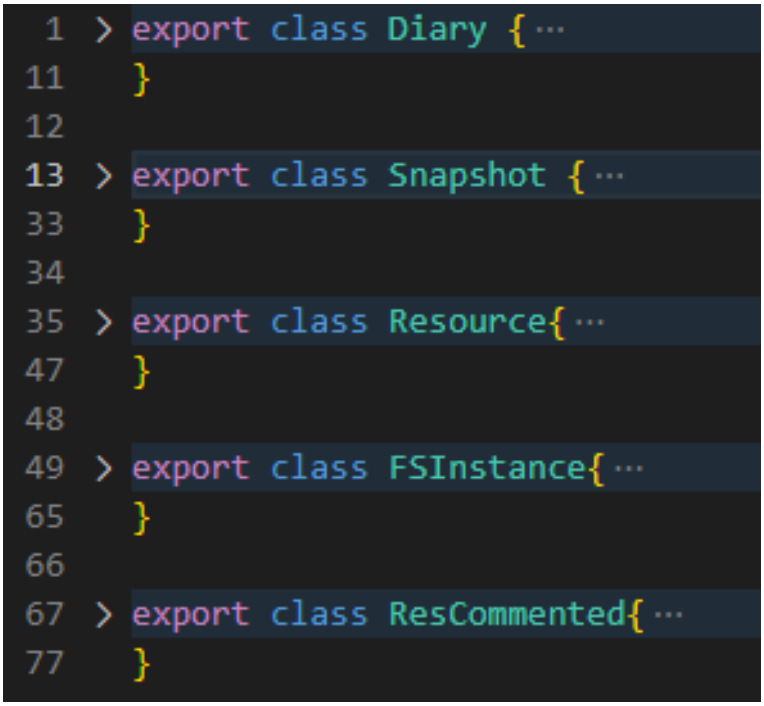

Figure 4.9: The main data structures utilized by the tool

Collecting the necessary data needed for the creation of a snapshot has been particularly tricky because of lack of the necessary API that would have allowed us to achieve our goal in an easy way. We then worked on some workarounds to overcome these hurdles.

For collecting the command line scripts launched and their relative results we had to force the extension to open the terminal of Visual Studio Code and then proceed with a "select all and copy" mechanism. By having the entire terminal output copied in the clipboard, we were able to isolate each singular command and properly store it in the snapshot data structure. This process made us dependent on the PowerShell command-line shell. As a matter of fact, in order to isolate the commands we relied on having a precise string that would appear every single time a new command had to be inserted. Because the string is different depending on the terminal in use, the tool was only available to capture the commands launched through the PowerShell.

To understand the structure of the project, in order to create the file tree, we relied on the file system API. In particular, the API allowed to scan each possible path in the project folder, letting us take note of each file and folder present.

In order to collect the dependencies we used two different approaches. In case the project had a "package.json" file, we would scan the file and retrieve and save the project dependencies. If that was not feasible, as the case of projects in Python, we scanned the files in search of imports and then saved all the libraries used.

Finally, to retrieve the content of the file or the selected piece of code that the user wanted to take the snapshot of, we again used the Visual Studio Code APIs.

For creating a diary, a special command has been created. Moreover, each of the previously described steps needed to gather the necessary project data connected to a snapshot, is accomplished by a function or another command.

Then, another special command has been created for the creation of singular snapshots associated to a previously created diary. The command is triggered whenever the user clicks on the view action button (the plus button) associated to the diary. The relative command function is similar to the one implemented for the creation of a diary, since many steps are equal. All the data connected to the snapshot is collected and the only difference is the insertion of the packaged snapshot in the previously created diary structure instead of inserting it in a newly created diary.

The view containing the lists of diaries and snapshots have been later developed and so were the "Files", "Command line scripts", and the "Dependencies" views. Each of these views has an associated class that is in charge, given an array of the data, of displaying the given data in the view.

Finally, we worked on the project type snapshot creation. Earlier in the process we were following a similar approach to the ones used with the code and file snapshot types. Specifically, we wanted to save locally the content of each file in the project the way we saved the content of a file or a piece of code in the other two types of snapshot. We later realized that this mechanism would not fit well in larger projects, making the tool too slow to be properly used.

We then opted to use Git given its reliability and ease of use. Moreover, the Git commands provided already the service that we wanted to re-create: saving all the files in a project for future reference. Utilizing the version control tool presented us another challenge. There was no way to use the features provided by the Git version control tool through the native Visual Studio Code APIs. So we opted for a function that, given a command in the string format, it would send the command to the terminal and then return its results. Through this mechanism we were able to efficiently utilize the Git features to both save the project content in the moment of taking a new snapshot and retrieve this content whenever a previous snapshot was being opened. In this way, to each snapshot taken a new commit was created and the associated code was saved in the snapshot data structure. Whenever a snapshot was opened, the tool would send the checkout command to Git in order to switch to the related commit.

Lastly, we worked on the comment feature. At first we aimed at create a user

interface that would pop-up directly in the code editor. The APIs only allowed to have this interface associated to specific lines in the code, which did not worked well if considering a project or file snapshot were the user might have wanted to comment on the entire file or project. Moreover, using this solution would have prevented the ability to add comments also to command line scripts and dependencies as previously designed. For these reasons we opted on using a Webview View to show the comment interface.

Because the left side bar was already cluttered, we decided to develop the comment view in the bottom panel with the option for the user to drag the view into the right side bar. We developed the view to show the Diary and snapshot titles and the name of the associated piece of data to which the comment is being added. An edit button would allow a text box to appear and let the user edit or delete a previously added comment, or simply add a new one. Once the changes are made, by pressing the "Save" button the changes would be stored.

4.4 Usage example

To better understand how the Dear Diary tool presents itself and how it works, an usage example is described. The scenario is based on a novice developer that want to develop her first React application. The various steps in the development are now described.

Angela is a novice developer who wants to develops its very first React application. Her favorite IDE is Visual Studio Code and she opens it as she is ready to start coding. Because she still needs to learn how to use the React library, she decides to firstly follow a "getting started" guide found on the internet.

The first steps described in the guide Angela is following involve the installation and set-up of the execution environment and its dependencies (Node and npm, in this case). These installation and set-up steps are performed by downloading and running installers in the machine. Then a series of command-line instructions with multiple parameters are executed.

Being a novice React developer, these command-line instructions are new to Angela so she follows her guide to the letter. After completing these first steps, she manages to create her first React application entitled "My first web app". When trying to run the application, everything goes well. Being enthusiastic of this first achievement she reached, she decides to document this first step. She opens the Dear Diary extension directly from Visual Studio Code by clicking on the feather icon in the Action Bar and she decides to create her first project Diary through which she saves the current state of the application. She then proceeds to click on the "New Project Diary" button in the "New Diary" view in the left side bar.

A pop up shows up at the top of the interface asking Angela to insert a title for the new Diary she's creating. She inserts "Main development steps" as the title and then the tool asks her to insert an option title for the snapshot. She then inserts "Initial state".

When creating the project diary and the first associated snapshots, the tool automatically collects a series of project-related data. Specifically, it makes a copy of all the command-line scripts executed, the project dependencies, and the project structure with its folders and files. Moreover, the tool creates a local commit of the currents state of the project. After all this data is collected and packaged in the associated data structures, this first diary shows up in the "Diaries" view.

Angela tries to make some changes to her new application and then tries to run it again. This time though, she gets an error that she does not quite understand. Naturally, she seeks for help on the internet and finds the solution to her problem: she needed to execute the application from the project root. She wants to remember this error and the relative solution that she found, so she takes a new snapshot which will be attached to her project diary previously created. Once the snapshot is created and is shown in the "Diaries" view, she opens it. All the snapshot's associated data are shown in the "Files", "Command line scripts", and "Dependencies" view. She then opens the comment view by clicking on the comment icon next to the command line script entry that previously gave her the error. The associated comment view shows up, and at that point she enters an explanation of the error and the relative solution in her own words and then she inserts the link of the website from which she found out the solution.

From there, Angela implemented an header for her new web application. She created a basic header, with just an icon and a title. Not being satisfied with the result, she wants to further edit it in order to make it more aesthetically pleasing and more functional overall. She also wants to remember where she started since this first component creation was an small achievement to her. Since the only additional component that she implemented from her last snapshot is the header, and because she's now focusing on just this specific component, she does not find the necessity to create a new project snapshot. Instead, she decide to create a new code diary in order to track the evolution of the header as she further edit its design and functionalities. So she proceeds to create a new code diary by pressing the associated button in the "New Diary" view. Then, she enters "Header evolution" as her diary title and "Basic header created" as her snapshot title. Then, she adds a comment to this just-created snapshot. She clicks on the associated comment button and when the relative comment view shows up, she proceeds to insert her reasoning for the snapshot, explaining the achievement of creating this first small component, and then further shares how she would like to change it next.

After further developing her web component, she reaches a satisfactory state. She then proceeds to create a new snapshot associated to her previously created

code diary that she entitled "Header evolution". After taking the snapshot, she opens the previous snapshot created and she compares the differences between her first header implementation and what she added and what she carried off. Being proud of work she did, she adds a comment to the just-added code snapshot. She explains what she added since she created her basic header and further explained which was the purpose of some tricky functions she used.

Angela returns to develop the main body of her application. She starts to design the structure she wants for her web app, and carries on with the creation of other components and other files in the project that will contain the functions used to create the aforementioned components. In order to keep track of this expanding number of components and their positioning in the project, Angela creates a new file diary through which she saves the content of her main file called "app.js". She entitles this new diary "Application structure" and then she inserts "Base structure" as the tool asks her for a snapshot title. Once the snapshot is created, she opens it and carries on with the insertion of various comments. First, she opens the comment interface for the file snapshot created and describes the general structure of her web app. Then, she continues with describing all the functions called in the main function in "app.js", their names, their functionalities, and which components their associated to. After that, she proceeds in commenting all the files she added containing the functions that return a component of her application. So she starts adding comments to each file by clicking on the comment button associated to these files that she finds in the file tree showed in the "Files" view. For each file, she describes the main function, the component the main function returns, and the general structure of this component and its function.

Finally, while she carries on with the development of her React application, Angela wants to document how her code is linked to a given architectural decision. In order to achieve this, she adds a series of explanations in which she describes the reasons behind her implementation, and continues by adding links to the resources she consulted, observations, reminders, and general notes. These explanations can be associated with any type of data collected by the tool. This means that Angela has the freedom to associate any of this comments to any snapshot, file, command line script, or dependency. This granularity in the process of adding a comment leaves Angela the flexibility to explain in details and in a structured way the whole development process.

From that point forward, Angela can create several diaries and associated snapshots each of which contains the related files, command line scripts, and dependencies. To each of these snapshots, Angela can carry out any of the previously described actions.

The presented scenario shows just a possible example of how the Dear Diary extension can be used to create seamlessly a more complete project documentation.

This type of documentation not only lets a future programmer timely understand the project structure, but also lets him perceive the design decision that led to that structure. Moreover, the developer has also the possibility to check out how the project evolved over time and get a reasoning of some design decisions made at a certain point in the development process. These features make Dear Diary a very powerful tool for the creation of documentation that not only is more complete, but can also describes the whole development process behind a project.

Chapter 5

User study

Once the tool has been developed, a test was conducted to assess its usability, helpfulness, and its ability to encourage novices to create documentation. In particular, we wanted to evaluate if, by letting novices create pieces of documentation seamlessly through the tool, they would create documentation more frequently.

5.1 Changes applied to the tool

We wanted to focus our attention on the core function of the tool. Thus, Dear Diary has been slightly modified for the user study. In particular, the extension has been slimmed down with a focus on its main functionalities. For this reason, the concept of the three different types of diaries, with their related snapshots, have been remove, leaving the ability to take project snapshot only. In this way, the user would just take snapshots, and the step of creating a diary and then add associated snapshot has been removed to simplify the workflow.

For this reason the "New Diary" and "Diaries" view have been modified accordingly. In particular, the "New Diary" view has been renamed "New Snapshot" and it now contained just one button to take a snapshot. The "Diaries" view has also been renamed into "Snapshots". The first view shows just one button, as opposed to the three for the three different types of diaries, that would allow the user to take a new snapshot. In the second view all the previously taken snapshots are shown. These changes can be seen in Figure 5.1.

Slight UI changes have also been made in order for the tool to be tidier and more intuitive to use. Notably, the comment interface has been refined: a white contour has been added to each comment in order to be more distinguishable and properly separate comments.

The other functionalities such as gather project related data automatically, letting the user consult any piece of data collected, and allowing the user to insert

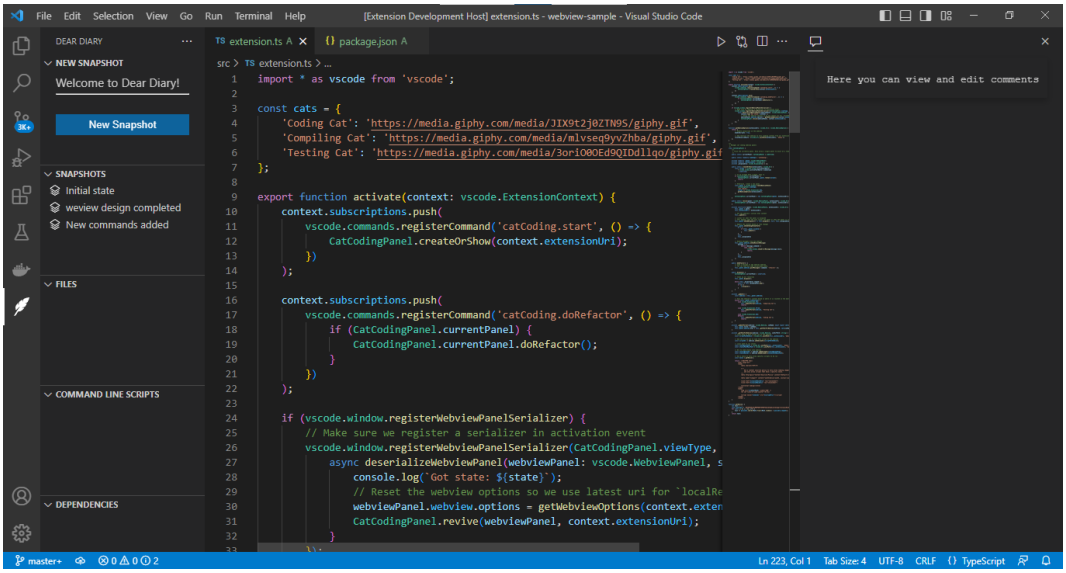

Figure 5.1: Modified tool structure

his own thoughts through comments has been left in this other version of the tool as considered key features.

The process of snapshot creation has remained the same. After clicking the "New snapshot" button, the user is invited to insert its title, which now has been promoted to mandatory and has not remained optional as before. This change was made because the Diary previously represented the core data structure that had to been unique. With the absence of the diaries' structure, the snapshot have been taken the position of core data structure and so the title needed to mandatory otherwise the user might find it difficult to search through a list of non-named snapshots.

Lastly, we added the ability for the user to add multiple comments when commenting on a file associated to a snapshot. This functionality was considered after evaluating the variety of comments that can be associated to a single file. In particular, a programmer can add multiple comments to describes the file structure and its content, functionalities, code behavior, design decisions connected to single functions or lines of code. The new interface is presented in Figure 5.2.

A complete view of the modified tool is shown in Figure 5.3.

5.2 Participants

The test was conducted on six students. An initial screening have been performed in order to select the testers. In particular, we required from our testers to be

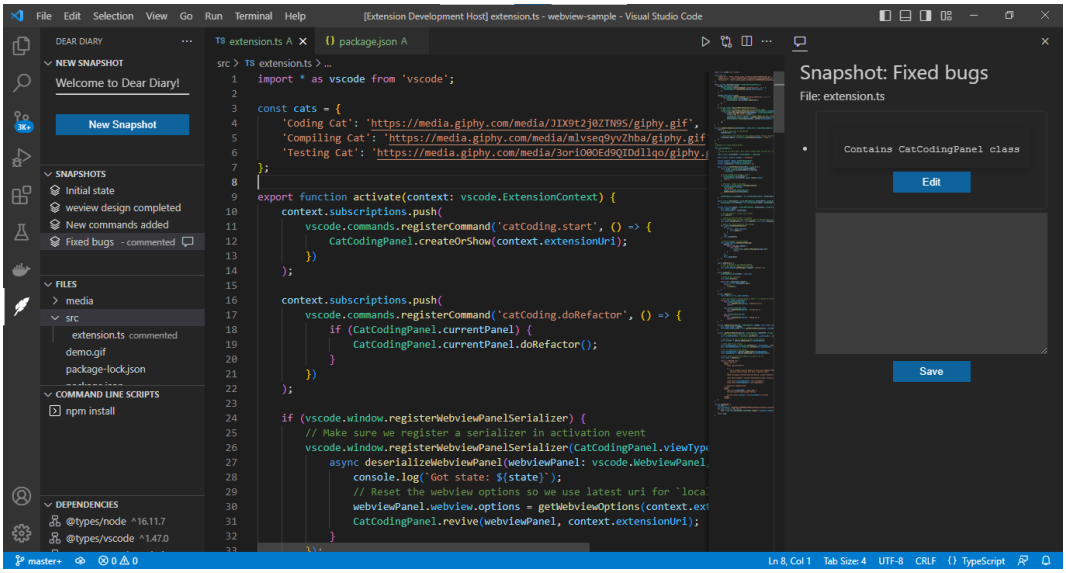

Figure 5.2: Multiple comments can be added to files

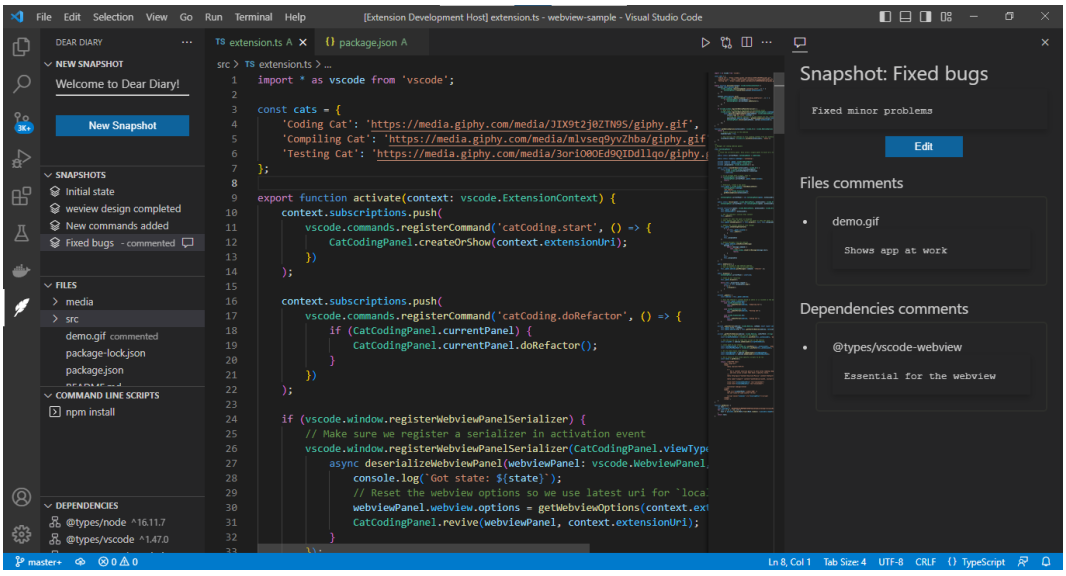

Figure 5.3: Modified version of the tool for the test

familiar with the Python programming language and that they used Visual Studio Code at least three times in the last six months.

The testers selected all had distinct levels of expertise in Python, giving us different perspectives depending on the each one's experience. More precisely, their experience in Python programming ranged from six months to five years 5.4.

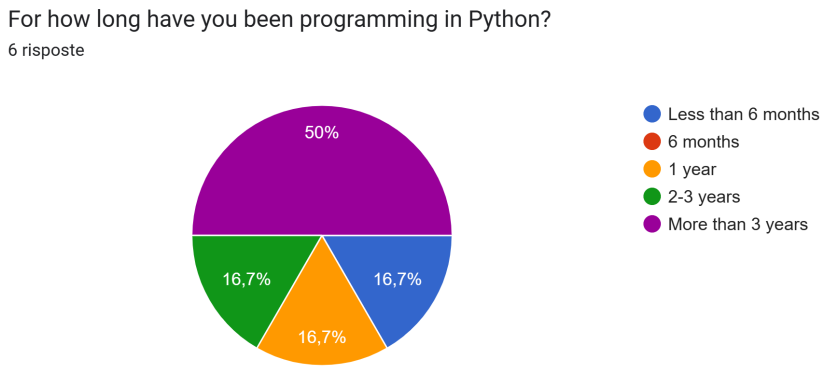

Figure 5.4: Years experience in Python programming of the testers

5.3 Method

The test was composed of four main parts:

- Tool tutorial
- First programming session
- Second programming session
- Survey

Initially, a tutorial was presented to the testers to familiarize themselves with the tool. All the main functionalities have been presented to the testers. The tutorial was performed as follows:

Each tester had to create a first snapshot to save the state of the program at the beginning of the test. Then, they needed to open the just-created snapshot. It was shown to them that the "Files" view was updated with the structure of the project files.

Consequently, they had to import the *numpy* library in the *"test1.py"* Python file where they were going to code. Then, they had to open the Powershell terminal and digit the command *"git version"*. Once all of this was done, they needed to take a new snapshot. Having the newly created snapshot visible in the "Snapshots" view, they opened it and the "Command line scripts" and "Dependencies" view were updated to show the *git version* command execute and the *numpy* library imported. In this way, it was shown to the testers all the types of data that the tool gathered when taking a snapshot.

They were then requested to click on the *git version* command in the "Command line scripts" view. Once this was done, the testers noticed that a new text file was open showing the command they previously executed and its associated returned value. In this way, it was shown that all the commands with its relative results were saved and could be consulted at any time.

At this point, they were requested to open the first snapshot taken. By doing that, the testers noticed two things. First, the view updated their content accordingly: the "Command line scripts" and the "Dependencies" view were then empty again. Second, the "test1.py" file they were working on went back to the version of the file without the *numpy* library import. So, it was shown to them that the file, when opening the snapshot, went back to how it was when they first created the snapshot.

Lastly, the comment features were presented. They were consequently requested to reopen the second snapshot so that the newer version of the file, the one containing the import, would come up. Then, they had to click on the comment button associated to the file *"test.jpg"*. This would make the comment view appear in the right side bar. The testers were asked to click on the edit button and insert the text "This is the image to be modified during the exercises"; then they needed to click the save button. In this way they learned how to add a comment. Finally, they needed to add a comment to the snapshot itself. They clicked on the comment icon associated to the snapshot and they were asked to add the comment "numpy library imported". Then, because both the just added comment and the comment previously added to the "test.jpg" file were shown, it was explained to testers that from the comment section of the snapshot they could both add/edit/delete a comment of the snapshot itself and also consult any other comment added to associated pieces of data connected to the opened snapshot.

Through this tutorial the testers were instructed on the functionalities of the tool and so they were ready for the programming sessions.

Once the testers understood how the tool worked, the first programming session was presented to them. Specifically, they needed to solve complex Python exercises 5.6, that involved image processing 5.5, while having access to the tool. Testers were allowed to use any type of documentation or source of information they liked, so searching on the internet was permitted. Moreover, they were allowed to create any type of documentation they wanted, included code comments, taking notes on a piece of paper, and creating documentation through the tool. The programming session lasted one hour.

The secondary programming session was similar to the first one: they needed to continue the previously presented exercises, but from where another tester left off, having access to any type of documentation they created. The second programming session also lasted one hour.

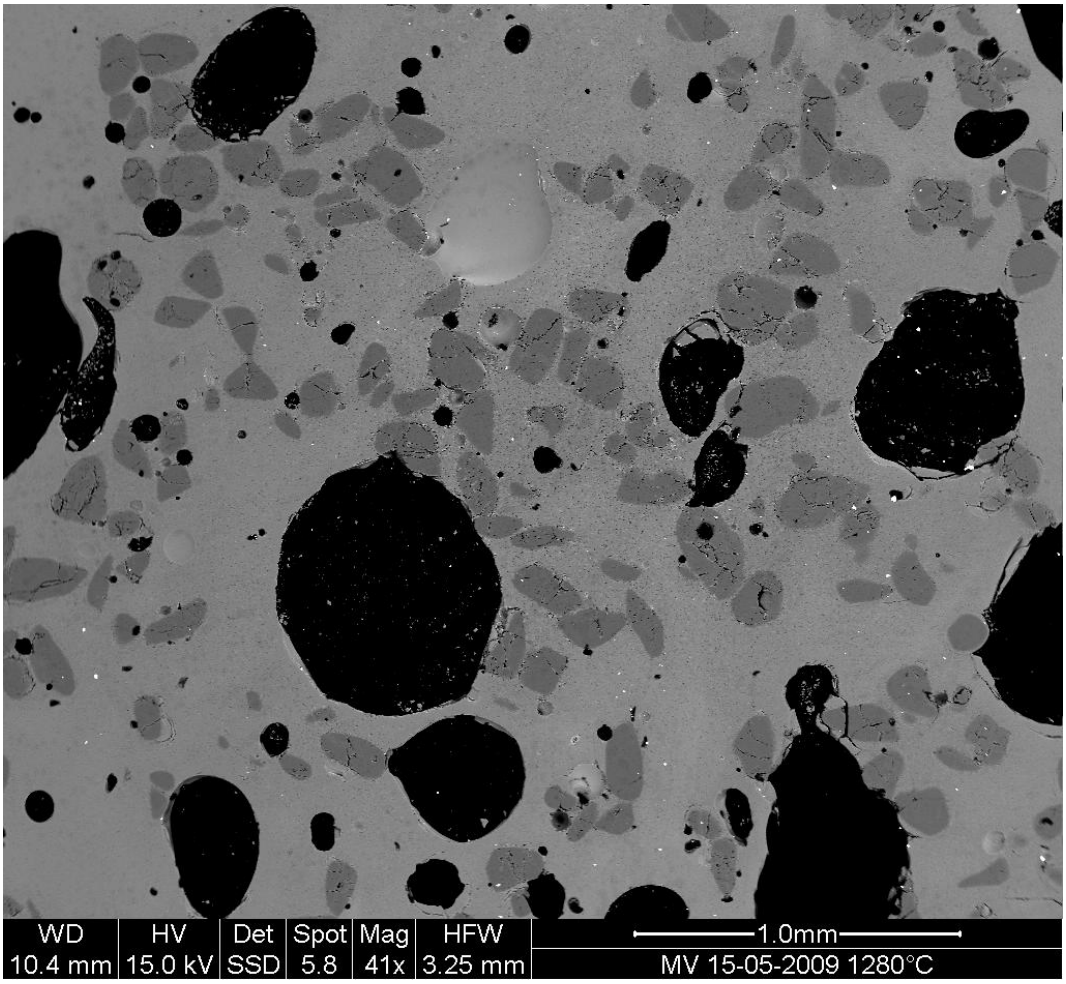

Figure 5.5: The image testers had to use to solve the exercises

Lastly, in order to gain insights and feedback on the tool, a survey, created and issued through Google Forms [25], has been presented to the testers. The survey contained both open and closed questions and it was divided into three main parts. In the end the test candidates were also encouraged to express any other consideration about the test and the tool by voice.

The first part of the survey aimed at assessing the tester's knowledge of the Python programming language as well as understanding the experience solving the presented exercises. In particular, it was asked if they have ever worked on similar tasks before and, if they did, how often they found themselves facing similar problems.

The second part of the survey focuses on documentation. Through this part we wanted to evaluate each tester's documentation creating habits. In particular it

was asked how often, when working on a project, they would create documentation, which type of documentation they created, and the reasoning behind their behaviors.

Lastly, the survey asked the users for feedback on the tool itself. Firstly, we wanted to analyze how they perceived the extension. We asked the testers to rate if they understood how to properly use the tool after the initial tutorial and how easy it was for them to learn the extension's functionalities. Next, they judged how intuitive, easy to use, and useful the tool was for them. Then, they needed to evaluate if they would keep on using the tool for personal use and how much probable it was for them to recommend Dear Diary to friends or colleagues. In the end, a series of open questions were presented in order for them to express their insights and overall thoughts on the tool. More specifically, they were asked for which goal they used the tool the most, which functionalities they found to be more useful for their tasks, which feature they found to be useless or superfluous. Finally, they were asked which functionalities they would add to the extension and how they would improve Dear Diary.

Student's test

Exercise 1

1. Open the image file test.jpg and display it. Browse through the keyword arguments in the docstring of *imshow* to display the image with the "right" orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).
2. Crop the image to remove the lower panel with measure information.
3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.
4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option: write a function that determines automatically the thresholds from the minima of the histogram.
5. Display an image in which the three phases are colored with three different colors.
6. Use mathematical morphology to clean the different phases.
7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use *ndimage.sum* or *np.bincount* to compute the grain sizes.
8. Compute the mean size of bubbles.

Exercise 2

1. Open as an array an image that you have on your computer.
2. Crop a meaningful part of the image, for example make the image a square.
3. Display the image array using *matplotlib*. Change the interpolation method and zoom to see the difference.
4. Transform your image to greyscale
5. Increase the contrast of the image by changing its minimum and maximum values. **Optional**: use *scipy.stats.scoreatpercentile* (read the docstring!) to saturate 5% of the darkest pixels and 5% of the lightest pixels.
6. Save the array to two different file formats (png, jpg, tiff)

Exercise 3

1. Create a binary image (of 0s and 1s) with several objects (circles, ellipses, squares, or random shapes).
2. Add some noise (e.g., 20% of noise)
3. Try two different denoising methods for denoising the image: gaussian filtering and median filtering.

Figure 5.6: The exercises presented to the participants during the test

Chapter 6

Results

The goal of this project was to design and develop a tool, directly implemented in the IDE, that could be used by novices and developers to seamlessly create pieces of documentation enriched with contextual data and comments, links, and thoughts of the user.

The outcome of the study we conducted seems to suggest that our goal have been reached. In particular, the results from our survey indicated that the tool efficiently helped users to create documentation more easily and frequently. In addition, one of the participants asked at the end of the test if the tool would have been publicly available in the future and if he could still use it for personal projects. He was particularly interested in the evolution of the project and he wanted to try out the complete version of the tool that also included the three types of diaries and snapshots.

Prior to the use of our tool Dear Diary, the majority of testers would make incomplete documentation (if at all) when programming, and the documentation they created was exclusively based on comments on the code. One of the testers, whom already had work experience and training to create documentation on files, was our only exception in this study. On the other hand, all participants in the study took at least one snapshot each time they solved one point of the exercises, marking their achievement and then adding comments to contextualize the functionalities implemented since the previous snapshot taken.

Moreover, three out of six participants created a snapshot when firstly having complications with the compiler not finding the necessary libraries to run the code. In particular, they first created a snapshot when they encountered the issue, then adding a comment explaining the problem they were facing. Once they overcame the issue, they took a secondary snapshot and indicated in the comments how they successfully solved the problem.

In general, the users were satisfied by the simplicity and intuitiveness of the tool 6.1. They also found the tool to be particularly useful for tracking the evolution

of the project. Additionally, they appreciated the ability to easily add complex comments that could also contain notes and reminders that could be consulted later on. The granularity through which comments could be added has also been acknowledged. This behavior aligned with our first utilization visualization in which we envisioned the usage of the tool as a way to keep track of problems and the relative solutions found during the development process.

Figure 6.1: Testers' tool perception

The participants also suggested a series of new functionalities that can further enhance the tool. For example, one of the suggestions highlighted the need for the extension to share the documentation that's been created with others; a feature they collectively agreed would be especially helpful when working on a team project. They suggested two ways to achieve this goal: by exporting the documentation in a file/document or by letting users transfer the documentation from one instance of Visual Studio Code in a pc to another. Another way to achieve this goal would be the creation of a static website that would display all the documentation associated to the project.

Further developing the idea of the documentation sharing feature, a tester suggested a visibility setting per each snapshot. In particular, he proposed a way for snapshot to be signaled as private or public. This clarification would be utilized when sharing the created documentation: the public snapshots would actually be shared while the private ones would not. In this way a distinction between documentation created for the project and documentation created for the programmer developing the application forms.

Another feature request included an integration with GitHub. In particular, one of the participants expressed the potential usefulness of having the ability of directly saving all the documentation created in GitHub with the project itself.

Because the test presented involved image processing, one participant would have liked for the tool to automatically save the output of the program. In this way he could check out the images created by the code saved in a previous snapshot.

A tester also expressed the desire of being able to delete previously take snapshots that he found to be useless in the end. This would unclutter the "Diaries" view, making it faster the search of effectively useful snapshots.

Lastly, a tester suggested the integration of GitHub issues in the tool itself.

Chapter 7

Conclusions

In this work, Dear Diary was introduced, a Visual Studio Code extension that enables beginners to record their development process and add self-explanatory notes about the code, the execution outputs, and project versions. These insights explain their decisions, the mistakes they made, the consulted sources, and pieces of advise to keep in mind or impart to others. We described the tool's look and functionality. We also explored the factors that influenced and directed the creation of Dear Diary.

At the beginning of this work, the results of the research made in literature and the consequent feature evaluation has been presented. Then, the focus shifted to the design and development of the "Dear Diary" tool, through which we aimed at seamless creation of better documentation. Finally, in the last part of this work, we described the user study we conducted, involving six participants that tried out the tool, were shown, and presented the relative results.

The aforementioned study showed us that novices often neglect to create pieces of documentation when working on a project. The documentation created is almost entirely composed by comments on the code, which reveals to have poor utility when used as future reference.

Through the creation of "Dear Diary" we achieved a way for novices and experienced programmers alike to create better documentation seamlessly. We think that our technology may transform documentation into a useful asset for them and other developers by depending on self-explanation and integrating into the working environment.

The tool allows developers to easily create documentation within the IDE, allowing them to follow the project's development and contribute their own thoughts on how they arrived at their development milestones. Developers may use the tool to produce "snapshots," documentation artifacts that preserve sections of code, files, or complete projects as well as contextual and helpful data like dependencies, the file tree, and performed command line scripts. These snippets are compiled into

bigger structures known as "diaries". Moreover, the developer may add links and comments to any registered data that is included in a "snapshot", further enhancing the documentation that is produced.

In order for the generated documentation to usefully support novices in the future, our strategy aimed to make it simple for beginners to document their development process and to encourage them to do so. The goal of our strategy was to record the entire development process. In particular, our approach aimed to record and document the project's state as it changed rather than at a single moment in time. Similar to this, Dear Diary sought to avoid being code-centric by allowing users to make notes about files, command-line processing results, and project dependencies in addition to code snippets.

7.1 Future work

Starting from the results obtained by this work, future studies can focus on improving the user experiment through the use of a larger testing pool that also utilizes a control group to better highlight the disparities in documentation creation behaviors between programmers using and not using the tool. The future test could also employ the complete version of the tool containing the three types of diaries and snapshots developed.

Another study can focus on the qualitative evaluation of Dear Diary in the context of a university course on web development to examine its usability and ascertain how well it aids beginners in recording their work. In order to assess the students' readiness to utilize the tool and the qualities of the resulting documentation artifacts, we specifically foresee a case study undertaken during the execution of a evaluated project developed by the students divided into small teams throughout the Politecnico di Torino's Web Application I course. This study would help us understand the tool effective usefulness in supporting novices when working on larger scale projects.

A series of adjustments can be made to the tool in the future. More precisely, the functionalities discussed in Chapter 3 that have not been judged as essential can also be reevaluated and possibly implemented in a future versions of the extension.

Furthermore, snapshots and diaries can be made project-dependent. By doing so, opening a project in Visual Studio Code and then opening the extension, would make only the diaries and snapshots related to the just-opened project available. This solution would unclutter the view that shows the list of diaries and snapshots, which can become quickly crowded by creating a series of diaries and snapshots in different projects.

Finally, the tool can also be further improved by adding sharing functionalities for the documentation created. Additionally, an integration with GitHub can

be employed for users to save into their project repositories all the associated documentation. This integration would also enable the incorporation of GitHub issues in the tool itself.

These last mentioned new functionalities can be found particularly handful when considering the aforementioned case study following the tool's usage by the students working on the course project along the semester.

Bibliography

[1] Stephen Oney and Joel Brandt. «Codelets: Linking Interactive Documentation and Example Code in the Editor». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, pp. 2697–2706. ISBN: 9781450310154. DOI: 10.1145/2207676.2208664. URL: https://doi.org/10.1145/2207676.2208664 (cit. on pp. 1, 5, 7).

[2] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. «Using Annotations for Sensemaking About Code». In: UIST '22. Bend, OR, USA: Association for Computing Machinery, 2022. ISBN: 9781450393201. DOI: 10.1145/3526113.3545667. URL: https://doi.org/10.1145/3526113.3545667 (cit. on pp. 1, 5).

[3] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. «What help do developers seek, when and how?» In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 142–151. DOI: 10.1109/WCRE.2013.6671289 (cit. on p. 1).

[4] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. «Unakite: Scaffolding Developers' Decision-Making Using the Web». In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: Association for Computing Machinery, 2019, pp. 67–80. ISBN: 9781450368162. DOI: 10.1145/3332165.3347908. URL: https://doi.org/10.1145/3332165.3347908 (cit. on p. 1).

[5] Teemu Lehtinen, André L. Santos, and Juha Sorva. «Let's Ask Students About Their Programs, Automatically». In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 2021, pp. 467–475. DOI: 10.1109/ICPC52881.2021.00054 (cit. on pp. 1, 5, 7).

[6] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. 2nd. Addison-Wesley Professional, 2010 (cit. on p. 4).

[7] Martin P. Robillard et al. «On-demand Developer Documentation». In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 479–483. DOI: 10.1109/ICSME.2017.17 (cit. on p. 4).

[8] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. «What makes a good code example?: A study of programming Q&A in StackOverflow». In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012), pp. 25–34 (cit. on p. 4).

[9] T.C. Lethbridge, J. Singer, and A. Forward. «How software engineers use documentation: the state of the practice». In: *IEEE Software* 20.6 (2003), pp. 35–39. DOI: 10.1109/MS.2003.1241364 (cit. on p. 4).

[10] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. «When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects». In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 643–653. DOI: 10.1145/3180155.3180176 (cit. on p. 4).

[11] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. «Active Documentation: Helping Developers Follow Design Decisions». In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 87–96. DOI: 10.1109/VLHCC.2019.8818816 (cit. on pp. 4, 6, 7).

[12] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. «Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code». In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015, pp. 3–12. DOI: 10.1109/VLHCC.2015.7356972 (cit. on p. 4).

[13] Austin Z. Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. «An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior». In: *Proceedings of the 43rd International Conference on Software Engineering: Joint Track on Software Engineering Education and Training*. ICSE-JSEET '21. Virtual Event, Spain: IEEE Press, 2021, pp. 165–170. ISBN: 9780738133201. DOI: 10.1109/ICSE-SEET52601.2021.00026. URL: https://doi.org/10.1109/ICSE-SEET52601.2021.00026 (cit. on pp. 5, 7).

[14] Tom-Michael Hesse, Arthur Kuehlwein, and Tobias Roehm. «DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally». In: *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*. 2016, pp. 30–37. DOI: 10.1109/MARCH.2016.9 (cit. on p. 6).

[15] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Wang, Sang Won Lee, Walter S. Lasecki, and Steve Oney. «EdCode: Towards Personalized Support at Scale for Remote Assistance in CS Education». In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–5. DOI: 10.1109/VL/HCC50065.2020.9127260 (cit. on pp. 6, 7).

[16] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. «StoryDroid: Automated Generation of Storyboard for Android Apps». In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 596–607. DOI: 10.1109/ICSE.2019.00070 (cit. on p. 6).

[17] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. «Retrieval-based Neural Source Code Summarization». In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 1385–1397 (cit. on p. 6).

[18] Paul W. McBurney and Collin McMillan. «Automatic Source Code Summarization of Context for Java Methods». In: *IEEE Transactions on Software Engineering* 42.2 (2016), pp. 103–119. DOI: 10.1109/TSE.2015.2465386 (cit. on p. 6).

[19] Michelle Ichinco and Caitlin Kelleher. «Exploring novice programmer example use». In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015, pp. 63–71. DOI: 10.1109/VLHCC.2015.7357199 (cit. on p. 9).

[20] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. «What Would Other Programmers Do: Suggesting Solutions to Error Messages». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 1019–1028. ISBN: 9781605589299. DOI: 10.1145/1753326.1753478. URL: https://doi.org/10.1145/1753326.1753478 (cit. on p. 9).

[21] *UX Guidelines*. https://code.visualstudio.com/api/ux-guidelines/overview. Accessed: 2022-05-06 (cit. on p. 13).

[22] *Your First Extension*. https://code.visualstudio.com/api/get-started/your-first-extension. Accessed: 2022-05-15 (cit. on p. 19).

[23] *Extension API*. https://code.visualstudio.com/api. Accessed: 2022-05-15 (cit. on p. 19).

[24] *Webview API*. https://code.visualstudio.com/api/extension-guides/webview. Accessed: 2022-06-7 (cit. on p. 21).

[25] *Google Forms*. `https://www.google.com/intl/it/forms/about/`. Accessed: 2023-02-17 (cit. on p. 33).