

# *Dynamic Visual Cues for Code Comprehension and Navigation*

Bachelor's Thesis  
submitted to the  
Media Computing Group  
Prof. Dr. Jan Borchers  
Computer Science Department  
RWTH Aachen University

*by*  
*Christopher Helios*

Thesis advisor:  
Prof. Dr. Jan Borchers

Second examiner:  
Prof. Dr.-Ing. Ulrik Schroeder

Registration date: 09.01.2024  
Submission date: 10.05.2024



# Eidesstattliche Versicherung

## Declaration of Academic Integrity

---

Name, Vorname/Last Name, First Name

---

Matrikelnummer (freiwillige Angabe)  
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/  
Masterarbeit\* mit dem Titel

I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis\* entitled

---

---

---

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without unauthorized assistance from third parties (in particular academic ghostwriting). I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

---

Ort, Datum/City, Date

---

Unterschrift/Signature

\*Nichtzutreffendes bitte streichen/Please delete as appropriate

### Belehrung:

#### Official Notification:

#### § 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

#### § 156 StGB (German Criminal Code): False Unsworn Declarations

Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.

#### § 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Strafflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

#### § 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence

(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.

(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

---

---

Ort, Datum/City, Date

Unterschrift/Signature



# Contents

|   |             |
|---|-------------|
| <b>Abstract</b>                         | <b>xi</b>   |
| <b>Überblick</b>                        | <b>xiii</b> |
| <b>Acknowledgments</b>                  | <b>xv</b>   |
| <b>Conventions</b>                      | <b>xvii</b> |
| <b>1 Introduction</b>                   | <b>1</b>    |
| 1.1 Motivation . . . . .                | 1           |
| 1.2 Goal and Approach . . . . .         | 3           |
| <b>2 Related Work</b>                   | <b>5</b>    |
| 2.1 Code Summarization . . . . .        | 6           |
| 2.2 Extending IDEs . . . . .            | 8           |
| <b>3 Concept and Design Choices</b>     | <b>13</b>   |
| 3.1 Deciding on Limitations . . . . .   | 15          |
| 3.2 Determining Local context . . . . . | 18          |

|          |                                     |           |
|----------|-------------------------------------|-----------|
| 3.3      | Displaying Line context . . . . .   | 22        |
| <b>4</b> | <b>Implementation</b>               | <b>25</b> |
| 4.1      | Generating the AST . . . . .        | 26        |
| 4.2      | Preparing the AST . . . . .         | 26        |
| 4.3      | AST Analysis . . . . .              | 27        |
| 4.4      | Highlighting . . . . .              | 27        |
| <b>5</b> | <b>Evaluation</b>                   | <b>29</b> |
| 5.1      | Technical Evaluation . . . . .      | 29        |
| 5.2      | User study . . . . .                | 34        |
| 5.2.1    | Study Design . . . . .              | 38        |
| 5.2.2    | Participants . . . . .              | 41        |
| 5.2.3    | Quantitative Analysis . . . . .     | 41        |
| 5.2.4    | Qualitative Analysis . . . . .      | 44        |
| <b>6</b> | <b>Conclusion</b>                   | <b>47</b> |
| 6.1      | Discussion . . . . .                | 47        |
| 6.2      | Summary and Contributions . . . . . | 49        |
| 6.3      | Future Work . . . . .               | 49        |
| <b>A</b> | <b>User Study Documents</b>         | <b>51</b> |
|          | <b>Bibliography</b>                 | <b>79</b> |







# List of Figures

|     |                                       |    |
|-----|---------------------------------------|----|
| 2.1 | <i>Code Bubbles</i>                   | 9  |
| 2.2 | <i>Patchworks</i>                     | 9  |
| 2.3 | <i>Mylar</i>                          | 11 |
| 2.4 | <i>Stackplorer</i>                    | 12 |
| 2.5 | Colored Structure Outline             | 12 |
| 3.1 | IDE Usage Statistic                   | 16 |
| 3.2 | Programming Language Usage Statistic  | 16 |
| 3.3 | Simple Tracing Example                | 20 |
| 3.4 | If-statement Tracing Example          | 22 |
| 3.5 | Example of prototype Visuals          | 23 |
| 5.1 | Unknown Horizons                      | 30 |
| 5.2 | Object method call                    | 32 |
| 5.3 | Bug: Control structure head           | 33 |
| 5.4 | Bug: Initialization Change Undetected | 33 |
| 5.5 | Bug: Special Case in current line     | 34 |

---

|      |  |    |
|------|--|----|
| 5.6  | Example: Initial Assignment . . . . .    | 35 |
| 5.7  | Example: Side Effects . . . . .          | 36 |
| 5.8  | Example: Class Attributes . . . . .      | 37 |
| 5.9  | Example: Iterations . . . . .            | 37 |
| 5.10 | Example: Scope Up and Down . . . . .     | 39 |
| 5.11 | Study Task Pictures . . . . .            | 40 |
| 5.12 | Study: Individual Task Results . . . . . | 42 |
| 5.13 | Study: Individual Task Results . . . . . | 43 |
|      |  |    |
| A.1  | Welcome and consent . . . . .            | 52 |
| A.2  | Demographics form . . . . .              | 53 |
| A.3  | Task Explanation . . . . .               | 54 |
| A.4  | Test task . . . . .                      | 55 |
| A.5  | Comments . . . . .                       | 56 |
| A.6  | Thank you . . . . .                      | 57 |
| A.7  | CA-1 . . . . .                           | 59 |
| A.8  | CA-2 . . . . .                           | 60 |
| A.9  | CA-3 . . . . .                           | 61 |
| A.10 | IA-1 . . . . .                           | 62 |
| A.11 | IA-2 . . . . .                           | 63 |
| A.12 | IA-3 . . . . .                           | 64 |
| A.13 | It-1 . . . . .                           | 65 |

---

|                     |    |
|---------------------|----|
| A.14 It-2 . . . . . | 66 |
| A.15 It-3 . . . . . | 67 |
| A.16 SD-1 . . . . . | 68 |
| A.17 SD-2 . . . . . | 69 |
| A.18 SD-3 . . . . . | 70 |
| A.19 SU-1 . . . . . | 72 |
| A.20 SU-2 . . . . . | 73 |
| A.21 SU-3 . . . . . | 74 |
| A.22 SE-1 . . . . . | 75 |
| A.23 SE-2 . . . . . | 76 |
| A.24 SE-3 . . . . . | 77 |



# Abstract

More than half of a developer's time is spent on code comprehension and navigation. This is due to existing tools not sufficiently supporting developers. Various promising approaches exist to mitigate this issue. However, we identified a gap in the research for a dynamic approach supporting developers on a local level.

This bachelor's thesis presents our idea for an approach to fill this gap, as well as an accompanying prototype. It provides developers with local context. To do so, it traces code dependencies and marks them within the IDE. Thus we hope to enable developers' exploration of the code within methods. This is so far lacking from other approaches. We conducted two evaluations of our approach. First, a technical evaluation was conducted on the prototype. It shows how well the prototype performs on real-world code, including its limitations. Second, we present a user study. The results answer some open questions that arose. These questions concern the topic of what even constitutes a useful local context for developers. Based on the evaluation results, we conclude our approach has merit. However, we also identify various issues that need to be solved to make the approach viable.



# Überblick

Mehr als die Hälfte der Zeit von Entwicklern wird aufgewendet für Codeverständnis und Navigation. Dies ist der Fall, weil existierende Werkzeuge Entwickler nicht adäquat unterstützen. Es existieren verschiedene vielversprechende Ansätze, um dieses Problem zu mildern. Wir haben jedoch eine Lücke in der Forschung für einen dynamischen Ansatz identifiziert, welcher Entwickler auf einer lokalen Ebene unterstützt.

Diese Bachelorarbeit präsentiert unsere Idee für einen Ansatz, welcher diese Lücke füllt, als auch ein dazugehöriger Prototyp. Dieser versorgt Entwickler mit lokalem Kontext. Um dies zu erreichen, verfolgt er Codeabhängigkeiten und markiert diese in der Entwicklungsumgebung. Damit hoffen wir Entwickler zu ermöglichen den Code innerhalb von Methoden zu erforschen. Dies fehlt bisher in anderen Ansätzen. Wir haben zwei Evaluationen unseres Ansatzes durchgeführt. Zuerst eine technische Evaluation, durchgeführt auf dem Prototyp. Dies zeigt, wie gut der Prototyp, in Anbetracht seiner Limitationen, auf realistischem Code funktioniert. Zweitens präsentieren wir eine Benutzerstudie. Die Resultate beantworten einige offene Fragen, welche aufkamen. Diese Fragen betreffen das Thema, was ein für Entwickler nützlicher lokaler Kontext überhaupt beinhaltet. Basierend auf den Ergebnissen der Evaluation folgern wir, dass unser Ansatz Wert besitzt. Jedoch identifizieren wir auch verschiedene Probleme, welche gelöst werden müssen, um den Ansatz brauchbar zu machen.





# Acknowledgments

First and foremost I want to thank my advisor Adrian Wagner for taking the time to give me helpful feedback on the progress of my thesis every week and for guiding me through the entire process.

Towards Prof. Dr. Borchers and Prof. Dr.-Ing. Schroeder I want to express my gratitude for offering up their valuable time to examine my thesis.

Additionally, I want to thank all the participants of the user study. Their extensive feedback was very insightful.

Lastly, I want to thank anybody else who supported me in the creation of this thesis, especially my family and friends.



# Conventions

Throughout this thesis we use the following conventions:

- The thesis is written in American English.
- The first person is written in plural form.
- Unidentified third persons are referred to with gender-neutral pronouns.
- Study participants are referred to as 'Px' with  $x \in \{1, \dots, 21\}$  to maintain their anonymity.

Short excursuses are set off in colored boxes.

**EXCURSUS:**  
Excursuses are set off in orange boxes.

Where appropriate, paragraphs are summarized by one or two sentences that are positioned at the margin of the page.

This is a summary of a paragraph.

Source code and implementation symbols are written in typewriter-style text.

`myClass`



# Chapter 1

## Introduction

### 1.1 Motivation

Nowadays the influence of computer code permeates almost every aspect of our lives. We connect with friends online, entertain ourselves with digital media, and use products designed on a computer. The underlying code bases can consist of [millions of lines of code](#)<sup>1</sup>. Developers are responsible for maintaining these. To do so, they need to comprehend and navigate vast and complex bodies of code. Failing to do so results in faulty code, that can cause severe damages (Glass [1981]; Zhivich and Cunningham [2009]).

Modern life depends on developers' ability to maintain code.

Modern integrated development environments (IDEs) like [Eclipse](#)<sup>2</sup> or [Visual Studio Code](#)<sup>3</sup> (VS Code) aim to support developers. Developers benefit from IDE features, such as the Call Hierarchy (Krämer et al. [2013]) or code completion (Han et al. [2009]).

IDEs help developers.

Researchers explore how different factors influence code comprehension and navigation. They aim to further improve these tools. Ko et al. [2006] investigated how developers explore unfamiliar code. They discovered devel-

What hinders code comprehension and navigation?

---

<sup>1</sup> <https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/>

<sup>2</sup> <https://eclipseide.org/>

<sup>3</sup> <https://code.visualstudio.com/>

opers use cues within the code to search for task-relevant parts. Searches often fail due to misleading or missing cues. A study by Xia et al. [2018] corroborates these findings. Their study suggests insufficient comments and documentation hinder code comprehension. Results from LaToza et al. [2006] explain why this might be the case. They found developers heavily rely on mental models of the code. This implicit knowledge is fragile compared to written-down documentation. So developers have to recover it often. Insufficient documentation forces them to explore the code or ask colleagues. Other factors that hinder code comprehension include nesting of code (Johnson et al. [2019]) and abbreviated names for variables and methods (Hofmeister et al. [2017]).

Better support tools  
could save a lot of time.

Studies of how time is spent on code comprehension and navigation came to similar results (see Adeli et al. [2020]; Minelli et al. [2015]; Ko et al. [2005]; Xia et al. [2018]). Xia et al. [2018] ran the most comprehensive study. They measured how developers spend their time. Unlike previous studies, their measurements included time spent outside the IDE, like looking up information. Despite all of the support offered by modern IDEs, they found that about half of it is spent on comprehension. Another quarter is consumed by navigation. Only about 5% of a developer's time is devoted to editing code. Thus the time needed for comprehension and navigation appears to be a significant bottleneck for the code writing process.

Current approaches

Promising approaches have emerged to reduce these times. Each addresses some aspect causing issues with code comprehension and navigation. IDEs like *Code Bubbles* (Bragdon et al. [2010]) support memory by integrating spatial code arrangements. Approaches like *MI* by Lee et al. [2015] extend IDEs with file edit recommendations. Others such as *Stacksplorer* by Karrer et al. [2011] implement new views to convey structural information. Lastly, there are code summarizations. *TASSAL* by Fowkes et al. [2016] removes code blocks unnecessary for overall understanding. Recently neural networks for generating natural language code summaries have yielded promising results. Although they cannot keep up with human-written ones yet (see Stapleton et al. [2020]).

## 1.2 Goal and Approach

While researching related work, we noticed a potential avenue to support code comprehension and navigation. One that has so far been neglected by researchers. Almost all current approaches focus on enabling developers to explore the project's overall structure (e.g. *Code Bubbles* by Bragdon et al. [2010]). They provide little to no benefit within the scopes of the code.

There is a lack of dynamic approaches on a scope-wide basis.

### CODE SCOPES:

Code scopes determine which variables, functions, and objects are accessible within a code section. In this thesis, we mean something slightly different, when referring to scopes. We only consider the global scope and a scope for each method. Talking about the local scope of a statement will always refer to the entire scope of the surrounding method (or if outside any method, the global scope). Thus for the sake of this thesis, control structures do not have their own local scope.

Excursus:  
*Code Scopes*

Approaches, that do provide details about individual scopes tend to be static. Most of them are code summarization techniques, providing short descriptions of **what** the code within a scope does (e.g. Iyer et al. [2016]). There is a lack of tools enabling developers to explore **how** the code within a scope works. Dynamically supporting developers during the exploration of a scope's contents might prove beneficial.

The goal of this thesis is to implement a prototype IDE extension. It should provide the above functionality. Its purpose is to support developers' code comprehension and navigation on a scope-wide basis. We want to support the developer's mental model of the code by enhancing their exploration process. The extension will display the local context of the currently selected line. The local context consists of information regarding what other statements the current line is dependent upon within the local scope. The developer can switch the current line at any point to explore a different part of the local scope (or to explore a different

Description of our approach.

Our evaluation strategy

scope). After implementing the prototype, we perform a technical evaluation. We test the prototype on an open-source GitHub repository. This will provide an indication of the prototype's performance and limitations. Finally, we conduct a user study to answer open questions. These arose during implementation. They are in regard to the overall question: "What information constitutes a useful local context?". The study will provide insights into which local dependencies are perceived as relevant by developers. Its results are essential for determining the future direction of our approach.

Thesis outline.

Chapter 2 "Related Work" will outline other approaches for improving code comprehension and navigation in more detail. In chapter 3 "Concept and Design Choices" the extension's concept and related design choices, made during development, will be discussed. Then chapter 4 "Implementation" will detail how the prototype is implemented. Afterwards, chapter 5 "Evaluation" will present the technical evaluation and the user study. Based on their results, chapter 6 "Conclusion" will first discuss the prototype's contributions and limitations. After that, we summarize this thesis, as well as outline potential directions for future work.



## Chapter 2

### Related Work

A large variety of approaches have been proposed to improve code comprehension and navigation. Many of them have shown promising results in studies. They can be grouped into two broad categories. Firstly there are code summarization techniques. Code summaries facilitate understanding of a code piece's functionality. They are a form of code documentation. But even though code documentation is effective (Tenny [1988]), its manual creation is often neglected (Fluri et al. [2007]). Thus the goal of these techniques is to generate code documentation automatically instead. Secondly, there are approaches aiming to support developers by introducing new IDE functionality. They either fundamentally change how developers interact with code in IDEs or they implement new support features. Their goal is to reduce the mental load required to work on the code. Some of these approaches also summarize code. However, they differ from those in the first category because they summarize the code's structure instead of its functionality.

There are two main approaches to supporting code comprehension and navigation.

## 2.1 Code Summarization

*“There is a need for tools that can automatically extract useful documentation, beyond simple UML diagrams or Javadocs, from source code, to substantially reduce program comprehension effort.”*

—Xia et al. [2018]

Code can be summarized by cutting out less relevant parts.

Manual folding is not a good option.

How granular should parts be removed?

Code summarization techniques automatically generate concise ways to understand a code fragment. This can take on different forms. One way to summarize code is by cutting out pieces. To do so an algorithm determines which parts best describe the code’s functionality. The rest is then removed. *TASSAL* by Fowkes et al. [2016] employs a topic model to make this determination. According to McBurney et al. [2014], a topic model of code associates code terms with each other based on their co-occurrence, thus forming groups called topics. These are assigned probabilities of being associated with each code section. A downside of relying on such a model is, that it needs to be trained on the project beforehand. *TASSAL* includes a vector space model requiring no pre-training as a fallback, but it does not perform as well. Regardless of the model used, *TASSAL* “removes” code pieces by folding them away. While folding code can already be done manually, it poses a dilemma. First, one must understand the source code to make an informed decision on which parts to fold, thus rendering the resulting summary redundant in aiding comprehension. The authors hope to make a reasonably informed decision automatically. In an accompanying study, developers preferred *TASSAL*’s summaries over existing folding approaches. According to the authors, they intentionally chose to fold code blocks instead of individual statements. They argue it is more intuitive since code blocks form natural units. Yuan et al. [2017] disagree and chose to implement a similar summarization technique. It removes each statement deemed unimportant for understanding the overall functionality, not just entire blocks. They argue it allows for more precise removal of nonessential statements. Their algorithm uses supervised machine learning, instead

of a topic model, to determine which parts to keep. It is trained on code abridgments written by humans. A user study indicates their method decreases code comprehension times while increasing accuracy. The percentage of code kept can be varied in both approaches. It is yet unclear what compression factor leads to the best summary. Especially since the optimal factor might depend on the situation. Both studies simply used a compression factor of 50%. Outside a study setting, developers might struggle to choose an appropriate factor since the tools are intended to be used on unfamiliar code.

How much should be cut out?

McBurney et al. [2014] also employ a topic model to summarize code functionality. But they use it to construct a structured view of it. Each topic found is assumed to correspond to a functionality. The topics and their associated keywords are put into a hierarchy and then displayed. The authors give the example that in a program the topics “decode mp3” and “open files” might have the topic “play sound” as a parent.

Code functionality is hierarchical and can be displayed as a tree.

Neural networks are another way to summarize code. This approach has become increasingly popular as neural network methods have improved. They generate natural language code summaries comparable to code comments. The first attempts only used source code as input to the neural network (see Iyer et al. [2016]). LeClair et al. [2019] showed, that including structured input like the abstract syntax tree (AST) can improve the resulting summaries, especially for code lacking comments and descriptive names.

Neural networks can summarize in natural language. They are an increasingly popular approach.

**ABSTRACT SYNTAX TREE:**

An abstract syntax tree (AST) represents the structure of code as a tree. Each tree node corresponds to a construct in the code. ASTs are abstract because they do not include “syntactic sugar”, like parentheses or semicolons. These are implicitly represented by the tree’s structure.

Excursus:  
*Abstract Syntax Tree*

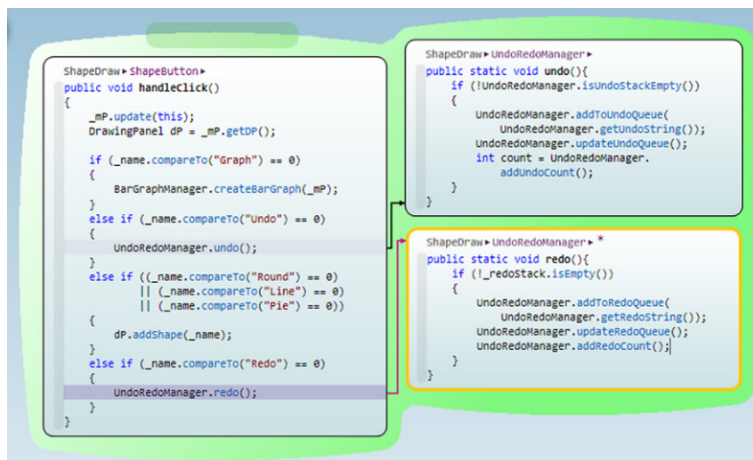
Furthermore Tang et al. [2022] demonstrated that how the AST is encoded plays a significant role. Their encoding improved output quality and reduced computation times. Stapleton et al. [2020] studied how machine-

generated summaries compare to human-written ones regarding code comprehension. While developers perceived their quality as similar, human-written summaries still led to significantly better comprehension. Additionally, they found that frequently used metrics for evaluating these approaches were not suitable for that purpose. Thus despite their promise, neural network approaches are still lacking for the moment.

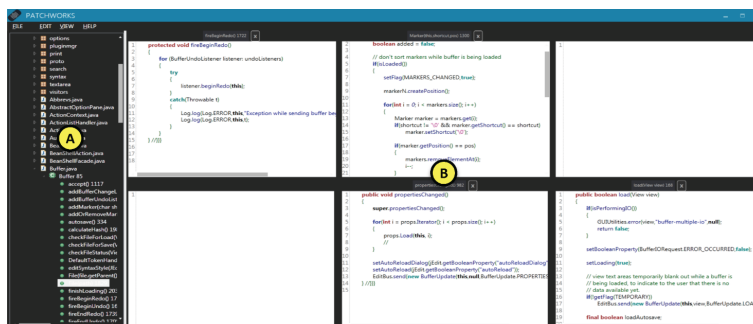
## 2.2 Extending IDEs

Spatial arrangements in IDEs are a promising approach.

*Code Bubbles* by Bragdon et al. [2010] introduces an IDE user interface based around spatial arrangements (see Fig. 2.1). Code fragments, like files, classes, and methods, can be freely positioned as “bubbles” on a two-dimensional plane. Additionally one can connect them with arrows. Thus the developer can arrange code fragments according to their mental model of the project. The authors aim to free up memory otherwise spent on remembering the location of currently relevant fragments within the file system. Their study found that *Code Bubbles* reduces both code comprehension and navigation times compared to a traditional IDE user interface. Adeli et al. [2020] also demonstrated spatial arrangements can help provide developers with the “right information at the right time and place”. *Code Canvases* by DeLine and Rowan [2010] is a very similar approach. It also provides a two-dimensional plane in which code fragments can be arranged. Unlike *Code Bubbles* the plane contains other project documents as well, like images or design documents. Depending on the zoom, the level of detail displayed is adjusted. So when zoomed out, a code file might only display important identifier names, while when zoomed in, the entire file is visible and can be edited. According to the authors, *Code Canvases* displays the entire project, while *Code Bubbles* focuses on currently relevant code fragments. *Patchworks* by Henley and Fleming [2014] restricts spatial arrangements to a grid (see Fig. 2.2). The user can scroll left and right along the grid and fill each grid cell with a code fragment. The grid extends indefinitely left and right, but not up and down. The author conducted a study to compare *Patchworks* to *Code Bubbles*. They



**Figure 2.1:** Code fragments arranged in "bubbles" within the *Code Bubbles* IDE by Bragdon et al. [2010]



**Figure 2.2:** Code fragments arranged in the grid of the *Patchworks* IDE by Henley and Fleming [2014]

found their restrictions discourage users from wasting time arranging the code. By reducing the number of possible navigation directions, *Patchworks* also reduces navigational mistakes. However, they did not investigate whether *Patchworks* retains the benefits of *Code Bubbles* regarding code comprehension.

Recommendation Systems for Software Engineering (RSSEs) support developers' code comprehension and navigation by recommending information related to their current task. According to Robillard et al. [2010], the aim is to "help people find information and make decisions

RSSEs help find information relevant to the current task.

where they lack experience or can't consider all the data at hand". These systems can use different context clues to produce their recommendations. *Mylar* by Kersten and Murphy [2005] computes an interest value for code elements to reflect their current relevance. It is based on how recently and how often the developer interacted with a code fragment, as well as structural information about the code. Within various IDE views, elements like code files or methods are then highlighted. Depending on the computed interest value the highlight strength is varied. The higher the value, the stronger the highlighting, thus making high-interest elements stand out more (see Fig. 2.3). Kersten and Murphy [2006] showed in a study, that *Mylar* reduces information overload. By presenting filtered information it increases developers' focus. Other RSSEs include *MI* by Lee et al. [2015], which is an extension of *ROSE* by Zimmermann et al. [2004]. *Mylar* provides a general overview of related files, while *ROSE* and *MI* directly recommend files to edit. *MI* considers not only the project's edit history, as opposed to *ROSE* and *Mylar*. *MI* additionally takes the view history into account as well. The result is an improvement in recommendation accuracy compared to *ROSE*. In his master's thesis Müller [2023] built an RSSE framework, which combines interaction history (like *MI*) with call graph recommendations. An accompanying study yielded promising results.

Excursus:  
*Call Graph*

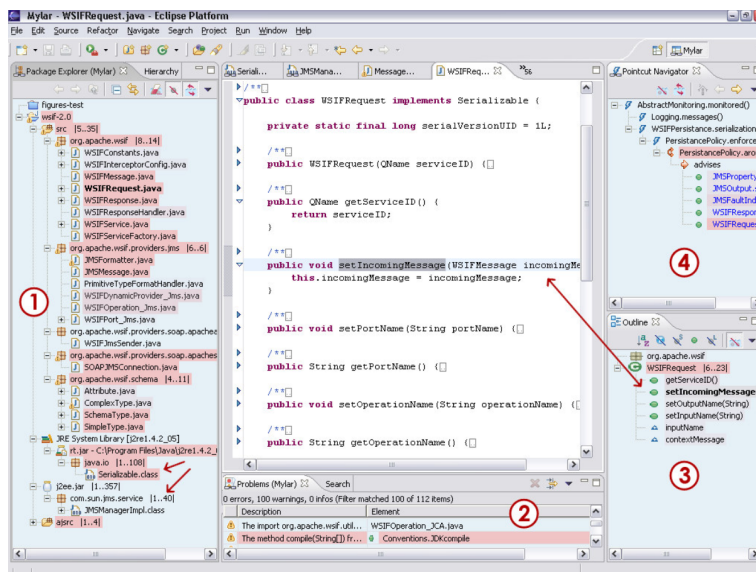
#### CALL GRAPH:

The call graph represents relationships between code methods. Each method is a graph node. Assume method a calls method b. Then there exists a directed edge from a to b.

New IDE views can convey the code's structure.

Apart from RSSEs there exist approaches improving how the code's structure is communicated to the developer. *Stacksplorer* by Karrer et al. [2011] enables an interactive traversal of the call graph. It does so by offering additional navigation options within the IDE *XCode*<sup>1</sup> (see Fig. 2.4). Using these options, developers can easily walk back and forth along the call graph. Both the methods called and

<sup>1</sup> <https://developer.apple.com/xcode/>



**Figure 2.3:** Mylar by Kersten and Murphy [2005] marks various files with a red highlight. The highlight strength depends upon their interest value.

where the current method is being called can be explored. They conducted a study where participants had to perform code maintenance tasks. For most tasks, completion times were significantly reduced compared to the same IDE without the tool. However, they were unable to show that the rate of success improved. Kristensson and Lam [2015] also add a view into an IDE, this time for Eclipse<sup>2</sup>. They use it to visualize the code structure within a file. To achieve this, each code construct, such as classes, functions, or fields, is associated with a different color (see Fig. 2.5). The tool is non-intrusive due to the separate view, but as a consequence, developers have to map the displayed structure onto the code themselves. This is especially the case because according to the authors, their new view does not normally have the same size as the code view (in contrast to Fig. 2.5).

<sup>2</sup> <https://eclipseide.org/>

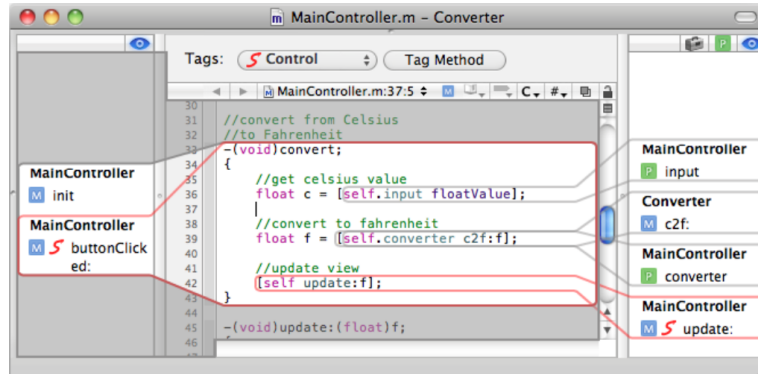


Figure 2.4: *Stacksplorer* by Karrer et al. [2011] shows in- and outgoing dependencies of the method in the call graph.

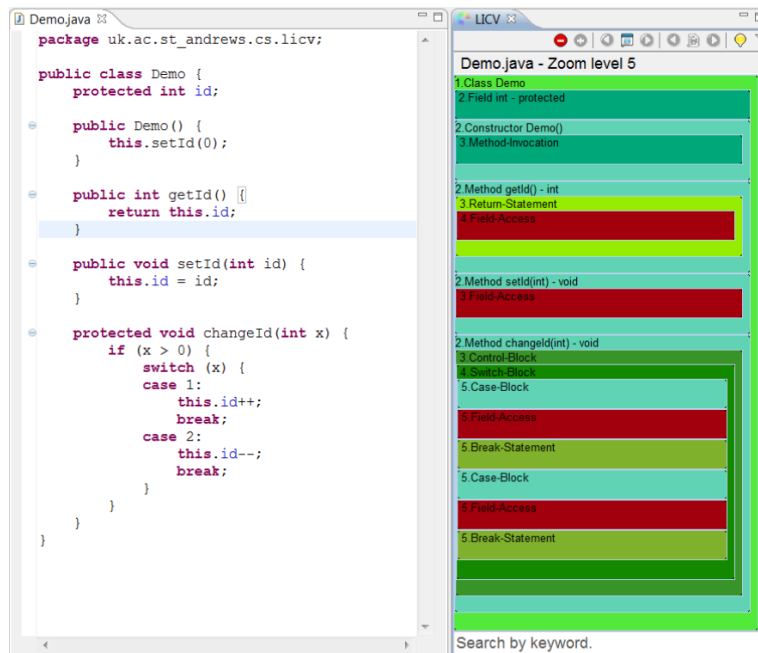


Figure 2.5: Kristensson and Lam [2015] add a new IDE view. It shows a colored outline of the code's structure



## Chapter 3

# Concept and Design Choices

As outlined in 1.2 “Goal and Approach”, we identified a gap within the existing approaches that we aim to fill. Despite their variety, almost all approaches focus on improving comprehension on a file-wide (e.g. Fowkes et al. [2016]) and often even project-wide basis (e.g. Kar-rer et al. [2011]). Dynamic approaches, such as *Code Bubbles* (Bragdon et al. [2010]) enable developers to explore the overall structure of a project. However, once developers explore within a scope of the code, the code is simply presented like in a modern IDE. While IDEs normally include syntax highlighting, Hannebauer et al. [2018] suggest the feature is not all that helpful. Some static approaches provide support concerning individual scopes. Most notably code summarizations. They are short descriptions of **what** a code fragment does. They are often applied to single scopes and not entire code files (e.g. LeClair et al. [2019]). Kristensson and Lam [2015] enrich the IDE’s visuals. They aim to make it easier to parse the structure of a scope. However, all of these do not truly allow developers to dynamically explore **how** the code within a scope works. We believe supporting developers in such a manner could supplement other approaches. Approaches tackling code comprehension and navigation on higher abstraction levels could be complimented by our approach.

There is a lack of dynamic approaches on a scope-wide basis.

|                                 |   |
|---------------------------------|---|
| Chapter outline                 | <p>In the following, we will outline the concept for a tool intended to enable such exploration. This chapter will detail what we aim to achieve, as well as how we aim to do so. Along with this, we will discuss important design choices. This includes some limitations that we decided upon. The concept behind the tool can be summed up as follows:</p>  |
| Summarized concept              | <p>The goal of this thesis is to implement a tool supporting code comprehension and navigation within the local scope, by providing local context regarding the current line in the IDE. The local scope refers to the scope within which the current line resides. The local context of the current line consists of information about variables of the local scope. More specifically, which of them the current line depends upon and in what way. The aim is to help developers trace this dependency. It is transitive and depends both on variables, as well as control structures and their conditions (which tend to be variables as well). It also depends on invoked methods. We call the context local because we will not trace dependencies outside the current scope. Thus our tool will not capture the complete context. The reasons for this choice will be discussed in 3.1 “Deciding on Limitations”.</p>  |
| We chose to use an AST analysis | <p>To determine the local context, our tool will implement an algorithm. This algorithm will analyze the AST. The AST is better suited as an input than the code file. It is much easier to extract structural information about the code from the AST. This is quite convenient as we are interested in code dependencies. It also has the added benefit of sparing the analysis from dealing with syntax. Semantics are similar across many programming languages. Especially basic concepts, such as variables, if-statements or method calls share similarities. By choosing AST analysis, it will decrease the effort required to adapt our algorithm to other programming languages. As part of this thesis, we will only analyze <a href="https://www.python.org/">Python</a><sup>1</sup>. The analysis can simply traverse the tree (except for loops, see 3.1 “Deciding on Limitations”) because we are only dealing with the local scope. Essentially our algorithm will go up line by line in the source code. Not only does this simplify the algorithm, but it also allows us to discard huge parts of the code file very eas-</p> |

<sup>1</sup> <https://www.python.org/>

ily. Lines below the current one are discarded, because the current line cannot be dependent on them. The hierarchical nature of the AST is beneficial for dealing with code nesting. Since scopes are determined by nesting, the local scope is a subtree of the AST. Thus the analysis only needs to consider this subtree. The rest is simply ignored.

We chose to implement the tool as an IDE extension for VSCode. Modern IDEs like VSCode offer extensive application programming interfaces (APIs) to develop them. This makes them a convenient option to test new features in the realm of software development. We chose VSCode in particular because it is the most popular IDE (see Fig. 3.1). As mentioned before, the prototype will only analyze Python. We chose it because it is very popular and widely used (see Fig. 3.2). Additionally, it comes with a [module to extract ASTs](#)<sup>2</sup> from given Python code. There also exists [extensive documentation](#)<sup>3</sup> for the Python AST. Furthermore AST nodes in Python store the text position they correspond to in the code. At the end of the analysis, we want to place markings in the code. This makes it much simpler to place them appropriately. In general, it should be noted that the choice of programming language is not as relevant. Most concepts represented by AST nodes, such as assignments or control structures, exist in many languages.

The tool will be implemented as an IDE extension for Python.

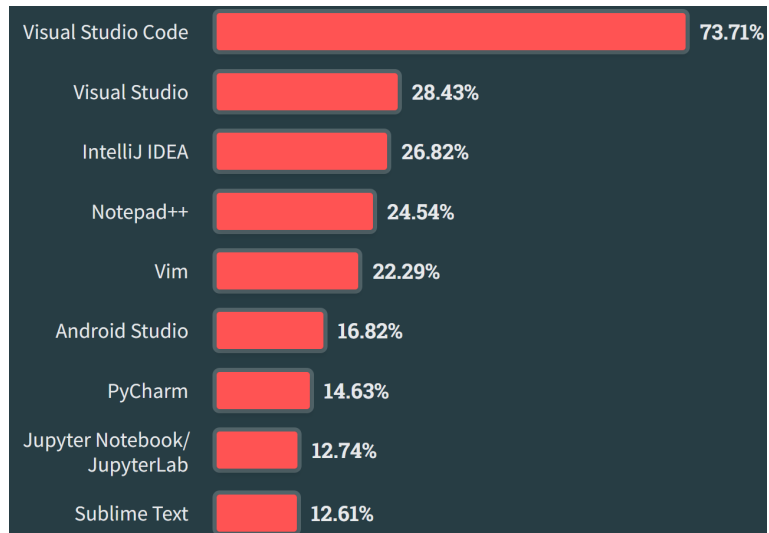
### 3.1 Deciding on Limitations

There are several reasons for limiting the considered context. On the one hand, it is simply not possible to truly capture the entire context with just a static code analysis. Especially in a language like Python, which allows adding new code during runtime. On the other hand, we impose some limits that we believe lead to a minimal prototype still sufficiently capable for testing our approach. The following section will detail three areas, where we decide to limit our prototype, as well as why we do so.

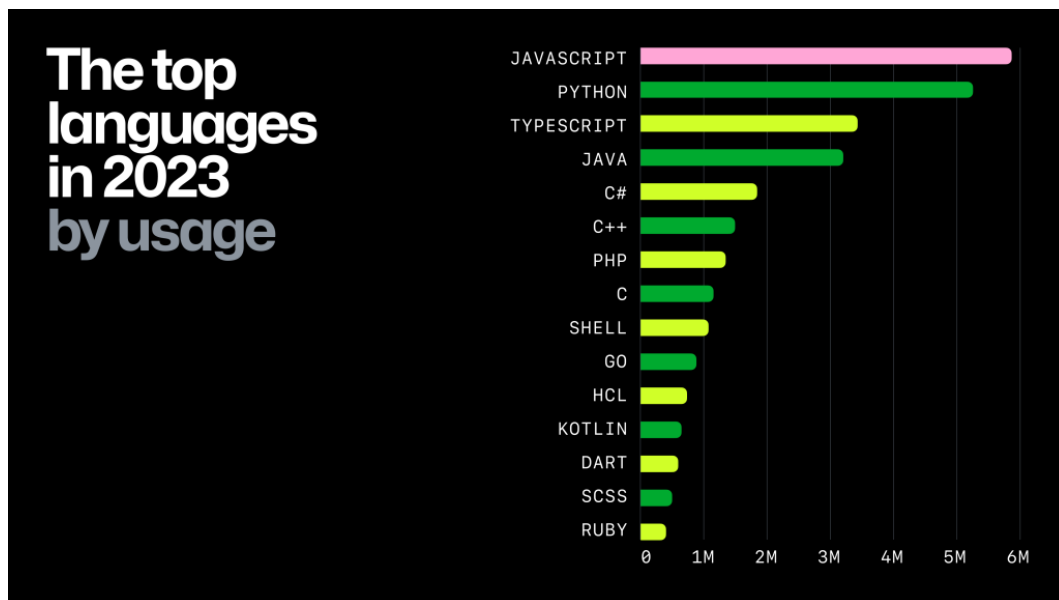
Why we limit our approach.

<sup>2</sup> <https://docs.python.org/3/library/ast.html>

<sup>3</sup> <https://greentreesnakes.readthedocs.io/en/latest/nodes.html>



**Figure 3.1:** IDE usage in 2023 according to Stackoverflow (<https://survey.stackoverflow.co/2023/>)



**Figure 3.2:** Programming language usage in 2023 according to GitHub (<https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>)

As mentioned before, we limit our prototype to an analysis within the local scope. It would be possible to trace dependencies across scope boundaries, but that entails huge issues. When going out of scope the analysis is likely to encounter situations, where a piece of code is part of the context several times on different paths of the dependency trace. This is especially the case for recursion. This raises several questions, such as: How often do you trace the same scope? How do you convey the information that there are several dependencies at once? Apart from that the analysis gets more complicated as well. Names of variables and methods can no longer be assumed to be unique. Not only that but one has to keep accurate track of when two variables of the same name are the same and when they are not. By staying within the local scope we can benefit from some assumptions during the analysis. For one, lines below the current one cannot influence it and thus do not have to be considered. There is one further reason for staying within the local scope. We believe tracing across scope boundaries is already covered by the call graph. If the developer sees that the current line is dependent upon the call of a method, they can use existing call graph tools to investigate that method. Once within the new method, they can once again use our tool. Although we do analyze while- and for-loops, we do not take their looping behavior into account. That is a further limitation we decided upon. It is unclear how many iterations should be traced. A static analysis is most often unable to determine how many iterations there will be. With both these limitations in place, the analysis only has to look at each tree node once. Of course such limitations raise the question, of whether the resulting context is still insightful enough. Can users still grasp the context surrounding a line with these limitations in place? This, along with several other questions, will be investigated in 5.2 “User study”. The results will indicate whether the limitations are reasonable.

Why we stay in the local scope

Only analyze loops once.

Can we set such limitations?

Unfortunately it is not trivial to decide, based upon the AST, which node best describes the start line. It is rarely a leaf node. To simplify our prototype, we make some assumptions about the starting line. It can be an assignment, a method call, or a return statement. If it is the head of

Why we limit possible starting lines.

a control structure, we will start the analysis on the first statement inside the structure's body.

Why and what Python subset is considered?

Due to the prototype nature we only consider a subset of Python for the analysis. We do this by only considering certain nodes in the Python AST. For some nodes, we only consider a few of their attributes. Our focus is on including the most commonly used features of the language. Those tend to be shared by many different programming languages. Many features left out deal with redundant syntax, e.g. lambda functions. They can be replaced by more common features. Others do not provide much benefit to our analysis, e.g. imports. We also do not consider the asterisk operator, used for positional and keyword arguments. Anything related to exceptions, asserts, etc., as well as asynchronous functionality, are ignored as well.

The subset used.

Our subset mainly consists of the following Python features:

- Literals (e.g. sets, dictionaries), variables, attributes, and subscripts (e.g. `array[index]`)
- Assignments, expressions (e.g. `a + b`) including method calls and return statements
- if-statements, for-, and while-loops
- Function and class definitions

Many features in this subset appear very similarly in other programming languages. Thus how we deal with them in the analysis should be largely transferrable. The next section will describe how to deal with the most important language features.

## 3.2 Determining Local context

How to trace variable dependencies.

This section will outline how to extract the local context

for our Python subset. Variables are at the core of the algorithm. In the case of attributes, we track the variable representing the underlying object. This is to compensate for the attribute name not being unique. During the analysis variable occurrences are marked with a value. The value reflects how distantly related the current line is to that variable occurrence. The local context consists of all variable occurrences that have been marked with a value different from -1 during the analysis. -1 represents that there is no dependency present. The algorithm traverses the AST such that it corresponds to going up line-by-line in the source code. The starting point of the analysis is the currently selected line within the IDE. If that line is an assignment the analysis only considers the variables on the left side of it. Otherwise, all variables on the line are considered. The analysis traces what this set of found variables depends upon. This set will be referred to as the watched variables.

After initializing the set, the analysis goes up line-by-line, or to be more precise statement-by-statement. When a watched variable is being assigned, we add any variable on the right-hand side to the set. We remove the watched variable from the set, unless it is also on the right-hand side or if we encounter an augmented assignment (e.g. `a += b`). If the right-hand side only consists of constants, then no new variable is added to the set. Overall the specific value and type of constants are ignored by the analysis. Fig. 3.3 contains a basic example of dealing with assignments.

How to deal with assignments.

The example also includes operators. It does not matter which operator is used specifically. Nor does it matter in which order operands are. What matters is which variables are inside the operands. In Fig. 3.3, `a` and `b` are operands of the multiplication in line 5, and `c` is dependent on both of them. So the analysis simply extracts all variables from the right-hand side regardless of any operators present. Similarly, variables are extracted from comparisons. Their ordering and their comparison symbols are not taken into account. For method calls, all variables within the parameters are considered. If a watched variable is dependent on the method's return value, then the found parameter variables are added to the watched variables. In the case of subscripts, e.g. `container[slice_criterion]`, both the `container` variable, as well as variables influenc-

How to deal with expressions.

```

1  a = 5
2  b = -2
3  name = 'Tom'
4  c = name
5  c = a * b
6
7  d = c + 2

```

**Figure 3.3:** Line 7 is the current line. It depends upon line 5, which in turn depends on lines 1 and 2. *d* is not dependent on lines 4 and 3, because *c* is completely overwritten in line 5.

ing the slice are considered. The slice consists of the index values addressed. Once again, if any watched variable depends upon these variables, they are added to the watched variables themselves.

How to deal with control structures.

The general idea is, that the current line is dependent on the control structure condition if the current line is dependent on a line in the structure's body. There is always the possibility that the structure's body is not entered because the condition is unsatisfied. An exception is an if-statement that also includes an else section. The watched variables need to take this into account.

How to deal with if-statements.

In the case of if-statements, the analysis deals with them as follows. A sub-analysis is called on both the if- and else-



section. Both sub-analyses start with the currently watched variables. The watched variables each of them had at the end are merged after the sub-analyses conclude. If there was only an if-section, the watched variables at the beginning and end of the sub-analysis are merged instead. As long as one of them contains a variable, the merged set will contain it as well. If both contain the same variable but with different relatedness values (see beginning), then the closer related value is used for the variable in the merged set. This merged set is now the new set of watched variables. Next, it is checked if the watched variables changed compared to before the sub-analyses. A change means that the current line is dependent on a variable occurrence within the if/else-body. That occurrence is dependent on the condition of the if-statement. Thus if a change is detected, any variables in the condition are added to the watched variables. While- and for-loops are treated as follows. First, a sub-analysis is started in the structure body. It starts with the currently watched variables. The watched variables at the end of that analysis are merged with those watched beforehand. If a change is detected, variables in the condition are added to the watch. Our approach does not take into account, that there might be multiple loop iterations (see 3.1 “Deciding on Limitations”). If the current line is inside a control structure, it is dealt with differently. First, an analysis is done within the structure body containing the current line. Other ones are ignored if they exist. The analysis starts with the current line. After finishing it adds any variable inside the structure condition to the watch. Finally, the analysis continues with the statement above the structure. Fig. 3.4 shows an example of the algorithm dealing with an if-statement.

How to deal with loops.

How to start inside a control structure.

There are two cases of how function and class definitions can occur within the local scope. Either as the definition of a sub-function/class or as the head of the scope. In the first case, they are simply ignored. Such function/class definitions contain a different scope. In the second case, function arguments are marked, if and only if they are within the watched variables when the beginning of the scope is reached (so when the analysis concludes). Class definitions do not have arguments, so no variables are added to the watch.

How to deal with function/class definitions.

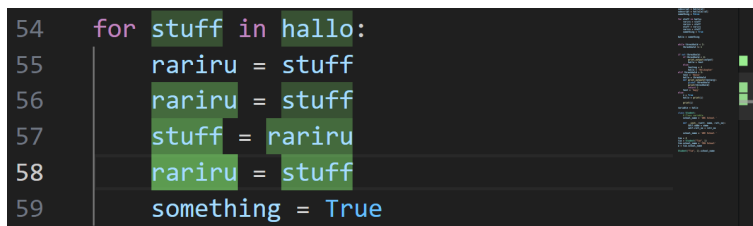
```
1  def is_positive(number):
2      if number > 0:
3          output = True
4      else:
5          output = False
6      return output
```

**Figure 3.4:** Line 5 is the current line. It is dependent on line 2 for its execution, which in turn relies on `number` from line 1. Line 3 is unrelated because it is in a different if-branch. The return statement is unrelated because it is executed afterwards.

### 3.3 Displaying Line context

Background highlights  
are used to convey the  
local context.

We will employ a simple approach regarding how we mark the local context. This thesis is not focused on discerning the optimal visuals. The prototype uses background highlights to convey the local context (see Fig. 3.5). Strobel et al. [2016] found, that they are an effective way to highlight text. While other highlighting techniques performed similarly, we chose background highlights. They can be layered on top of existing IDE markings. Additionally, they can be faded out, depending on how prominently the target should be marked. Generally, we could also convey the information via a new IDE view. This however has the downside, that the developer has to map the information on the code themselves. As mentioned in the previous section, the local context consists of variable occurrences. Thus we only mark variables. Each occurrence is associated with a relatedness value. Based on this value we differ the strength of the marking. Higher relatedness is marked stronger, and thus more prominent. If the value indicates there is no dependency, then we place no marking on that variable occurrence. Developers are likely less interested in longer dependencies, so we fade them out. Our approach



```
54 for stuff in hallo:
55     rariru = stuff
56     rariru = stuff
57     stuff = rariru
58     rariru = stuff
59     something = True
```

**Figure 3.5:** This example shows all three aspects of our markings. (1) Background Highlights, that (2) fade out. (3) The scroll bar on the right contains a marking as well

supports navigation as well. Developers can instantly see which code fragments are relevant to consider. They can directly skip those that are not. We additionally indicate where marked lines are with a green indicator on the scroll bar (see Fig. 3.5). This enables navigating to code fragments off-screen.



## Chapter 4

# Implementation

As previously mentioned, we implement the prototype as a VSCode extension. Extensions for VSCode are written in [TypeScript](https://www.typescriptlang.org/)<sup>1</sup>. Additionally we use of [Node.js](https://nodejs.org/en)<sup>2</sup>. The extension interacts with the IDE via the [VSCode API](https://code.visualstudio.com/api)<sup>3</sup>. This API allows us to set in which situations the extension should be active. For our purposes that is when a Python file is currently open. Whenever the active file gets modified, or a new (Python) file becomes the active file, the method `updateAST()` is invoked. It starts the process of analyzing the code and marking it. Each of the main steps in that process will be discussed in its section of this chapter.

The prototype is a VSCode extension.

What does the extension do?

1. Extract the code and generate an AST from it.
2. Prepare the AST for the analysis.
3. Conduct the AST analysis as described in 3.2 “Determining Local context”.
4. Set highlights according to the analysis results.

---

<sup>1</sup> <https://www.typescriptlang.org/>

<sup>2</sup> <https://nodejs.org/en>

<sup>3</sup> <https://code.visualstudio.com/api>

## 4.1 Generating the AST

The extension communicates with a Python script to generate the AST.

To generate the AST, we first need the current code file. The API provides an easy option to extract the active file's contents. To generate the AST we call a separate Python script using Node.js. The script is fed the extracted code file. It generates the AST with Python's AST module. The resulting object is parsed into a JavaScript Object Notation (JSON) string using the [ast2json module](#)<sup>4</sup>. Now the string is transferred back to the extension. Interfaces are used to convert the JSON string into a TypeScript Object. Thanks to the interfaces, we can interact with the AST and its various node types. `annotateAST()` is called with the AST object as a parameter. This initiates further processing and analysis of the AST.

## 4.2 Preparing the AST

Determine anchor node and initialize watched variables.

Cutting out the scope's subtree.

First, we add a parent node attribute to each node. Doing so makes it easier to work with the tree. The Python AST does not include them by default. Then `analyseAST()` is called. The method first determines the anchor node. The anchor node is how we will refer to the node that was found to correspond best to the current line. It is determined according to the limitations outlined in 3.1 "Deciding on Limitations". Next, we initialize the watched variables. These are all variables in the subtree, that has the anchor node as its root. Except when dealing with an assignment. Then only the variables on the left-hand side are watched variables. The analysis only needs to consider the AST subtree corresponding to the current scope. So the AST is trimmed. The subtree of the current scope is cut out. Any nodes for lines below the current one are removed as well. If the current line is inside an if-statement, the branch not containing the current line is cut out. Sub-function and -class definitions in the scope are also removed. Lastly, if the current scope is within a function, then the function definition node

<sup>4</sup> <https://pypi.org/project/ast2json/>

is moved, such that the analysis will consider it last. Thus the AST is now ready for the analysis.

### 4.3 AST Analysis

Generally the analysis is conducted according to 3.2 “Determining Local context”. The thing to note in the implementation are the variables `passDown` and `passUp`. These are used to pass values up and down in the tree. Such a value serves two purposes. First, it signals to parent (and indirectly sibling) nodes, that something has happened. In the case of an assignment, this means a watched variable was found on the left-hand side. Thus variables on the right-hand side need to be added to the watch. For control structures, it signals, that their contents are part of the context. Thus variables in their condition must be added as well. Second, the specific value passed around determines, which relatedness value newly added variables get. The relatedness value is called `relevancy` in the implementation. A higher value means closer relatedness. The value decreases by one for each new dependency. But it can never go lower than zero. Additionally it is important to know, that the `ctxType` of nodes is used. The `ctxType` describes whether a variable, method, etc. is being stored or loaded. We use this value to differentiate different situations. For example, variables on the left-hand side of an assignment are being stored, while those on the right-hand side are loaded.

Signals are passed along the tree.

`ctxTypes`

### 4.4 Highlighting

After the analysis concludes, the tree is traversed. Each node’s `relevancy` is checked. If it is something different from `-1`, we place a decoration. Code decorations can be placed via the VSCode API. The decoration consists of a green background highlight and sidebar marking. This corresponds to the description in 3.3 “Displaying Line context”. The decoration is placed on the code associated with

Place VSCode decorations based on `relevancy`.

the node. The strength of the highlight is based on the relevancy value. Higher values result in stronger highlights because higher values mean closer relatedness.



## Chapter 5

# Evaluation

In this section we evaluate the merits of our prototype. We performed a technical evaluation on a real-world project. It demonstrated how well the current prototype version lives up to the concept outlined in chapter 3 “Concept and Design Choices”. We identify implementation bugs, as well as situations overlooked during the design phase. We also conducted a user study to investigate the open questions, that arose in 3.1 “Deciding on Limitations”. The study results indicate whether or not some fundamental design decisions made, need to be reconsidered. Participants’ comments provide an insight into how they felt about the concept. They also gave some interesting suggestions on how to improve it.

We performed a technical evaluation and a user study.

### 5.1 Technical Evaluation

We tested the prototype on an open-source GitHub project called [Unknown Horizons](https://github.com/unknown-horizons/unknown-horizons)<sup>1</sup> (see Fig. 5.1). It is a 2D real-time strategy simulation video game. We found it through Müller [2023]’s master’s thesis. To best test out the algorithm determining the context, long dependencies emanating from the current line are preferable. Unknown Hori-

Evaluation took place on an open-source project.

---

<sup>1</sup> <https://github.com/unknown-horizons/unknown-horizons>



**Figure 5.1:** An in-game screenshot of the open-source game Unknown Horizons.

zons is a real-world project containing large files. Thus it is suitable for testing the prototype.

Our supported Python subset limited the evaluation.

Large files could not be processed.

The concept is better suited for large scopes.

In general the extension worked as intended. Especially when dealing with basic structures. Code dependencies were successfully tracked over long distances. However, the evaluation was hindered by the limited Python subset our prototype supports. Many code sections contained language features intentionally ignored by the prototype. Naturally, the prototype failed in these sections. Unfortunately, any large-scale project is likely to contain more than basic Python features. So choosing a different project would not have helped. In the end, evaluation was limited to simpler code sections in the project. Furthermore we encountered another significant problem, that further limited our evaluation. On large code files, the current version of the prototype fails to generate an AST. The Python file generating the AST writes it to the standard output buffer. Once it is completely full the Python file cannot write to it anymore. But the extension only empties the buffer once the complete AST is written to it. The result is a deadlock between the Python file and our TypeScript extension. This ruled out the longest files in the project entirely. In the end, the prototype could only be tested on code files of a few hundred lines.. During the testing it became clear, that the concept is rather unsuited for very compact and efficient code.

Such code fragments tend to be very interconnected. As a result, almost every variable occurrence is being marked. Such dense markings are unlikely to provide the user with any insightful information. They might as well read the entire code fragment.

Apart from these more general problems regarding the prototype's applicability to real-world code, we also noticed a notable oversight in our concept. From here onwards all code examples in the subsection stem from `terraincache.py`, an Unknown Horizon's file. It will be used to exemplify our most interesting findings. The oversight mentioned is about how methods called on objects are dealt with. Our concept did not consider object methods different from normal methods. Normal methods are not traced, because they inevitably leave the local scope (recursive calls are their own scope). But if a method is called on an object, it is reasonable to assume that internal values influence the return value. Thus if a variable is dependent on the return value, it is also likely dependent on the object. So it would be sensible to add the object to the context as well. Fig. 5.2 portrays an example of an object method call. Line 214 is the current line. In line 213 the second occurrence of the variable `result` has the method `intersection` called on it. Currently, only the parameters are added to the context. `result` leaves the context because it is completely overwritten. It would make more sense for it to stay since it is dependent on itself via the method. This example also illustrates some of the prototype's limitations in practice. It marks the object of attributes, but not the attributes themselves, e.g. `cache_layer.cache` in line 213. It also does not consider the asterisk operator. That is why `other_cache_layer` in line 210 is not marked, even though it is in line 212.

Our concept did not consider object methods.

Lastly the evaluation also unearthed some bugs within the implementation. In the following, three noteworthy bugs will be presented in detail. Most bugs are related to edge cases. They defy certain assumptions made during implementation. Most often they lead to an incorrect start of our algorithm, which thus computes the wrong context or no context at all. These bugs serve to show that implementing

We found bugs as well.

```

210     def get_buildability_intersection(self, terrain_type, size, *other_cache_layers):
211         result = self.cache[terrain_type][size]
212         for cache_layer in other_cache_layers:
213             result = result.intersection(cache_layer.cache[size])
214         return result

```

**Figure 5.2:** Example of an object method call, that is not tracked.

our presented concept is not trivial, even for such a limited subset of Python.

Bug: Starting at the head of a control structure.

The prototype does not behave as intended when the starting line is the head of a control structure. In Fig. 5.3 Line 116 is the starting line, but the algorithm starts its analysis in 119. As outlined in 3.1 “Deciding on Limitations”, our prototype does not support starting with the head of a control structure. In that case, the algorithm should start at the first line of the control structure’s body. This minimizes the deviation from the context, that would arise when starting in the head. However, due to an error in the code, the last line is started with instead. Consequently, the entire control structure is traced and marked. It leads to a significant increase in variables being marked compared to the first line’s context. So the deviation is much greater. There is an additional bug in this example. Since line 119 is the starting line, due to the first bug, `sq3` is marked. It is also marked in line 112. But it should be forgotten after the first algorithm step because it is completely overwritten by the tuple (`has_land`, `has_coast`). A subscript in the starting line is a special case, that is not correctly handled by our implementation.

Bug: Initialization not counted as change within control structure.

In the next example, visible in Fig. 5.4, the conditions of a control structure are incorrectly left unmarked. The cause is a bug, that can happen when the current line is inside a control structure. The algorithm starts in line 63. Since the line is inside both the if-statement and for-loop, both their conditions should be marked. The reason why that does not happen lies in how the algorithm decides to mark control structure conditions. That decision is based on whether or not the set of watched variables changes during the analysis of the structure. If no change is detected, the structure is assumed to have no relation to the context. The algorithm neglects to count the initialization of the starting line vari-

```

112     sq3 = {}
113     for coords in row3:
114         coords2 = (coords[0], coords[1] + 1)
115         coords3 = (coords[0], coords[1] + 2)
116         if coords2 in row3 and coords3 in row3:
117             has_land = row3[coords][0] or row3[coords2][0] or row3[coords3][0]
118             has_coast = row3[coords][1] or row3[coords2][1] or row3[coords3][1]
119             sq3[coords] = (has_land, has_coast)

```

**Figure 5.3:** Example of a bug, where the analysis incorrectly starts, when the current line is the head of a control structure.

```

61         for coords, tile in self._island.ground_map.items():
62             if 'constructible' in tile.classes:
63                 land.add(coords)
64             elif 'coastline' in tile.classes:
65                 coast.add(coords)

```

**Figure 5.4:** Example of a bug where the prototype incorrectly considers the condition irrelevant.

ables as a change because it happens before the start of the main analysis. As a result, if no other change takes place, as is the case in this example, then the structure is deemed as unrelated.

The last example, depicted in Fig. 5.5, is a case, where the variables are not correctly initialized. As a consequence the analysis starts with no variables, so it can also find no new variables on which they depend. The starting line in the example is line 208. Our implementation assumes the left- and right-hand sides of an assignment can be distinguished based on their `ctxType` (see 4 “Implementation”). It assumes variables and attributes on the left side will be marked as storing, while those on the right side as loading. It turns out, that is not always the case. Here the left side consists of a subscript of an attribute. In this special combination, the attribute is marked as loading, because the subscript surrounding it is marked as storing. As a result the algorithm incorrectly concludes there are no attributes to mark on the left side. The starting variables end up as the empty set and the empty set is dependent on no variables.

Bug: Special cases  
lead to an unexpected  
`ctxType`.

```

200 ✓   for bx, by in coast_set:
201 ✓       for dx, dy in nearby_coords_list:
202           coords = (bx + dx, by + dy)
203 ✓       if coords in water_bodies and water_bodies[coords] == sea_number:
204           near_sea.add((bx, by))
205           break
206
207       self.cache[TerrainRequirement.LAND_AND_COAST_NEAR_SEA] = {}
208       self.cache[TerrainRequirement.LAND_AND_COAST_NEAR_SEA][(3, 3)] = near_sea

```

**Figure 5.5:** Example of a bug where variables in the starting line are incorrectly initialized due to a special case.

## 5.2 User study

### Goal of the study

Originally the intention for our user study was to investigate **how** to best mark the context. After all, there are countless possible ways to do so. However, several interesting questions arose during the design and implementation process. Thus we decided to pivot the user study to investigate them instead. The overall question we aim to answer with the user study is: "What information constitutes a useful context?". This is not as clear-cut as determining dependencies and conveying them to the developer. On the one hand, just because some dependency exists, does not mean it would be helpful for the developer to know about it. On the other hand, some code markings might provide insight into the code's purpose, even though the current line is not dependent on the marked code. Whether or not to include these additional markings could be left up to the user. However, we believe it best to try to find a reasonable standard. Because, when developers explore the code, they are unlikely to know which options would be most beneficial for them to explore that specific code fragment. The study investigated the following six open questions.

### Show overwritten initial assignments?

- **Initial Assignment (IA):** *Should overwritten initial assignments be marked?* Sometimes the initial assignment of a variable is overwritten before the current line. Thus there is no dependency. However, developers might still garner important clues about the variable from its initialization. Fig. 5.6 contains an example.

```

1 def print_linewise(text):
2     line = ''
3     for letter in text:
4         if letter == '\n':
5             print(line)
6             line = ''
7         else:
8             line += letter

```

**Figure 5.6:** Start line is 6. Variable `line` is completely overwritten in the starting line. Left is without and right with a marking for the overwritten IA.

- **Side Effects (SE):** *Should side effects be highlighted?* While it seems likely that developers want to know about any side effects, there are potential downsides to marking them. Most of all they might be confusing. Oftentimes they are caused in a different scope, so it is not immediately clear why they are marked. The markings might also be misleading to a certain degree since a static analysis will not be able to find every side effect. Thus marking a limited subset could lure developers into a false sense of security. Another downside is that even if the analysis does not mark beyond the local scope, it would still have to analyze beyond it. Fig. 5.7 contains an example.

Show side effects?

- **Class attributes (CA):** *Should attributes be traced as part of the entire object instead of individually?* When an attribute is part of the context, it is unclear whether only this specific attribute of the associated object, or the entire object is relevant for the context. When an object method is called it might influence attributes. This is missed if only the attribute is taken into consideration. However, as long as the analysis does not trace outside the local scope, it is impossible to know whether that is truly the case. Additionally tracking the entire object might provide valuable insights into the object's purpose, even if the other object attributes do not affect the current line's value. To avoid overlap with the side effects question, we avoided object methods in study tasks investigating CA. So the study only investigated whether developers were interested

Trace attribute or entire object?

```

1  ✓ def reverse_word(word):
2      global reverse
3  ✓  for character in word:
4      |     reverse = character + reverse
5
6  word = input('Please enter a word:').lower()
7  reverse = ''
8  reverse_word(word)
9  print('The reverse is ' + reverse)
1 ✓ def reverse_word(word):
2     global reverse
3     for character in word:
4     |     reverse = character + reverse
5
6     word = input('Please enter a word:').lower()
7     reverse = ''
8     reverse_word(word)
9     print('The reverse is ' + reverse)

```

**Figure 5.7:** Start line is 9. Above is without, below is with side effects marked. The method `reverse_word` has a side effect affecting the variable `reverse`.

in other object attributes being marked. Fig. 5.8 contains an example.

Trace loops more than  
once?

- **Iterations (It):** *Should loops be traced more than once?* In a static analysis, it is often unclear how often a loop will be executed. But it is likely to be more than once. Otherwise, the developer would not have used a loop. Currently, our prototype only traces one iteration, as if we were dealing with an if-statement. The difference in the resulting markings is especially noticeable when the current line is towards the beginning of a loop. If we trace the dependencies for more than one loop iteration, we have to consider lines below the current line as well. Such markings might be confusing to developers since they are inconsistent with all other cases. Otherwise, only lines above the current line are marked. But even if such markings are confusing they might still be insightful. Fig. 5.9 contains an example.



```

1  class Student:
2      def __init__(self, name, school_type):
3          self.name = name
4          self.school_type = school_type
5
6  student = Student('Max', 'High School')
7  print(student.name + ' has graduated.')
8  student.school_type = 'University'
9
10 print('Now they attend ' + student.school_type)

```

---

```

1  class Student:
2      def __init__(self, name, school_type):
3          self.name = name
4          self.school_type = school_type
5
6  student = Student('Max', 'High School')
7  print(student.name + ' has graduated.')
8  student.school_type = 'University'
9
10 print('Now they attend ' + student.school_type)

```

**Figure 5.8:** Start line is 10. Above only the attribute’s dependencies are traced. Below the entire object is traced.

|  |  |
|--|--|
| <pre> 7  while start != pi: 8      steps_needed += 1 9      if start + step &gt; pi: 10         step /= 2 11     else: 12         start += step </pre> | <pre> 7  while start != pi: 8      steps_needed += 1 9      if start + step &gt; pi: 10         step /= 2 11     else: 12         start += step </pre> |
|--|--|

**Figure 5.9:** Start line is 12. The left side is tracing the loop without taking iterations into account. The right side traces with iterations.

- **Scope Down / Up (SD / SU):** *Should the context be traced downwards / upwards in scope?* The main limitation imposed upon the prototype is that it only analyses the local scope. The reasons for that were outlined in 3.1 “Deciding on Limitations”. This is a severe restriction and should thus not be made arbitrarily. Ideally, the study will validate this limitation as a sensible restriction, that still enables developers to compre-

Trace out of scope  
downwards / upwards?

hend the local context. We split up the question about going out of scope. Going down in scope seems more likely to provide useful context than going up. After all, when exploring a method, understanding called methods helps to piece together what the method itself does. While where a method is being called does not. It just gives a hint about the method's usage. Fig. 5.10 contains an example, both for going up in scope and for going down.

### 5.2.1 Study Design

The study was conducted online and made with [SoSci Survey](#)<sup>2</sup>. Pictures of the study and all of the task descriptions can be found in A "User Study Documents". Participants underwent the following steps when taking part in the survey:

- |                     |   |
|---------------------|---|
| Welcome and consent | 1. Upon starting the survey, participants first received a welcome message. They also had to consent to the conditions of the study.  |
| Demographics        | 2. After participants agreed to participate, they had to fill out some demographic information.   |
| Task explanation    | 3. Next participants were explained the tasks. Throughout the study, they repeatedly ranked three pictures. In each task, a short context description, related to the content of the pictures was given as well. Within a task, all three pictures contained the same piece of code. The code was marked in different ways. There was a red line marking, which indicated the current line. It was the same across all three pictures. There also were green line markings, that participants were told should support them in understanding the local context of the current line. The markings were done with <i>Easy Highlight</i> <sup>3</sup> (version 1.2.0), a VSCode extension. Participants were asked to rank how well the different markings helped with understanding |

<sup>2</sup> <https://www.soscisurvey.de/>

<sup>3</sup> <https://github.com/BrandonBlaschke/vscode-easy-highlight/>

```

1 def linear(x, coefficient, constant):
2     result = x * coefficient + constant
3     return result
4
5 a = 5
6 b = -2
7 print(linear(17, a, b))
1 def linear(x, coefficient, constant):
2     result = x * coefficient + constant
3     return result
4
5 a = 5
6 b = -2
7 print(linear(17, a, b))
1 def linear(x, coefficient, constant):
2     result = x * coefficient + constant
3     return result
4
5 a = 5
6 b = -2
7 print(linear(17, a, b))

```

**Figure 5.10:** Start line is 3 for top left and 7 for bottom left. The left side shows markings without going out of scope. These two cases produce the right side markings: Top left, but with going up in scope. Bottom left, but with going down in scope.

the local context of the current line. The three pictures always corresponded to three conditions. "Minimal Visuals". "Additional Visuals" and a baseline. Their order was random and participants were not told about them. "Minimal Visuals" corresponded to what should be marked according to what was laid out in 3 "Concept and Design Choices". "Additional Visuals" included the same markings, but also additional markings. These were based on what should be marked if one of the previously mentioned open questions was answered with yes. The baseline was based on an IDE feature, that marks all occurrences of a variable. If there was more than one variable in the current line, only the occurrences of one of them were marked. To minimize the effect of how we marked the code, we decided to always either fully mark a line or not at all. Fig. 5.11 shows an example of the different markings in one of the tasks. Our goal for the study was to get a ranking of the conditions for each open question. Based on the resulting rankings, we attempt to answer them in 5.2.3 "Quantitative Analysis".

4. Before the actual tasks, participants were first asked to solve a test task. They were told in which order to rank the three pictures. This made sure they were familiar with SoSci Survey's ranking system and had no technical issues.

Test task

```

1 def print_linewise(text):
2     line = ''
3     for letter in text:
4         if letter == '\n':
5             print(line)
6             line = ''
7         else:
8             line += letter

```

```

1 def print_linewise(text):
2     line = ''
3     for letter in text:
4         if letter == '\n':
5             print(line)
6             line = ''
7         else:
8             line += letter

```

```

1 def print_linewise(text):
2     line = ''
3     for letter in text:
4         if letter == '\n':
5             print(line)
6             line = ''
7         else:
8             line += letter

```

**Figure 5.11:** The three different code markings presented during one of the tasks. Top to bottom are: "Minimal Visuals", "Additional Visuals" and the baseline.

Solving the tasks

5. Now they were given 18 real tasks. Each open question had three associated with it. The task order was randomized, just like the contained pictures.

6. After finishing up the tasks, participants were asked to leave comments. They were invited to leave positive, negative, and neutral feedback via three different text boxes. The results will be discussed in 5.2.4 "Qualitative Analysis".

Thanking participants

7. On the last page of the survey participants were thanked for their participation.

### 5.2.2 Participants

The demographics section gives an insight into the participants of the user study. There was a total of 21 participants. Their ages ranged from 20 to 62 years old. The median age however was 22. Two-thirds (14) of them were male, the other third (7) female. 17 of them study computer science or something related to it. Two are already working in a profession related to computer science. The last two participants study something unrelated. The time they spent in their profession, including studying for it, ranges from 1.5 to 35 years, but the median is much lower at 3.5 years. Programming experience ranges from 1.5 to 45 years, with a median of 6 years. Except for one participant, all had experience programming with Python. Most also had experience with C/C++/C# or Java. The top three IDEs they use are VSCode, Visual Studio, and IntelliJ (in that order). In summary, most participants were fellow computer science students with programming experience you would typically expect. All participants knew either Python or a similar language. This is unsurprising, considering the survey invitation mentioned that basic Python knowledge would be required.

Most participants were typical CS students.

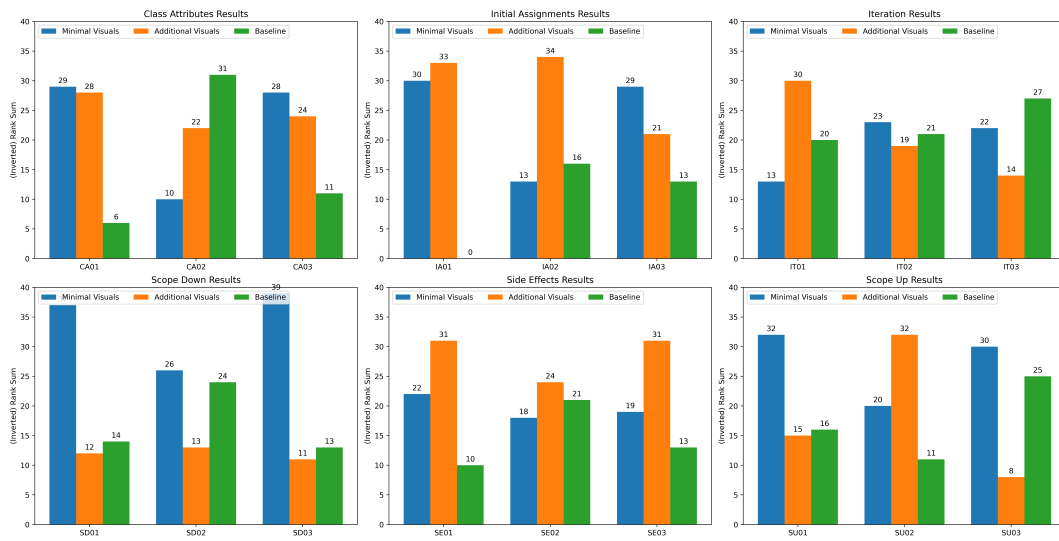
### 5.2.3 Quantitative Analysis

To extract useful information from the ranking results we form rank-sums for each of the conditions. To get more easily readable results, the rankings are inversed and subtracted by one. So rank 1 becomes 2, rank 2 becomes 1, and rank 3 becomes 0. This way the highest rank-sum corresponds to the highest ranking. No information is lost by this conversion. First, we calculate the inverted rank-sums of each condition for each task. Fig. 5.12 shows them, grouped for each investigated question.

Inverted rank-sums were analyzed.

It is clear that for some questions, the results were strongly influenced by the specific task. We will try to provide potential explanations for two examples. The most clear variation is between the tasks of the class attribute question. It might indicate that the context of the code influences,

Strong variations between tasks for some questions.

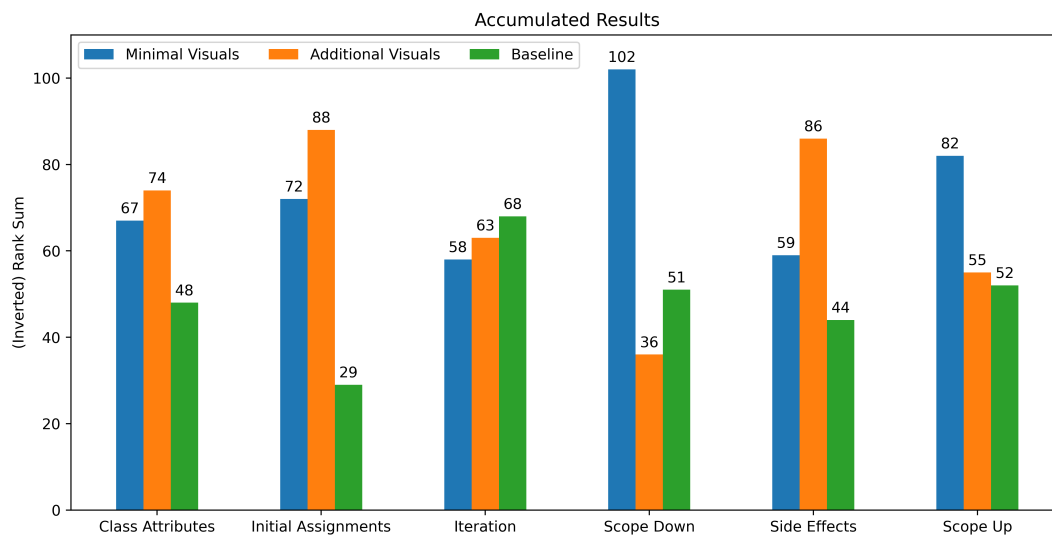


**Figure 5.12:** The graphs show the rank-sums for each task. The tasks are grouped by their question. The condition with the highest bar scores best.

whether users want the entire object to be tracked or not. P12 commented "Highlighting "sibling" data fields of an object and their usage can be useful, but more often they are not.". So this does not yield a definitive answer on whether to include such markings by standard. In the case of the scope-up tasks, "Additional Visuals" scores much better in the second than the other two tasks. This might be, because in the second task going up in scope only leads to two additional markings. Many participants stated too many markings made them rank pictures lower. This likely skewed the results against "Additional Visuals", since it tends to produce the most markings. Especially in tasks going out of scope, since the dependencies cascade easily. We will discuss the issue in more detail in 5.2.4 "Qualitative Analysis". Here it indicates that developers are interested in the markings of going up in scope. But only as long as they do not come with a plethora of markings.

Accumulated results  
per question

Next we sum up the results from the different tasks for each question. Thus there are now three rank-sums per question, one per condition. These are presented in Fig. 5.13.



**Figure 5.13:** The graph shows the accumulated rank-sums for each question. The condition with the highest bar scores best.

A Friedman test was performed on the accumulated rank-sums of each question. This tests for significant differences between the conditions. There are significant differences for three of the six questions. For initial assignments "Minimal Visuals" and "Additional Visuals" are both significantly different from the baseline. Thus it is uncertain whether overwritten initial assignments should be marked. However both conditions mark code very similarly to what is outlined in 3 "Concept and Design Choices". "Additional Visuals" only adds one more marking. This indicates that the concept is useful for developers to some degree. It seems to improve upon what is currently available in IDEs to highlight variables. For going down in scope, the difference between "Minimal Visuals" and the other two is significant. Apparently tracing the scope downwards is not particularly insightful for developers. But this result should be taken with a grain of salt. When tracing down in scope, it usually results in plenty of new markings. Especially for compact code fragments as in the study. As mentioned before participants disliked it when there were a lot of them. This might also explain why "Additional Visuals" performed so badly. Thus it is unclear whether this result supports our decision to limit the tool to the local scope. Apart from that, the difference to the baseline once again

IA results indicate our concept is useful.

Is the local scope limitation validated by the SD results?

Developers prefer SE  
marked.

indicates the usefulness of our concept. Lastly regarding side effects, "Additional Visuals" is significantly different from the other two. Developers appear to prefer being made aware of potential side effects. It should however be noted, that within our study the side effect was always caused within the same file. Thus it was quite easy to detect. Whether this result still holds for side effects not easily apparent, needs to be investigated further.

#### 5.2.4 Qualitative Analysis

Participants left  
comments.

After finishing all tasks, participants were invited to leave comments. They were offered a comment box for positive, negative, and neutral comments. Most participants used them extensively, providing invaluable feedback. The following section will detail what they liked, what they disliked, and potential improvements they suggested.

The approach seems to  
resonate with  
participants.

In general participants responded positively to our approach. While participants were not told the markings are based on code dependencies, many figured that out themselves. Their criticism was more focused on how the code was marked and how much. Like the results of 5.2.3 "Quantitative Analysis", it indicates that providing such context about the current line is a reasonable approach.

Our highlighting method  
skewed the results.

Comments mainly pointed out a significant issue with our study. As already mentioned in 5.2.3 "Quantitative Analysis", excessive highlighting likely skewed the result. Most participants thought that in some cases the highlights were rendered useless due to their amount. Several reasons were causing this issue. When deciding on how to mark the context within the study, we aimed to minimize the influence of the markings. Simple marking rules, like marking the entire line, seemed sensible to achieve this. However, we did not take into account, that marking entire lines would make them very prominent. This was especially problematic due to several compounding factors. As P4 pointed out, sometimes there is a "cascade of [variables leading] up to the line currently worked on". This is related to an issue brought up in 5.1 "Technical Evaluation". The markings are more use-



ful for long scopes because smaller ones tend to be compact and highly interconnected. The result is almost every line being marked. Unfortunately, the small code sizes were a limitation of our study. Otherwise, we would not have been able to test out more than one question in a reasonable time frame. Participants also wrote they often had to choose between too much or too little highlight (e.g. P7 "information of one [picture] was too much but the crucial part of [it] was not present in the reduced form"). But they preferred too little over too much. This indicates our approach needs to be quite careful about how much is being marked. Furthermore, P7 stated, "highlighting conditions [...] can be helpful but only if the outcome does affect the code part, [...] [a] time iterator [...] has an effect but not as much as [...] a break condition". So the usefulness of condition highlights depends on context. Iterators are less interesting than break conditions. Maybe the head of for-loops should be marked less prominently than those of if-statements.

If markings as in the prototype were used, the study would likely have fared better. The context in our approach consists of variables. Our prototype only marks them, instead of the entire accompanying line. Such markings take up much less screen space and are thus significantly less prominent. In addition, the prototype fades out variables depending on how distantly related they are to the current line. Participants' comments indicate this would have been beneficial. P15 wrote that they "preferred when lines were highlighted that were "one" level up". Many other participants agreed, that indirect dependencies are less important to know about. P9 even noted that "maybe recursive dependencies should be marked less occupant, to make direct distinction to direct dependencies more apparent". That of course is exactly what the prototype's fade-out does.

Apart from stating what they liked and disliked, participants also suggested several features. P1 asked that different structures, e.g. assignments and conditions, be given different colors. P12 proposed giving the option to determine how many dependency steps should be traced. This would potentially solve a point brought up by many participants. The amount of context they need depends on factors, such as how familiar the code is and how insightful the

prototype highlights  
likely better.

Participants proposed  
features.

names are. By providing such an option developers could extend the context as far as they need. P12 also suggested enabling the developer to switch to tracing future dependencies instead. P8 also thought that "it can also be helpful to see where [a variable] is being used later on".

## Chapter 6

# Conclusion

### 6.1 Discussion

In general the results of the evaluation are promising. They indicate that an approach on a scope-wide basis could be beneficial. However, the approach will need to be properly studied during usage to make a definitive determination of its usefulness. Especially the dynamic aspect is yet untested. During the study, participants were only given one static marking. They were not allowed to explore different markings by changing the current line. It is also not yet clear, what controls would be best for developers. Currently, the markings update actively, when the current line is switched. Developers might prefer a passive version, where they manually decide when to update them.

Our approach is promising, but requires further testing.

On the technical side, our prototype works in principle. But it also has significant limitations. The most obvious are bugs in the implementation. In particular edge cases need further attention during the analysis. The oversights found will require extending the concept to account for them. Furthermore, two important aspects need to be discussed. The approach is more suitable for longer code scopes. However, it is unclear how long scopes need to be for the tool to be of use. The code length is probably a bad indicator of this. Instead how strongly interconnected the code is, would probably provide a better basis for further investi-

Discussing technical aspects

gation. The prototype only supports a subset of Python. The technical evaluation made clear, that this subset is quite limiting. The current version is not fully applicable to a real-world project yet. Thus the subset will need to be expanded. It seems likely that most aspects of Python will have to be supported. Our prototype focuses on the core language features. The reason is not just because it is the minimum required to test it out. These features are also the ones shared with many other programming languages. Outside of them, many languages tend to have their own quirks. Thus if full language support is necessary, it will make it more difficult to transfer the tool to different languages.

Discussing conceptual  
aspects

Unfortunately the user study conducted did not definitively answer any of the open questions. Still, it provides valuable insights into how developers react to the approach. The main finding was that it is of utmost importance how and what information is presented. It is the difference between the tool being helpful and being a hindrance. Our study used markings that were too prominent. Participants' comments suggest that our prototype presents markings better. Generally, participants preferred too little markings over too much. As a consequence limiting the tool to the local scope is likely better than risking a ton of markings by going beyond it. We believe some features proposed by participants could improve the tool. In particular, the option to set how many dependency steps are being traced. It would enable users to better fine-tune how they explore the code. This perfectly aligns with our goal to allow for dynamic exploration. However other proposed features appear less promising. Having the option to trace dependencies in the other direction as well would probably not be detrimental. But only a few participants mentioned that they wished to explore future assignments. And even those said that they were only interested in some cases. Thus such an extension of the tool seems unlikely to be worthwhile.

## 6.2 Summary and Contributions

This bachelor's thesis presented a new approach to supporting developers' code comprehension and navigation. Developers spend large chunks of time on these aspects of software development. Our approach differs from existing ones by dynamically providing context within scopes. Previous approaches focused on improving a wider understanding of a code project.

Summary

We outlined in detail, which limitations our approach has. In particular, we explained why we limited ourselves to staying within the local scope. Then we described how the local context is determined using an abstract syntax tree. Finally, we detailed how the resulting context is displayed to the user. Afterward, we described the implementation of a prototype.

The prototype was evaluated in two ways. A technical evaluation was performed to determine implementation problems, as well as current limitations. Apart from that, a user study was conducted. It investigated what information constitutes a useful context. We discussed the results of both evaluations in detail. Their results were promising, but also clearly demonstrated a need for further improvements.

In conclusion this thesis contributes a new promising approach. It enables developers to dynamically explore local scopes, thus filling a gap in the research. Along with that two evaluations provide an outline for its future direction.<sup>1</sup>

Contribution

## 6.3 Future Work

As a conclusion, we will outline some potential directions for future work. We think that our approach has shown promise, albeit still flawed in many respects. The most basic way to build upon this thesis is to fix the bugs of the current implementation. After that, it could be expanded to deal with more edge cases. Building on that one could

The implementation needs to be improved.

start to consider further parts of Python in the analysis as well.

Study how to mark the context better.

Otherwise future work could explore how to best mark the context. Our prototype used highlighting and we believe that to be a reasonable starting point. However, there is a wide variety of potential markings that could be explored. Apart from that other visual indicators could be incorporated as well, such as different highlight colors to signal which type of feature is marked. Furthermore, it would be interesting to explore whether the tool can be connected to the call graph. This would enable developers to easily go to and explore connected scopes. It would avoid having to extend the tracing algorithm to go beyond scope borders. Potential features described in 6.1 “Discussion” could also be tested.

Study how well it supports developers.

Lastly so far the approach has not been studied in its intended use-case. Conducting such a study would be able to conclusively prove, whether our approach is helpful. A study could also determine to what degree it is helpful. Especially the dynamic aspect has not been explored by us. Since our approach is intended to supplement wider-scope approaches, it might also be insightful to study how developers switch between tools supporting them in different scopes.

## Appendix A

# User Study Documents

This appendix includes non-task survey pages, as well as the task description and pictures for each of the tasks in the user study. For an example of how the ranking and descriptions looked like see Fig. A.4.

Under each task description we included the following hint (written in grey, "print statement" replaced with statement type in start line):

Please rank the pictures based on how well their green markings help you understand the context of the print statement in the red line.

Rank 1 means most helpful and rank 3 least helpful.


**Tip:** Tap/click on a picture to give it the best unused ranking. Tap/click it again to take away its ranking.

### **Class Attributes**

#### *Task 1*

**Context:** In the below code fragment, a student's school status is printed out after their graduation is processed.

#### *Task 2*



0% completed

**Welcome to our study!**

**Below is a short overview about the content of the study and how your personal data will be handled.**

Purpose: The goal of this study is to obtain insights into how code should be marked to help programmers understand the current context better.

Procedure: The study is conducted online in SoSciSurvey. Participating in this study involves ranking screenshots of lines of python code with colored markings. Additionally, we will collect general demographic information and ask general questions about the experience. All information will be confidential. (See 'Confidentiality' below for details.)

Risks: The study should take approximately half an hour. Beside possible exhaustion or tiredness, there are no known risks. You can take breaks or abort at any time.

Confidentiality: All information gathered during the study will be kept confidential. You will be identified only through identification numbers and background information you divulge in publications, theses, or reports.

Costs and Compensations: Participation in this study is voluntary. You are free to withdraw or discontinue the participation. Participation in this study will involve no cost to you and there will be no financial compensation.

Principal investigator: Christopher Helios, RWTH Aachen University, christopher.helios@rwth-aachen.de

**I have read and understood the information and participate of my own volition.**

Next

Figure A.1: Welcome and consent page of the survey





17% completed

### Demographics

Please tell us some general information about yourself.

Age

Gender

Current Occupation / Field of Study

How long have you been in this profession? Please include the time you spent studying for it. (years)

For how long have you been programming? (years)

Please list programming languages you have used before

Please list IDEs (Code development environments) you have used before

Next

Figure A.2: Demographics form in the survey



33% completed

In the following you will repeatedly rank three pictures.  
They will always show the same piece of code, where the same one line is marked in red and different lines are marked in green.

The red line marks the one you would be currently working on.  
The green markings are automatically created by the IDE to mark context for the red line.

Your task will be to rank the pictures according to the specificity and helpfulness of the green markings for **understanding** the line marked in red.

If you don't understand a code snippet or its context remains unclear, you are able to skip by clicking next without ranking.

Example:

```
1 class Room:
2     def __init__(self, uid, building_uid, professor = '', caretaker = ''):
3         self.uid = uid
4         self.building_uid = building_uid
5         self.professor = professor
6         self.caretaker = caretaker
7
8 building = [Room('A1', 'B12'), Room('A2', 'B12'), Room('A3', 'B12')]
9
10 print('We have a new university building.')
11 print('But we still need to assign professors & caretakers!')
12 for room in building:
13     print('Now we will assign room ' + room.uid)
14     room.professor = input('The professor will be: ')
15     room.caretaker = input('The caretaker will be: ')
16 print('Great! Now the new building is all set up.')
```

Next

Figure A.3: Explanation of the tasks in the survey

### Getting familiar with the Ranking system

Before starting the actual ranking tasks, we would like you to familiarize yourself with how to rank images in SoSciSurvey:

Please assign rank 2 to the image labelled with A, rank 3 to the one labelled B and rank 1 to the one labelled C.

**Tip:** Tap/click on a picture to give it the best unused ranking. Tap/click it again to take away its ranking.

Then press 'next' to begin the actual tasks.

|  |                   |
|--|-------------------|
| <pre> 1 def insertion_sort(numbers: list[float]): 2     i = 1 3     tmp = 0 4 5     while i &lt; len(numbers): 6         j = i 7         while j &gt; 0 and numbers[j-1] &gt; numbers[j]: 8             tmp = numbers[j] 9             numbers[j] = numbers[j-1] 10            numbers[j-1] = tmp 11            j -= 1 12            i += 1           </pre> <p><b>C</b></p> | 1: most helpful   |
| <pre> 1 def insertion_sort(numbers: list[float]): 2     i = 1 3     tmp = 0 4 5     while i &lt; len(numbers): 6         j = i 7         while j &gt; 0 and numbers[j-1] &gt; numbers[j]: 8             tmp = numbers[j] 9             numbers[j] = numbers[j-1] 10            numbers[j-1] = tmp 11            j -= 1 12            i += 1           </pre> <p><b>B</b></p> | 2: medium helpful |
| <pre> 1 def insertion_sort(numbers: list[float]): 2     i = 1 3     tmp = 0 4 5     while i &lt; len(numbers): 6         j = i 7         while j &gt; 0 and numbers[j-1] &gt; numbers[j]: 8             tmp = numbers[j] 9             numbers[j] = numbers[j-1] 10            numbers[j-1] = tmp 11            j -= 1 12            i += 1           </pre> <p><b>A</b></p> | 3: least helpful  |

Next

Figure A.4: Test task to make sure SoSci Survey ranking system works correctly.



67% completed

### 1. Remarks

Please explain what you considered to be helpful context in order to rank the images

Please explain what you considered to be unnecessary context in order to rank the images

Further Comments or final remarks

[Next](#)

**Figure A.5:** Participants were invited to leave comments.



## Thank you for completing this questionnaire!

We would like to thank you very much for helping us.

Your answers were transmitted, you may close the browser window or tab now.

If you have any questions or wish to get into contact, please send an e-mail to:

Christopher Helios, [christopher.helios@rwth-aachen.de](mailto:christopher.helios@rwth-aachen.de)

[Christopher Helios](#), RWTH Aachen – 2024

**Figure A.6:** Thanking participants for participation.

**Context:** The code fragment below enables the user to track their favorite element in the periodic table.

### *Task 3*

**Context:** The code fragment below enables a university to keep track of rooms in their buildings and who is responsible for them.

## **Initial Assignments**

### *Task 1*

**Context:** The below code fragment initializes a new lobby in a multiplayer game, ensuring certain limitations are met.

### *Task 2*

**Context:** The code fragment below is a simple program for registering a new user.

### *Task 3*

**Context:** The code below breaks a string up into lines and prints them individually.

## **Iterations**

### *Task 1*

**Context:** The code below calculates the Fibonacci Sequence and prints it out.

### *Task 2*

**Context:** The code below calculates the total cost of a shopping basket, taking into account the buyer's budget.

### *Task 3*

**Context:** The code below aims to guide the user of an app to give it a rating.

```
1 class Student:
2     def __init__(self, name, school_type, year):
3         self.name = name
4         self.school_type = school_type
5         self.year = year
6
7     student = Student('Max', 'High School', 3)
8
9     print(student.name + ' has graduated.')
10
11     student.school_type = 'University'
12     student.year = 1
13
14     print('Now they attend ' + student.school_type)
15     print('They are starting in year ' + str(student.year))
```

```
1 class Student:
2     def __init__(self, name, school_type, year):
3         self.name = name
4         self.school_type = school_type
5         self.year = year
6
7     student = Student('Max', 'High School', 3)
8
9     print(student.name + ' has graduated.')
10
11     student.school_type = 'University'
12     student.year = 1
13
14     print('Now they attend ' + student.school_type)
15     print('They are starting in year ' + str(student.year))
```

```
1 class Student:
2     def __init__(self, name, school_type, year):
3         self.name = name
4         self.school_type = school_type
5         self.year = year
6
7     student = Student('Max', 'High School', 3)
8
9     print(student.name + ' has graduated.')
10
11     student.school_type = 'University'
12     student.year = 1
13
14     print('Now they attend ' + student.school_type)
15     print('They are starting in year ' + str(student.year))
```

Figure A.7: Class Attributes, task 1, condition 1-3 from top to bottom

```

1 class Element:
2     def __init__(self, name, number, symbol):
3         self.name = name
4         self.number = number
5         self.symbol = symbol
6
7 favorite = Element('Hydrogen', 1, 'H')
8
9 print('Welcome to Favorite Element Tracker!')
10 if input('Current favorite: Hydrogen. Press c to change') == 'c':
11     print("Awesome let's start!")
12     favorite.name = input('Name your favorite element: ')
13     favorite.number = input('Atomic number: ')
14     favorite.symbol = input('Symbol: ')
15 else:
16     print('You really like Hydrogen??')

```

```

1 class Element:
2     def __init__(self, name, number, symbol):
3         self.name = name
4         self.number = number
5         self.symbol = symbol
6
7 favorite = Element('Hydrogen', 1, 'H')
8
9 print('Welcome to Favorite Element Tracker!')
10 if input('Current favorite: Hydrogen. Press c to change') == 'c':
11     print("Awesome let's start!")
12     favorite.name = input('Name your favorite element: ')
13     favorite.number = input('Atomic number: ')
14     favorite.symbol = input('Symbol: ')
15 else:
16     print('You really like Hydrogen??')

```

```

1 class Element:
2     def __init__(self, name, number, symbol):
3         self.name = name
4         self.number = number
5         self.symbol = symbol
6
7 favorite = Element('Hydrogen', 1, 'H')
8
9 print('Welcome to Favorite Element Tracker!')
10 if input('Current favorite: Hydrogen. Press c to change') == 'c':
11     print("Awesome let's start!")
12     favorite.name = input('Name your favorite element: ')
13     favorite.number = input('Atomic number: ')
14     favorite.symbol = input('Symbol: ')
15 else:
16     print('You really like Hydrogen??')

```

Figure A.8: Class Attributes, task 2, condition 1-3 from top to bottom



```

1 class Room:
2     def __init__(self, uid, building_uid, professor = '', caretaker = ''):
3         self.uid = uid
4         self.building_uid = building_uid
5         self.professor = professor
6         self.caretaker = caretaker
7
8     building = [Room('A1', 'B12'), Room('A2', 'B12'), Room('A3', 'B12')]
9
10    print('We have a new university building.')
11    print('But we still need to assign professors & caretakers!')
12    for room in building:
13        print('Now we will assign room ' + room.uid)
14        room.professor = input('The professor will be: ')
15        room.caretaker = input('The caretaker will be: ')
16    print('Great! Now the new building is all set up.')

```

```

1 class Room:
2     def __init__(self, uid, building_uid, professor = '', caretaker = ''):
3         self.uid = uid
4         self.building_uid = building_uid
5         self.professor = professor
6         self.caretaker = caretaker
7
8     building = [Room('A1', 'B12'), Room('A2', 'B12'), Room('A3', 'B12')]
9
10    print('We have a new university building.')
11    print('But we still need to assign professors & caretakers!')
12    for room in building:
13        print('Now we will assign room ' + room.uid)
14        room.professor = input('The professor will be: ')
15        room.caretaker = input('The caretaker will be: ')
16    print('Great! Now the new building is all set up.')

```

```

1 class Room:
2     def __init__(self, uid, building_uid, professor = '', caretaker = ''):
3         self.uid = uid
4         self.building_uid = building_uid
5         self.professor = professor
6         self.caretaker = caretaker
7
8     building = [Room('A1', 'B12'), Room('A2', 'B12'), Room('A3', 'B12')]
9
10    print('We have a new university building.')
11    print('But we still need to assign professors & caretakers!')
12    for room in building:
13        print('Now we will assign room ' + room.uid)
14        room.professor = input('The professor will be: ')
15        room.caretaker = input('The caretaker will be: ')
16    print('Great! Now the new building is all set up.')

```

Figure A.9: Class Attributes, task 3, condition 1-3 from top to bottom

```

1 class Multiplayer_Game_Instance:
2     def __init__(self, lobby_name: str, min_players: int, max_players: int):
3         self.lobby_name = lobby_name
4         self.min_players = min_players
5         self.max_players = max_players
6         if min_players < 2:
7             self.min_players = 2
8             print('Minimum Players raised to ' + str(self.min_players) + '!')
9         if max_players > 64:
10            self.max_players = 64
11            print('Maximum Players lowered to ' + str(self.max_players) + '!')

```

```

1 class Multiplayer_Game_Instance:
2     def __init__(self, lobby_name: str, min_players: int, max_players: int):
3         self.lobby_name = lobby_name
4         self.min_players = min_players
5         self.max_players = max_players
6         if min_players < 2:
7             self.min_players = 2
8             print('Minimum Players raised to ' + str(self.min_players) + '!')
9         if max_players > 64:
10            self.max_players = 64
11            print('Maximum Players lowered to ' + str(self.max_players) + '!')

```

```

1 class Multiplayer_Game_Instance:
2     def __init__(self, lobby_name: str, min_players: int, max_players: int):
3         self.lobby_name = lobby_name
4         self.min_players = min_players
5         self.max_players = max_players
6         if min_players < 2:
7             self.min_players = 2
8             print('Minimum Players raised to ' + str(self.min_players) + '!')
9         if max_players > 64:
10            self.max_players = 64
11            print('Maximum Players lowered to ' + str(self.max_players) + '!')

```

Figure A.10: Initial Assignment, task 1, condition 1-3 from top to bottom

## Scope Down

### Task 1

**Context:** The below code prints out a title with padding on both sides to center it and make it stand out.

### Task 2

```

1 print('Please register yourself:')
2 name = input('Username: ')
3
4 while not name.isalpha():
5     print("Please only enter letters!")
6     name = input('Username: ')
7
8 password = input('Password: ')
9 first_try = password
10 password = input('Confirm password: ')
11
12 while first_try != password:
13     print("Passwords don't match! Please retry.")
14     password = input('Password: ')
15     first_try = password
16     password = input('Confirm password: ')
17
18 print('Welcome ' + name + '!')
```

```

1 print('Please register yourself:')
2 name = input('Username: ')
3
4 while not name.isalpha():
5     print("Please only enter letters!")
6     name = input('Username: ')
7
8 password = input('Password: ')
9 first_try = password
10 password = input('Confirm password: ')
11
12 while first_try != password:
13     print("Passwords don't match! Please retry.")
14     password = input('Password: ')
15     first_try = password
16     password = input('Confirm password: ')
17
18 print('Welcome ' + name + '!')
```

```

1 print('Please register yourself:')
2 name = input('Username: ')
3
4 while not name.isalpha():
5     print("Please only enter letters!")
6     name = input('Username: ')
7
8 password = input('Password: ')
9 first_try = password
10 password = input('Confirm password: ')
11
12 while first_try != password:
13     print("Passwords don't match! Please retry.")
14     password = input('Password: ')
15     first_try = password
16     password = input('Confirm password: ')
17
18 print('Welcome ' + name + '!')
```

Figure A.11: Initial Assignment, task 2, condition 1-3 from top to bottom

```
1  def print_linewise(text):
2      line = ''
3      for letter in text:
4          if letter == '\n':
5              print(line)
6              line = ''
7          else:
8              line += letter
```

```
1  def print_linewise(text):
2      line = ''
3      for letter in text:
4          if letter == '\n':
5              print(line)
6              line = ''
7          else:
8              line += letter
```

```
1  def print_linewise(text):
2      line = ''
3      for letter in text:
4          if letter == '\n':
5              print(line)
6              line = ''
7          else:
8              line += letter
```

Figure A.12: Initial Assignment, task 3, condition 1-3 from top to bottom

**Context:** The below program enables customers of a company to book a vacation with them.

### Task 3

```
1 fibonacci = [0, 1]
2
3 for length in range(2,6):
4     upper = fibonacci[length - 1]
5     lower = fibonacci[length - 2]
6     fibonacci.append(lower + upper)
7     print('Added another one')
8
9 print('The Fibonacci Sequence starts as follows:')
10 print(fibonacci)
```

```
1 fibonacci = [0, 1]
2
3 for length in range(2,6):
4     upper = fibonacci[length - 1]
5     lower = fibonacci[length - 2]
6     fibonacci.append(lower + upper)
7     print('Added another one')
8
9 print('The Fibonacci Sequence starts as follows:')
10 print(fibonacci)
```

```
1 fibonacci = [0, 1]
2
3 for length in range(2,6):
4     upper = fibonacci[length - 1]
5     lower = fibonacci[length - 2]
6     fibonacci.append(lower + upper)
7     print('Added another one')
8
9 print('The Fibonacci Sequence starts as follows:')
10 print(fibonacci)
```

Figure A.13: Iterations, task 1, condition 1-3 from top to bottom

```

1 products = {
2     'Beef': 4.99,
3     'Spoon': 1.99,
4     'Water': 0.49
5 }
6
7 basket = ['Beef', 'Spoon', 'Spoon', 'Water']
8
9 total = 0
10 budget = 20
11
12 for item in basket:
13     total += products[item]
14     if total > budget:
15         print('Oh no that brings us over the budget!')
16         total -= products[item]
17         print('We have to drop that item!')
18
19 print('The total cost is ' + str(total) + '$.')

```

```

1 products = {
2     'Beef': 4.99,
3     'Spoon': 1.99,
4     'Water': 0.49
5 }
6
7 basket = ['Beef', 'Spoon', 'Spoon', 'Water']
8
9 total = 0
10 budget = 20
11
12 for item in basket:
13     total += products[item]
14     if total > budget:
15         print('Oh no that brings us over the budget!')
16         total -= products[item]
17         print('We have to drop that item!')
18
19 print('The total cost is ' + str(total) + '$.')

```

```

1 products = {
2     'Beef': 4.99,
3     'Spoon': 1.99,
4     'Water': 0.49
5 }
6
7 basket = ['Beef', 'Spoon', 'Spoon', 'Water']
8
9 total = 0
10 budget = 20
11
12 for item in basket:
13     total += products[item]
14     if total > budget:
15         print('Oh no that brings us over the budget!')
16         total -= products[item]
17         print('We have to drop that item!')
18
19 print('The total cost is ' + str(total) + '$.')

```

Figure A.14: Iterations, task 2, condition 1-3 from top to bottom

```
1 times = 0
2 got_rating = False
3
4 while not got_rating:
5     times += 1
6     print('Please review our app in the store!')
7     rating = input('Your rating (1-5): ')
8     if rating.isnumeric() and int(rating) == 5:
9         print('Thanks')
10        got_rating = True
11    else:
12        print('Please reconsider')
13    if times > 5:
14        print('You have to rate to continue')
```

```
1 times = 0
2 got_rating = False
3
4 while not got_rating:
5     times += 1
6     print('Please review our app in the store!')
7     rating = input('Your rating (1-5): ')
8     if rating.isnumeric() and int(rating) == 5:
9         print('Thanks')
10        got_rating = True
11    else:
12        print('Please reconsider')
13    if times > 5:
14        print('You have to rate to continue')
```

```
1 times = 0
2 got_rating = False
3
4 while not got_rating:
5     times += 1
6     print('Please review our app in the store!')
7     rating = input('Your rating (1-5): ')
8     if rating.isnumeric() and int(rating) == 5:
9         print('Thanks')
10        got_rating = True
11    else:
12        print('Please reconsider')
13    if times > 5:
14        print('You have to rate to continue')
```

Figure A.15: Iterations, task 3, condition 1-3 from top to bottom

```
1 fill_symbol = '#'
2 title = ' Welcome! '
3
4 ∨ def changeSymbol(new_symbol):
5     global fill_symbol
6     fill_symbol = new_symbol
7
8 print(title.center(25, fill_symbol))
```

```
1 fill_symbol = '#'
2 title = ' Welcome! '
3
4 ∨ def changeSymbol(new_symbol):
5     global fill_symbol
6     fill_symbol = new_symbol
7
8 print(title.center(25, fill_symbol))
```

```
1 fill_symbol = '#'
2 title = ' Welcome! '
3
4 def changeSymbol(new_symbol):
5     global fill_symbol
6     fill_symbol = new_symbol
7
8 print(title.center(25, fill_symbol))
```

Figure A.16: Scope Down, task 1, condition 1-3 from top to bottom

**Context:** Below is a program helping users search for items in their drawers.

**Scope Up**



```

1  daily_cost = {
2      'Berlin': 99, 'London': 144,
3      'Vatican': 67, 'Los Angeles': 237
4  }
5
6  def cost_multiplier(days, loyal):
7      if loyal == 'y':
8          days *= 0.99 ** days
9      return days
10
11 print('Welcome to our booking service!')
12 dest = input('Where would you like to enjoy your vacation?')
13 days = input('How long should your trip be (in days)?')
14 loyal = input('Are you a member in our loyalty program? (y/n)')
15
16 total = daily_cost[dest] * cost_multiplier(int(days), loyal)
17
18 print('Thanks for booking with us!')
19 print('Your total will be ' + str(total) + '$.')

```

```

1  daily_cost = {
2      'Berlin': 99, 'London': 144,
3      'Vatican': 67, 'Los Angeles': 237
4  }
5
6  def cost_multiplier(days, loyal):
7      if loyal == 'y':
8          days *= 0.99 ** days
9      return days
10
11 print('Welcome to our booking service!')
12 dest = input('Where would you like to enjoy your vacation?')
13 days = input('How long should your trip be (in days)?')
14 loyal = input('Are you a member in our loyalty program? (y/n)')
15
16 total = daily_cost[dest] * cost_multiplier(int(days), loyal)
17
18 print('Thanks for booking with us!')
19 print('Your total will be ' + str(total) + '$.')

```

```

1  daily_cost = {
2      'Berlin': 99, 'London': 144,
3      'Vatican': 67, 'Los Angeles': 237
4  }
5
6  def cost_multiplier(days, loyal):
7      if loyal == 'y':
8          days *= 0.99 ** days
9      return days
10
11 print('Welcome to our booking service!')
12 dest = input('Where would you like to enjoy your vacation?')
13 days = input('How long should your trip be (in days)?')
14 loyal = input('Are you a member in our loyalty program? (y/n)')
15
16 total = daily_cost[dest] * cost_multiplier(int(days), loyal)
17
18 print('Thanks for booking with us!')
19 print('Your total will be ' + str(total) + '$.')

```

Figure A.17: Scope Down, task 2, condition 1-3 from top to bottom

```

1 class Drawer:
2     def __init__(self, uid: int, contents: list[str]):
3         self.uid = uid
4         self.contents = contents
5
6     def contains(self, item: str):
7         return item in self.contents
8
9 desk_drawer = Drawer(3, ['Paper', 'Pen', 'Pen'])
10 kitchen_drawer = Drawer(1, ['Knife', 'Potato', 'Spoon'])
11 item = input('What should I search for in the desk drawer?')
12 if desk_drawer.contains(item):
13     print('Found it!')
14 else:
15     print(item + ' is not in the drawer!')
16     if input('Should I search the kitchen instead? (y/n)') == 'y':
17         if kitchen_drawer.contains(item):
18             print('Found it!')
19         else:
20             print(item + ' is not in the drawer!')
21     else:
22         print('Guess I can do without it..')

```

```

1 class Drawer:
2     def __init__(self, uid: int, contents: list[str]):
3         self.uid = uid
4         self.contents = contents
5
6     def contains(self, item: str):
7         return item in self.contents
8
9 desk_drawer = Drawer(3, ['Paper', 'Pen', 'Pen'])
10 kitchen_drawer = Drawer(1, ['Knife', 'Potato', 'Spoon'])
11 item = input('What should I search for in the desk drawer?')
12 if desk_drawer.contains(item):
13     print('Found it!')
14 else:
15     print(item + ' is not in the drawer!')
16     if input('Should I search the kitchen instead? (y/n)') == 'y':
17         if kitchen_drawer.contains(item):
18             print('Found it!')
19         else:
20             print(item + ' is not in the drawer!')
21     else:
22         print('Guess I can do without it..')

```

```

1 class Drawer:
2     def __init__(self, uid: int, contents: list[str]):
3         self.uid = uid
4         self.contents = contents
5
6     def contains(self, item: str):
7         return item in self.contents
8
9 desk_drawer = Drawer(3, ['Paper', 'Pen', 'Pen'])
10 kitchen_drawer = Drawer(1, ['Knife', 'Potato', 'Spoon'])
11 item = input('What should I search for in the desk drawer?')
12 if desk_drawer.contains(item):
13     print('Found it!')
14 else:
15     print(item + ' is not in the drawer!')
16     if input('Should I search the kitchen instead? (y/n)') == 'y':
17         if kitchen_drawer.contains(item):
18             print('Found it!')
19         else:
20             print(item + ' is not in the drawer!')
21     else:
22         print('Guess I can do without it..')

```

Figure A.18: Scope Down, task 3, condition 1-3 from top to bottom

**Task 1**

**Context:** The code below is for a video game and determines what happens when the player is damaged.

**Task 2**

**Context:** The code below enables users to book courses and to receive a discount if they are students.

**Task 3**

**Context:** The code below determines a customer's favorite drink so it can be served to them.

**Side Effects****Task 1**

**Context:** The code below tells the user whether a given word is a palindrome.

**Task 2**

**Context:** The code below approximates the Golden Ratio using the Fibonacci Sequence.

**Task 3**

**Context:** The below program keeps track of ongoing games.

In the following condition 1 is "Minimal Visuals", 2 is "Additional Visuals" and 3 the baseline.

```

1 health = 100
2 armor = 3
3
4 def game_over(debug = False):
5     if debug:
6         print('You cannot die!')
7     else:
8         print('Game over!')
9
10 def take_damage(damage: float, debug = False):
11     global health, armor
12     damage -= armor
13     if debug:
14         print('Health before damage: ' + str(health))
15     if damage > 0:
16         health -= damage
17     if debug:
18         print('Health after damage: ' + str(health))
19     if health <= 0:
20         game_over()

```

```

1 health = 100
2 armor = 3
3
4 def game_over(debug = False):
5     if debug:
6         print('You cannot die!')
7     else:
8         print('Game over!')
9
10 def take_damage(damage: float, debug = False):
11     global health, armor
12     damage -= armor
13     if debug:
14         print('Health before damage: ' + str(health))
15     if damage > 0:
16         health -= damage
17     if debug:
18         print('Health after damage: ' + str(health))
19     if health <= 0:
20         game_over()

```

```

1 health = 100
2 armor = 3
3
4 def game_over(debug = False):
5     if debug:
6         print('You cannot die!')
7     else:
8         print('Game over!')
9
10 def take_damage(damage: float, debug = False):
11     global health, armor
12     damage -= armor
13     if debug:
14         print('Health before damage: ' + str(health))
15     if damage > 0:
16         health -= damage
17     if debug:
18         print('Health after damage: ' + str(health))
19     if health <= 0:
20         game_over()

```

Figure A.19: Scope Up, task 1, condition 1-3 from top to bottom

```
1  ∨ course_cost = {
2    'Cooking': 17, 'Crafting': 54, 'Programming': 23
3  }
4
5  ∨ def cost_multiplier(people, student, debug = False):
6    ∨ if debug:
7      print('Multiplier without student discount: ' + str(people))
8      return (people - student) + (0.8 * student)
9
10 print('Welcome to our booking service!')
11 course = input('Which course would you like to book?')
12 people = input('For how many people do you want to book?')
13 student = input('And how many of these are students?')
14
15 total = course_cost[course] * cost_multiplier(int(people), int(student))
16 print('Thanks for booking with us!')
17 print('Your total will be ' + str(total) + '$.')
```

```
1  course_cost = {
2    'Cooking': 17, 'Crafting': 54, 'Programming': 23
3  }
4
5  def cost_multiplier(people, student, debug = False):
6    if debug:
7      print('Multiplier without student discount: ' + str(people))
8      return (people - student) + (0.8 * student)
9
10 print('Welcome to our booking service!')
11 course = input('Which course would you like to book?')
12 people = input('For how many people do you want to book?')
13 student = input('And how many of these are students?')
14
15 total = course_cost[course] * cost_multiplier(int(people), int(student))
16 print('Thanks for booking with us!')
17 print('Your total will be ' + str(total) + '$.')
```

```
1  course_cost = {
2    'Cooking': 17, 'Crafting': 54, 'Programming': 23
3  }
4
5  def cost_multiplier(people, student, debug = False):
6    if debug:
7      print('Multiplier without student discount: ' + str(people))
8      return (people - student) + (0.8 * student)
9
10 print('Welcome to our booking service!')
11 course = input('Which course would you like to book?')
12 people = input('For how many people do you want to book?')
13 student = input('And how many of these are students?')
14
15 total = course_cost[course] * cost_multiplier(int(people), int(student))
16 print('Thanks for booking with us!')
17 print('Your total will be ' + str(total) + '$.')
```

Figure A.20: Scope Up, task 2, condition 1-3 from top to bottom

```

1 class Customer:
2     def __init__(self, name, favourite_drink):
3         self.name = name
4         self.favourite_drink = favourite_drink
5
6     def change_favourite(self, new_favourite):
7         self.favourite_drink = new_favourite
8         print('I noted down ' + new_favourite + ' as your new favourite.')
9
10 Bob = Customer('Bob', 'Coffee')
11
12 answer = input('Hey Bob! Is ' + Bob.favourite_drink + ' still your favourite? (y/n)')
13 if answer == 'n':
14     Bob.change_favourite(input('Good to know! What do you like instead?'))
15
16 print('Great, I will prepare ' + Bob.favourite_drink + ' for you.')

```

```

1 class Customer:
2     def __init__(self, name, favourite_drink):
3         self.name = name
4         self.favourite_drink = favourite_drink
5
6     def change_favourite(self, new_favourite):
7         self.favourite_drink = new_favourite
8         print('I noted down ' + new_favourite + ' as your new favourite.')
9
10 Bob = Customer('Bob', 'Coffee')
11
12 answer = input('Hey Bob! Is ' + Bob.favourite_drink + ' still your favourite? (y/n)')
13 if answer == 'n':
14     Bob.change_favourite(input('Good to know! What do you like instead?'))
15
16 print('Great, I will prepare ' + Bob.favourite_drink + ' for you.')

```

```

1 class Customer:
2     def __init__(self, name, favourite_drink):
3         self.name = name
4         self.favourite_drink = favourite_drink
5
6     def change_favourite(self, new_favourite):
7         self.favourite_drink = new_favourite
8         print('I noted down ' + new_favourite + ' as your new favourite.')
9
10 Bob = Customer('Bob', 'Coffee')
11
12 answer = input('Hey Bob! Is ' + Bob.favourite_drink + ' still your favourite? (y/n)')
13 if answer == 'n':
14     Bob.change_favourite(input('Good to know! What do you like instead?'))
15
16 print('Great, I will prepare ' + Bob.favourite_drink + ' for you.')

```

Figure A.21: Scope Up, task 3, condition 1-3 from top to bottom

```
1 def reverse_word(word):
2     global reverse
3     for character in word:
4         reverse = character + reverse
5
6 word = input('Please enter a word:').lower()
7 reverse = ''
8 reverse_word(word)
9 print('The reverse is ' + reverse)
10
11 if word == reverse:
12     print('The given word is a palindrome.')
13 else:
14     print('The given word is NOT a palindrome!')
```

```
1 def reverse_word(word):
2     global reverse
3     for character in word:
4         reverse = character + reverse
5
6 word = input('Please enter a word:').lower()
7 reverse = ''
8 reverse_word(word)
9 print('The reverse is ' + reverse)
10
11 if word == reverse:
12     print('The given word is a palindrome.')
13 else:
14     print('The given word is NOT a palindrome!')
```

```
1 def reverse_word(word):
2     global reverse
3     for character in word:
4         reverse = character + reverse
5
6 word = input('Please enter a word:').lower()
7 reverse = ''
8 reverse_word(word)
9 print('The reverse is ' + reverse)
10
11 if word == reverse:
12     print('The given word is a palindrome.')
13 else:
14     print('The given word is NOT a palindrome!')
```

Figure A.22: Side Effects, task 1, condition 1-3 from top to bottom

```

1 fib1 = 0
2 fib2 = 1
3
4 def next_fib():
5     global fib1, fib2
6     fib3 = fib1 + fib2
7     fib1 = fib2
8     fib2 = fib3
9
10 while fib1 < 1000:
11     next_fib()
12
13 golden_ratio = fib2 / fib1
14
15 print('Approximating Golden Ratio using Fibonacci Sequence.')
16 print('Current calculation: ' + str(fib2) + '/' + str(fib1))
17 print('Resulting approximation: ' + str(golden_ratio))

```

```

1 fib1 = 0
2 fib2 = 1
3
4 def next_fib():
5     global fib1, fib2
6     fib3 = fib1 + fib2
7     fib1 = fib2
8     fib2 = fib3
9
10 while fib1 < 1000:
11     next_fib()
12
13 golden_ratio = fib2 / fib1
14
15 print('Approximating Golden Ratio using Fibonacci Sequence.')
16 print('Current calculation: ' + str(fib2) + '/' + str(fib1))
17 print('Resulting approximation: ' + str(golden_ratio))

```

```

1 fib1 = 0
2 fib2 = 1
3
4 def next_fib():
5     global fib1, fib2
6     fib3 = fib1 + fib2
7     fib1 = fib2
8     fib2 = fib3
9
10 while fib1 < 1000:
11     next_fib()
12
13 golden_ratio = fib2 / fib1
14
15 print('Approximating Golden Ratio using Fibonacci Sequence.')
16 print('Current calculation: ' + str(fib2) + '/' + str(fib1))
17 print('Resulting approximation: ' + str(golden_ratio))

```

Figure A.23: Side Effects, task 2, condition 1-3 from top to bottom



```

1 class Game:
2     def __init__(self, name: str, players: list[str]):
3         self.name = name
4         self.players = players
5         self.player_count = len(players)
6
7     def add_player(self, new_player: str):
8         self.players.append(new_player)
9         self.player_count = len(self.players)
10
11 tennis = Game('Tennis', ['Axel', 'Felix', 'Kim', 'Hana'])
12 volleyball = Game('Volleyball', ['Timothy', 'Ada'])
13 chess = Game('Chess', ['Vladimir', 'Miguel'])
14
15 active = volleyball
16
17 volleyball.add_player(['Max'])
18 print(active.name + ': ' + str(active.player_count) + ' players')

```

```

1 class Game:
2     def __init__(self, name: str, players: list[str]):
3         self.name = name
4         self.players = players
5         self.player_count = len(players)
6
7     def add_player(self, new_player: str):
8         self.players.append(new_player)
9         self.player_count = len(self.players)
10
11 tennis = Game('Tennis', ['Axel', 'Felix', 'Kim', 'Hana'])
12 volleyball = Game('Volleyball', ['Timothy', 'Ada'])
13 chess = Game('Chess', ['Vladimir', 'Miguel'])
14
15 active = volleyball
16
17 volleyball.add_player(['Max'])
18 print(active.name + ': ' + str(active.player_count) + ' players')

```

```

1 class Game:
2     def __init__(self, name: str, players: list[str]):
3         self.name = name
4         self.players = players
5         self.player_count = len(players)
6
7     def add_player(self, new_player: str):
8         self.players.append(new_player)
9         self.player_count = len(self.players)
10
11 tennis = Game('Tennis', ['Axel', 'Felix', 'Kim', 'Hana'])
12 volleyball = Game('Volleyball', ['Timothy', 'Ada'])
13 chess = Game('Chess', ['Vladimir', 'Miguel'])
14
15 active = volleyball
16
17 volleyball.add_player(['Max'])
18 print(active.name + ': ' + str(active.player_count) + ' players')

```

Figure A.24: Side Effects, task 3, condition 1-3 from top to bottom



## Bibliography

- [1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, 2020. doi.org/10.1109/VL/HCC50065.2020.9127264.
- [2] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10*, page 2503–2512, New York, NY, USA, 2010. Association for Computing Machinery. doi.org/10.1145/1753326.1753706.
- [3] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, page 207–210, New York, NY, USA, 2010. Association for Computing Machinery. doi.org/10.1145/1810295.1810331.
- [4] Beat Fluri, Michael Wursch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79, 2007. doi.org/10.1109/WCRE.2007.21.
- [5] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. TASSAL: autofolding for source code summarization. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16*, page 649–652, New York, NY, USA, 2016. Association for Computing Machinery. doi.org/10.1145/2889160.2889171.
- [6] R.L. Glass. Persistent Software Errors. *IEEE Transactions on Software Engineering*, SE-7(2):162–168, 1981. doi.org/10.1109/TSE.1981.230831.

- 
- [7] Sangmok Han, David R. Wallace, and Robert C. Miller. Code Completion from Abbreviated Input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2009. doi.org/10.1109/ASE.2009.64.
- [8] C. Hannebauer, M. Hesenius, and V. Gruhn. Does syntax highlighting help programming novices? *Empirical Software Engineering*, 23:2795–2828, 2018. doi.org/10.1007/s10664-017-9579-0.
- [9] Austin Z. Henley and Scott D. Fleming. The patchworks code editor: toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '14*, page 2511–2520, New York, NY, USA, 2014. Association for Computing Machinery. doi.org/10.1145/2556288.2557073.
- [10] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 217–227, 2017. doi.org/10.1109/SANER.2017.7884623.
- [11] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing Source Code using a Neural Attention Model. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics. doi.org/10.18653/v1/P16-1195.
- [12] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An Empirical Study Assessing Source Code Readability in Comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution (IC-SME)*, pages 513–523, 2019. doi.org/10.1109/ICSME.2019.00085.
- [13] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST '11*, page 217–224, New York, NY, USA, 2011. Association for Computing Machinery. doi.org/10.1145/2047196.2047225.
- [14] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD '05*, page 159–168, New York, NY, USA, 2005. Association for Computing Machinery. doi.org/10.1145/1052898.1052912.
- [15] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14*, page 1–11, New

- York, NY, USA, 2006. Association for Computing Machinery. doi.org/10.1145/1181775.1181777.
- [16] Amy J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, page 126–135, New York, NY, USA, 2005. Association for Computing Machinery. doi.org/10.1145/1062455.1062492.
- [17] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. doi.org/10.1109/TSE.2006.116.
- [18] Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How tools in IDEs shape developers' navigation behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '13*, page 3073–3082, New York, NY, USA, 2013. Association for Computing Machinery. doi.org/10.1145/2470654.2466419.
- [19] Per Ola Kristensson and Chung Leung Lam. Aiding programmers using lightweight integrated code visualization. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2015*, page 17–24, New York, NY, USA, 2015. Association for Computing Machinery. doi.org/10.1145/2846680.2846683.
- [20] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, page 492–501, New York, NY, USA, 2006. Association for Computing Machinery. doi.org/10.1145/1134285.1134355.
- [21] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806, 2019. doi.org/10.1109/ICSE.2019.00087.
- [22] Seonah Lee, Sungwon Kang, Sunghun Kim, and Matt Staats. The Impact of View Histories on Edit Recommendations. *IEEE Transactions on Software Engineering*, 41(3):314–330, 2015. doi.org/10.1109/TSE.2014.2362138.
- [23] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, page 291–294, New York, NY, USA, 2014. Association for Computing Machinery. doi.org/10.1145/2597008.2597793.

- [24] Roberto Minelli, Andrea Mocci, and Michele Lanza. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015. doi.org/10.1109/ICPC.2015.12.
- [25] Leon Müller. User-Centered Edit Recommendations in IDEs. Master’s thesis, RWTH Aachen University, Aachen, May 2023. URL <https://hci.rwth-aachen.de/publications/mueller2023a.pdf>.
- [26] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, 2010. doi.org/10.1109/MS.2009.161.
- [27] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension, ICPC ’20*, page 2–13, New York, NY, USA, 2020. Association for Computing Machinery. doi.org/10.1145/3387904.3389258.
- [28] Hendrik Strobelt, Daniela Oelke, Bum Chul Kwon, Tobias Schreck, and Hanspeter Pfister. Guidelines for Effective Usage of Text Highlighting Techniques. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):489–498, 2016. doi.org/10.1109/TVCG.2015.2467759.
- [29] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguang Huang, Zhelin Zhu, and Bin Luo. AST-trans: code summarization with efficient tree-structured attention. In *Proceedings of the 44th International Conference on Software Engineering, ICSE ’22*, page 150–162, New York, NY, USA, 2022. Association for Computing Machinery. doi.org/10.1145/3510003.3510224.
- [30] T. Tenny. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988. doi.org/10.1109/32.6171.
- [31] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. doi.org/10.1109/TSE.2017.2734091.
- [32] Binhang Yuan, Vijayaraghavan Murali, and Christopher Jermaine. Abridging source code. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi.org/10.1145/3133882.
- [33] Michael Zhivich and Robert K. Cunningham. The Real Cost of Software Errors. *IEEE Security Privacy*, 7(2):87–90, 2009. doi.org/10.1109/MSP.2009.56.
- [34] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings. 26th International Conference*

*on Software Engineering*, pages 563–572, 2004. doi.org/10.1109/ICSE.2004.1317478.





---

# Index

|  |   |
|--|---|
| Abstract Syntax Tree .....               | 7, 14, 15, 17–19, 25–27, 30   |
| AST .....                                | <i>see</i> Abstract Syntax Tree   |
| code comprehension .....                 | 1–5, 7–9, 13, 14, 49  |
| concept .....                            | 4, 13–23, 29–32, 43, 44, 47   |
| context .....                            | 3, 4, 10, 14, 15, 17–19, 22, 25, 27, 29, 31, 32, 34, 35, 37–39, 41, 44–46, 49, 50 |
| evaluation .....                         | 29–46   |
| extension .....                          | 3, 4, 10, 15, 25, 26, 30, 38  |
| future work .....                        | 49–50   |
| IDE .....                                | <i>see</i> Integrated Development Environment                                     |
| Integrated Development Environment ..... | 1–3, 5, 8–16, 19, 22, 25, 39, 41, 43  |
| navigation .....                         | 1–5, 8–10, 13, 14, 23, 49   |
| prototype .....                          | 3, 4, 15, 17, 18, 22, 25, 29–33, 36, 37, 45, 47–50                                |
| Python .....                             | 14, 15, 18, 19, 25, 26, 30, 32, 41, 48, 50  |
| user study .....                         | 4, 29, 34–46, 48, 49  |
| VSCoDe .....                             | 15, 25, 27, 38, 41  |



