Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

User-Centered Edit Recommendations in IDEs

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

by
Leon Müller

Thesis advisor:
Prof. Dr. Jan Borchers

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

Registration date: 02.12.2022
Submission date: 24.05.2023

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung

Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

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

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

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.

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or 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) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 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

List of Figures

List of Tables

Abstract

More than a third of software developers' working time is spent navigating and comprehending source code. Despite improvements and innovations in modern Integrated Development Environments (IDEs), the issue remains. Research has shown successful navigation support through prominent call graph recommendations, however, these tools have not yet seen adoption in practice.

Recently, recommendation systems for software engineers (RSSEs) have shown promising development and increasing recommendation accuracy. These systems mine frequent patterns from developer interaction histories to uncover evolutionary couplings between code elements. These couplings can then be used to recommend related code artifacts to other developers based on their current work context. Despite the technological advancements in the field, these tools have also not been implemented into IDEs.

We suspect that this could partly be due to the lack of Human-Computer Interaction (HCI) research on the tools as many of them exist purely theoretically without actual interface implementations. In this thesis, we propose a novel approach that combines structural recommendations from the call graph with evolutionary recommendations from interaction histories. The recommendations are integrated into the VS Code IDE through an interactive tree-based view and a color highlighting system. In a user study, we show that such a system can support developers in navigating and comprehending unfamiliar source code. We also show that going beyond standard list-based recommendation outputs can significantly improve developers' mental models of code projects.

Überblick

Softwareentwickler verbringen mehr als ein Drittel ihrer Arbeitszeit damit Source Code zu navigieren und zu verstehen. Dieses Problem besteht fortwährend trotz der Entwicklung und der vielen Innovationen in modernen IDEs. Forschung in dem Feld hat gezeigt dass Entwickler durch prominent dargestellte Vorschläge aus dem Call Graph in ihrer Navigation unterstützt werden können. Jedoch haben diese Werkzeuge bisher keinen Einzug in die Praxis gehalten.

In den vergangenen Jahren haben *recommendation systems for software engineers* (RSSEs) vielversprechende Fortschritte in Sachen Vorschlagsgenauigkeit gezeigt. Diese Systeme extrahieren häufige Muster aus dem Navigationsverhalten von Entwicklern um so evolutionäre Zusammenhänge zwischen Code Elementen zu finden. Diese Zusammenhänge können genutzt werden um anderen Entwicklern relevante Code Elemente basierend auf ihrem aktuellen Arbeitskontext vorzuschlagen. Trotz der technischen Fortschritte in dem Gebiet haben auch diese Tools bisher keine Implementierung in modernen IDEs gesehen.

Wir vermuten dass dies durch den Mangel an Human-Computer Interaction (HCI) Forschung bezüglich dieser Tools bedingt sein könnte. Insbesondere, da viele dieser Tools rein algorithmisch existieren aber keine Interface Implementierung haben. In dieser Arbeit stellen wir ein neuartiges Framework vor welches strukturelle Vorschläge aus dem Call Graph mit evolutionären Vorschlägen aus Interaktionshistorien kombiniert. Diese Vorschläge werden in der Visual Studio Code IDE durch interaktive Baumansichten und ein Farbmarkierungssystem integriert. In einer Nutzerstudie zeigen wir dass ein solches System in der Tat Entwickler in ihrer Navigation in unbekannten Codeumgebungen unterstützen kann. Desweiteren zeigen wir dass Visualierungskonzepte für Vorschläge, welche über einfache Listenformate hinausgehen, deutliche Verbesserungen im Projektverständnis von Entwicklern erzielen können.

Acknowledgements

I would like to thank my supervisor, Adrian Wagner, for his continued support over the course of this thesis and for giving me the options and freedom to pursue my interests in this particular field of research.

Also, I would like to thank Prof. Dr. Jan Borchers for the opportunity of writing my thesis with the ever-supportive Media Computing Group. Thank you as well, everyone, at the chair that has helped me in one way or another, especially everyone that kindly participated in the user study.

I wish to thank all the special people around me for being so supportive and also welcome distractions at times. Finally, I wish to thank my Mum, Anna, and Judy.

Für Papa.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in colored boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

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

`RecommendationView`

Named tools and concepts are written in italics.

get Recommendations

The whole thesis is written in American English.
When writing about unspecified, singular persons, female pronouns are used.

Chapter 1

Introduction

> *"A central goal of software engineering is to improve developers' productivity and the quality of their software. This requires an efficient and effective way to explore code since most developers will encounter code with which they are not familiar. Understanding code in modern codebases is challenging because of the size and complexity of the codebase, and the use of indirection."*
>
> *—LaToza and Myers [2010]*

1.1 Motivation

Software drives the world. And we have HCI research to thank for its usability and accessibility.

In today's world, software is everywhere. This statement has been used and thrown around for several decades now. But as time has gone by the phrase has gotten closer and closer to actually being true. Software helps us drive our cars, control the heating in our homes, and lets us continue working in teams even during a pandemic. On top of that, there is an entire digital world that we have constructed in parallel to the physical world. Social networks connect us to friends and peers around the globe and digital entertainment has long overtaken theatres and record stores. Speaking to someone on the other side of the earth, face to face, in real-time, through a device that fits in our pockets has not been miraculous to us for many years now. A big part of our modern lives is based on some kind of functioning software. Interacting with – and to some extent relying on – software has become imperative in order to take an active part in modern society. Hence, we have seen big improvements not only in the quality and limitations of the software we use but also in the way we can interact with it. Through research in human-computer interaction (HCI), the usage of modern technologies has become intuitive for large parts of the population and more inclusive for more people to benefit from it.

HCI also concerns how developers create software.

However, we do not only interact with software as consumers. As developers, we also shape and create it. This marks another important interface between humans and computers: How programmers write source code and create software. The research and development of support tooling and development assistance have culminated in today's programming languages and integrated development environments (IDEs) as quintessential tools for software development.

Source Code Navigation

A big part of developer interaction with code is navigation.

A central goal of software engineering still is the improvement of developer productivity and – closely related – the quality of the software they produce (LaToza and Myers [2010]). Both are highly dependent on two factors. First, how efficiently developers can do the actual coding work, and second, how efficiently they can explore and navigate written code. The first factor is heavily supported in today's IDEs through assistance tools such as syntax highlighting, modern debuggers, or code completion tools. However, every bit of coding done by a programmer is usually preceded by a navigational act to the relevant point in the program. Navigation and exploration of source code are crucial to understanding the structure of a program, dependencies within the code, and specific contexts in which code fragments execute. The better code can be explored, the better the developer's mental model of the code becomes. A better mental model subsequently leads to more

Call graph tools can help developers navigate more efficiently.

efficient code navigation. IDEs support code navigation through the standard package explorer, search tools, and more specialized tools such as the call hierarchy. Research by Murphy et al. [2006] has shown that Java developers in Eclipse do the most navigation using the explorer. For code navigation with the explorer to be efficient however, the developer needs to already have a good idea where what she is looking for might be located. Advanced call hierarchy tools (such as *Stacksplorer* by Karrer et al. [2011]) allow for code exploration beyond the developer's mental model along the call graph of a given context. These tools have proven to reduce task completion times in software development, by prominently displaying their information within the IDE (Krämer et al. [2013]). Despite the availability of some assistance tools, their adoption into modern IDEs has been lackluster and as a result code navigation still takes up immense portions of developers' time. Developers spend 35% of their time navigating (Ko et al. [2006]) and in general 50% of their time foraging for information related to code comprehension (Piorkowski et al. [2013]). In economic terms, navigation or comprehension tasks like maintenance and debugging account for up to 70% of total expenses in software projects (Pressman [2005]). Comprehending the source code and maintaining mental models is one of the biggest issues developers face today. And it is especially challenging for inexperienced programmers or developers that have joined new teams and are working on unfamiliar code bases (LaToza et al. [2006]). A recently published study[1] conducted with over 500 developers from big software development organizations has shown that code bases have seen significant growth in volume over the past decade. The study found that this has further increased the challenges that new developers face and that code comprehension is now more difficult than ever, resulting in more code breaks. It is becoming increasingly apparent that there is a need for new and improved navigation tools to assist developers – both experienced and new – in exploring code and boosting code comprehension.

As codebases get bigger and bigger, developers spend most of their time navigating and comprehending code.

[1] https://info.sourcegraph.com/hubfs/CTA%20assets/sourcegraph-big-code-survey-report.pdf (accessed on May 11th, 2023)

Recommendation Systems for Software Engineering

An emerging field of research could help tackle these emerging challenges. Recommendation Systems for Software Engineering (RSSEs) are tools that aim to provide developers with valuable information based on their current working context (Robillard et al. [2010]).

Definition: *Recommendation Systems for Software Engineering (RSSEs)*

RECOMMENDATION SYSTEMS FOR SOFTWARE ENGINEERING (RSSES):
RSSEs are tools that assist developers in their decision-making. They provide developers with information that is potentially interesting or related to what they are currently working on. In this thesis, we are particularly interested in RSSEs that guide developers to related code elements and assist them in their navigation.

"Developers who worked on this piece of code also worked on these code pieces ..."

These systems are similar to those famously deployed in search engines or e-commerce applications. Whereas these famous relatives provide recommendations such as "Customers who bought this item also bought ...", we are interested in RSSEs that provide recommendations along the lines of "Developers who worked on this piece of code also worked on these code pieces ...". Or generally speaking, RSSEs that output code fragments that might be points of interest (POIs) for the developer. The recommendations produced by these systems are typically ignorant of dependencies of structural kind, like those that are displayed in the call hierarchy. They produce recommendations based on past developer interactions with the code. By detecting frequent patterns in interaction histories or commit histories, they find couplings between code fragments that might not be visible in the call hierarchy. According to a recent survey, this type of RSSE makes up the biggest branch of recommendation systems for software engineering (Gašparič and Janes [2015]). These systems have been researched and discussed as potential improvements for code navigation, new developer assistance, and code comprehension. However, the RSSEs developed by the scientific community are not yet deployed in practice and their adoption in modern IDEs has been non-existent. Among other reasons Gašparič and Janes [2015] identify a lack of UI integration, context-less recommendations, lackluster recommendation explanations, and reactive (as opposed to proactive) systems as some of the most likely reasons why we do not see RSSEs in real-world practice. These reasons can be summed up as follows: There has been a lack of HCI research and application of good HCI practices to the developed solutions. While there have been lots of advancements in the field on the technical side, user-centered research has been cut short.

There have been promising technical advancements but no UI integration or HCI research.

In this section, we have seen two promising approaches that try to improve code navigation through tool assistance. Both approaches – the call graph tools, and the RSSEs – however, aim to provide the developer with wholly different kinds of POIs. While call graph tools display strictly hierarchical code dependencies, the RSSEs display relevant code pieces whose relation is derived from past developer interactions. These different POI sets are not always completely distinct but, as we will see later in this thesis, can be far from identical. Choosing only one set of the two to base navigation recommendations on seems insufficient.

Call graph tools and RSSEs are both promising approaches to support code navigation and could be combined.

1.2 Approach

In this thesis, we aim to tackle some of the challenges and shortcomings that have been motivated in the previous section. To do so, we developed a framework for a RSSE within the popular IDE Visual Studio Code[2] (VS Code) that provides a rich extension API. The two major takeaways from 1.1 "Motivation" that we built upon are:

We want to build an RSSE framework that combines recommendations from the call graph with those from interaction histories.

1. Call hierarchy tools can significantly improve developer navigation if they promote the call graph information prominently, and
2. The technical side of navigational RSSEs is promising but the lack of good UI practices and integration could be what is missing for them to see practical use.

[2] https://code.visualstudio.com (accessed on May 11th, 2023)

It should be proactive and provide more visual output than a standard list.

The resulting system merges information from the call graph with data-driven recommendations based on past developer activity and displays it prominently in the IDE's interface. Following the suggestions from the literature, the framework implements a richer integration of recommendations into the IDE beyond that of standard list output. The system acts proactively meaning that it does not rely on the developer to prompt or query it for recommendations. Rather, based on the current working context of the developer, suiting POI recommendations are automatically generated and displayed. Additionally, a color highlighting system is implemented to provide reasoning behind why certain artifacts are recommended.

1.3 Thesis Goals

By taking the successes of several previous research endeavors, merging them, and integrating them into a popular IDE by employing good UI design practices, we hope to get a better understanding of what could be the missing pieces to see the use of RSSEs for navigation in practice. We also aim to show that a proactive recommendation system for potential points of interest can aid developers in navigating unfamiliar code bases. We define the following, more specific goals for this thesis:

G0: **Novel Recommendation Framework** Conceptualize and implement a novel framework combining structural and collaborative recommendation techniques.

G1: **New Developer Assistance** Improve navigation for developers in big, unfamiliar software environments. These scenarios have proven to be especially harsh code navigation challenges and require addressing.

G2: **Code Comprehension Support** Improve code comprehension/mental models of developers through assisted code exploration. Proactively recommending potential related points of interest in the code should help developers create a richer mental model of the software project.

G3: **Towards RSSE Adoption** Show that developers can successfully use RSSEs and want to do so when they are available and integrated well into the IDE.

G4: **New Output Mode** Show that Color Highlighting for recommendation explanation and user guidance could be a valuable aspect of RSSEs.

G5: **Comparison** Evaluate different interface aspects and compare their impact on navigation efficiency.

A framework that combines the successful research on call graph integration into the IDE's UI with the promising technical side of RSSE research could be a good step in the direction of tackling modern code navigation challenges. Especially in big code environments, and for new developers.

1.4 Structure

This thesis will be structured as follows. Chapter 2 will review some of the literature that has preceded this work. This will include both relevant technological foundations as well as more HCI-focused research that has provided insight into the areas of code navigation and RSSEs. Building upon these foundations we have developed a code edit recommendation system for VS Code. In Chapter 3, we present the conceptual approach to the system and explain the design choices that led up to its implementation. Afterward, in Chapter 4, the actual implementation will be presented on a more technical level. To evaluate how well different parts of the system are able to aid developers in navigating unfamiliar code we conducted a user study. The setup and results of a user study are presented and discussed in Chapter 5. Finally, a summary of the thesis, as well as an outlook on possible future work on the topic, will be given in Chapter 6.

Chapter 2

Related work

In this chapter, we will introduce some of the related work and literature that preceded this thesis. This will include research on how developers use IDEs and the field of source code navigation. Behavioral insights as well as tools that leveraged these will be presented and discussed. Also, a brief history of RSSEs will be given. Various technical advancements, different recommendation sourcing strategies, and shortcomings will be discussed in detail.

2.1 IDE Usage and Source Code Navigation

Modern IDEs are powerful tools that assist software developers in most of their day-to-day work scenarios. With tools like code completion, integrated version management, refactoring, interactive debuggers, and many more they now provide a wide range of assistance for developers. In the past decades, a multitude of studies has been conducted observing how developers interact with IDEs, what strategies and tools they deploy to solve their tasks, and in general how programmers go about spending their work days (Ko et al. [2006], LaToza and Myers [2010], LaToza et al. [2006], Amann et al. [2016], Minelli et al. [2015], Murphy et al. [2006], Sharafi et al. [2022]).

Ko et al. found that developers spend 35% of their time navigating.

In a landmark study, Ko et al. [2006] investigated how developers collect and manage information in software maintenance tasks. Observing a group of developers working on these maintenance tasks, they discovered that around 35% of the developers' time was spent (often redundantly) navigating between code fragments across the code base. A number that sparked the initial interest in source code navigation and invited suspicions that the way we as developers navigate our code projects in IDEs may be a bottleneck to our efficiency.

Developers create mental models of the code they work on. These are closely linked to navigation.

A study on the work habits of developers by LaToza et al. [2006] discovered that the most prominent challenge of developers is comprehending source code. Code comprehension requires vast mental models that are maintained and constructed with knowledge about a software project such as code relationships, change impact chains and information about where in the project certain functionalities are located. Maintaining a mental model is how developers understand their code and building it is done through code exploration and navigation. The creation of a mental model is labor- and time-consuming – especially the bigger a software project is – for both experienced and new developers alike. New developers joining existing teams however present an especially tough case as "creating a mental model from scratch requires a lot of energy for the new team member and the team as a whole." (LaToza et al. [2006]).

Code comprehension and navigation are dependent on each other.

The two concepts of code comprehension and code navigation are closely linked. A good understanding of the inner works of a project enables the developer to efficiently navigate between code elements. Vice versa, having a good sense of where things are located in a project and how to traverse along their relationships is fundamental to understanding how the code functions.

In a subsequent study by LaToza and Myers [2010], the authors investigate in more detail how exactly developers go about code comprehension. Based on findings that understanding control flow is essential for program comprehension – and often the first step in creating a mental model (Detienne [2001], Pennington [1987]) – LaToza and

Myers observed how bugs are introduced into software, asked developers what questions they often asked themselves about code, and looked at developers critical activities in the workflow. They found that all of the above issues can be associated with reachability questions. A reachability question is a search along possible paths through a program, looking for certain statements or code blocks that match some initial search criteria. In other words, they concern how pieces of code in a project are related to each other and how one can efficiently traverse from one to the other in the IDE. The study found that most issues regarding comprehension and navigation can be related to difficulties in answering these reachability questions. The authors suggest that effectively helping developers answer these questions could prove to reduce the time spent on code comprehension and navigation. Typically, reachability questions are answered through code exploration and standard IDE tools (like call hierarchy views) prove to not be helpful enough to do so efficiently.

Many developer issues can be formulated into reachability questions. How do get I get from point A to point B in the code without necessarily knowing what B is?

Instead of analyzing how developers interact with code in the IDE, Lawrance et al. [2007, 2008] applied insights from information foraging theory (Pirolli and Card [1999]) to investigate how developers navigate code bases. Adapted from information foraging in web navigation, each link to a source code artifact has a scent that determines how likely a developer in a certain debugging context is to navigate across that link. The scent is determined by topological information as well as word-match likeness with words in the bug report. The resulting model can be used to predict where a developer will navigate in search of information during a debugging task. Study results (Lawrance et al. [2013]) were promising and matched results from prediction models based on interaction data. However, this approach requires the existence of a bug report in order to gain context on the task of the developer. Therefore this approach would be less applicable in assisting new developers in their navigation. In another study, Piorkowski et al. [2016] used information foraging theory to show that over 50% of developers' navigation actions are less productive than they predict prior to the action. The majority of navigational choices are therefore disappointing and less valuable than expected, solidifying the navigational issue

Lawrence et al. used information foraging to predict developer navigations.

Piorkowski et al. found that more than half of developer navigation actions are disappointing.

of modern IDEs. The authors of the study survey existing development support tools and find that only a few of them aim to increase the value of navigational choices. As a result, they argue that future development assistance tools should focus on providing navigational support that actively increases the efficiency of source code navigation.

The most used development tools are package explorers and search tools.

Singer et al. [2010] conducted a study on the daily activity data of developers. They analyzed the tools used by software engineers and found that search tools were among the most prominently used, confirming that efficient code exploration is one of the main points of interest when trying to improve developers' efficiency. Similarly, Murphy et al. [2006] investigated how developers use the Eclipse IDE for Java and found that the two most frequently used navigation views in Eclipse were the Package Explorer – leading by a big margin – and the Search view. By using sequential pattern mining on developer interaction data, Damevski et al. [2017] came to similar results, that developers often use search tools for their navigational purposes and in doing so visit up to 8 unrelated results before finding the desired code fragment or abandoning the search. More recently, Amann et al. [2016] conducted a study on developers in the Visual Studio IDE to understand how much time is spent using the different features and tools of the IDE. The goal of the study was to gain insights into how to support developers and what the next generation of IDEs should look like. The authors found that 28.5% of the developers' time was spent on code editing and execution, closely followed by navigating documents and source code, which took up 22.4% of the time spent. However, they do not include the additional 37.6% time spent classified as *short inactivity* in the navigation data, even though research by Minelli et al. [2015, 2014] suggests that this time of inactivity is likely to be connected to source code comprehension. As we have seen above, source code comprehension and navigation are closely intertwined concepts. Amann et al. found that often developers navigate without the intention of editing code, further suggesting that navigation and comprehension may be two concepts of developing work that can not be regarded as independent of one another. Confirming the earlier work of Murphy and Singer, they also found that the most frequently used navigational tools in the IDE were search tools such as textual and code search. These studies on the usage of IDE tools and the gained insight that search tools are the most frequently used among them suggest that the state of the art of source code navigation in IDEs may be tailored to use cases in which the developer already knows what she is looking for. As text-based searches rely on contextual-knowledge input by the developer they may be less useful for new developers or developers working on unfamiliar code. Amann et al. conclude that ″new concepts for code understanding and exploration might be valuable for developers″.

Minelli et al. confirm that vast amounts of development time are spent on comprehension and call for better IDE interfaces.

In an effort to visualize developer interactions in the IDE and gain insight into how developers use the UI and to what extent the UI supports the developers in their work, Minelli et al. [2014] developed a tool called *DFlow* that monitors interactions of programmers in the IDE. In a later study by Minelli et al. [2015], they abstracted the fine-grained interaction data into higher-level activities: *understanding, navigation, editing, UI interaction* and *time spent outside the IDE*. They found that a total of 70% of developer time could be attributed to program comprehension and argue that future IDEs should make an effort to tackle these challenges. It should be noted here that the used notion of *understanding* includes most idle times between actions and *navigation* only included actual usages of the explorer and search tools. The authors conclude that ″IDEs are far from perfect when it comes to the way their user interfaces are built″. And this is a call that can be found throughout the literature on IDE usage. Looking at the discussed studies and gained insights, the shortcomings of modern IDEs become apparent. The closely tied concepts of source code navigation and program comprehension still present a big challenge for developers and define most of the faced issues and time spent in a workday. The underlying issues – although present for all kinds of developers – become all the more important when we consider scenarios in which developers are either new to the profession altogether or simply new to the code they are working with.

2.2 Supporting Developers in Navigation

In this section, we will take a closer look at some of the tools that have been developed with the goal to tackle the challenges introduced in the previous section. These tools share the common denominator of trying to support developers in the way they comprehend code, find specific code fragments, or simply explore code bases. In other words, they provide novel ways of source code navigation.

In the following, we will divide the approaches into new IDE paradigms, call-hierarchy-based tools, and various other approaches.

2.2.1 New IDE Paradigms

Software engineers spend a third of their work time in IDEs. The IDE paradigm itself might be part of the navigation and comprehension issues. These tools try to shift away from the standard window and file-based IDE paradigm.

Professional software engineers spend about a third of their time working in IDEs (Sillitti et al. [2012]). It should come as no surprise that this makes IDEs the most used application group in a developer's work life. Issues that developers commonly face – such as the comprehension and navigation issues discussed in Section 2.1 – are therefore likely to be issues in the way that modern IDEs are designed. Standard IDEs are designed to be file-based and follow a window/tab paradigm in which each file is contained in one window and typically only one window at a time is in an editable visual focus. Several research approaches have identified this standard way of designing IDEs, in combination with the scattered nature of modern code bases, as a culprit for the navigational issues developers face.

Window-based IDEs lead to cluttered workspaces. *Autumn Leaves* calculates the relevance of open windows in the IDE and closes irrelevant ones.

Roethlisberger et al. [2009] argue that the need for keeping track of many code artifacts across different files leads to a *window plague* in IDEs. Developers often find themselves misnavigating and faced with a large number of open tabs and windows that lead to further confusion and negative impact on navigation. As it is often unclear which windows still bear reasonable importance and which ones can be closed without causing any future inconvenience, developers tend to leave open way more windows than necessary. Röthlisberger et al. introduce *Autumn Leaves*, a tool that assigns weights to open windows and automatically adjusts those weights based on certain user interactions. Weights of rarely visited windows decrease over time while frequently used windows receive a higher weighted importance. Based on the windows' individual weights, they may be greyed out if deemed less relevant, displayed more prominently if they are more likely to be interacted with, or closed altogether if they become very unlikely to be needed in the near future. In an evaluation of the tool, the authors found it to accurately find candidates for closing, however, the automatic closing feature was not well received by developers.

Code Bubbles shifts away from file-based visualization. Code fragments can be put into bubbles and grouped with related bubbles.

Other approaches focus on redesigning the file-based paradigm of IDEs itself. Bragdon et al. [2010] present a non-file based development environment called *Code Bubbles*. Instead of assigning each code file or document a single editable window, *Code Bubbles*' user interface consists of many small editable fragments of code presented in so-called bubbles. Fragments can be grouped together into screen-sized views, giving developers the ability to simultaneously view and edit fragments strewn across many classes and files. The concept achieved good usability results in a qualitative user study.

Code Canvas lets developers arrange all of a project's code on an infinitely zoomable canvas.

In a similar approach, DeLine and Rowan [2010] presented *Code Canvas*, an IDE that replaces the box-partitioned interface with an infinitely zoomable canvas. This canvas hosts all editable documents grouped by structural concepts such as classes and dependencies. Having all of a project's code in one space is supposed to leverage spatial memory to support navigation within the project. The authors argue that eliminating hyperlink navigation from software development yields a better understanding of where certain fragments are located in the code.

Patchworks also arranges code into small boxes that are arranged on a horizontal grid.

Patchworks is an IDE that arranges code into a grid of patches that can scroll horizontally (Henley and Fleming [2014], Henley et al. [2014]). Each of the patches is a small-scale editor that holds a code fragment of different possible levels of granularity such as a method or an entire class.

In contrast to *Code Bubbles* and *Code Canvas*, the *Patchworks* interface still provides a view for the package explorer. The explorer can be used to drag and drop fragments into patches. The authors evaluated the IDE in a comparative user study to Eclipse and *Code Bubbles* and found improvements in terms of developer navigation over both alternatives. In a later study on the *Patchworks* editor, Henley et al. [2017] did not find any significant difference in success rate or task completion time between a participant group solving tasks in a tabbed editor and a group using the *Patchworks* editor. However, they observed that participants in the *Patchworks* editor made significantly fewer clicks per navigation and overall made fewer navigation mistakes.

While these ideas did not revolutionize IDE design, they show promising improvements in navigation through access to interesting code pieces.

The results of the presented research on novel IDE paradigms and their user studies suggest several insights that are interesting in regards to our own research endeavours. For one, they show that the navigation issues developers face do not have to be just accepted as a byproduct of human limitation and performance. The results promise significant improvements through the means of efficient support tooling and IDE design. And additionally, the results show that easy access to interesting code fragments, without clogging up additional screen space, is a valuable resource in supporting developers in their navigation.

2.2.2 Call-Hierarchy based Navigation Tools

Elements of the call graph are important for code navigation and comprehension.

A big area of research on developer navigation support is focused on leveraging call graph information to guide programmers along interesting relations in the code. The call graph (sometimes control-flow) describes the hierarchy of callers and callees of a callable code fragment (usually functions and methods). Along the call graph of an element one usually finds closely related code pieces and navigating it has proven to be particularly important for code comprehension (LaToza and Myers [2010]).

Ko and Myers [2004] designed the *Whyline*, an interrogative debugger. The *Whyline* allows developers to ask questions about why something happened or did not happen in the textual or graphical output of a program. The questions are asked by choosing certain code elements and properties. To answer the questions, the *Whyline* returns a slice of the program containing relevant methods and variables from the call graph that influenced the outcome in question. A conducted user study deemed the tool useful for answering *Why* questions about a program and showed that it helped users complete debugging tasks faster and more successfully. Later studies by Ko and Myers confirmed the tool's general usability as well as its potential use for real-world debugging tasks in a comparative study to breakpoint debuggers (Ko and Myers [2008, 2009]).

The *Whyline* made developers more efficient in debugging by using call graph slices to explain program outputs.

LaToza and Myers [2011] developed *Reacher*, a tool that restricts user searches to results along the call graph of a method. When evaluated in a user study, the authors found that participants performed significantly better in answering code comprehension questions when using *Reacher*.

Stacksplorer by Karrer et al. [2011] is a plug-in for the Xcode IDE that visualizes the direct call graph neighborhood of code elements. It aims to improve code comprehension and navigation, especially for unfamiliar code. For a focus method, *Stacksplorer* visualizes the callers of the method in a column to the left of the editor and the method's callees in a column to the right. The authors conducted user studies and found that software maintenance tasks on unknown code were performed much faster by participants using *Stacksplorer*. Participants also showed a heightened awareness of side effects of changes. Overall the prominent display of recommended code pieces based on structural relationships showed great promise in terms of improving developer navigation. The authors suggest that additional relationships besides call hierarchy structures could further improve the approach's beneficial impact.

Stacksplorer prominently recommends code elements from the call graph.

It helped developers perform maintenance tasks on unfamiliar code faster and more aware of side effects.

Blaze by Krämer et al. [2012] is another IDE extension that displays a path through the call graph for a method in focus. Similar to *Stacksplorer*, the call graph is visualized next to the editor but contains graph elements from a larger neighborhood. A study comparing the tool to *Stacksplorer* and standard call hierarchy implementations in IDEs confirmed earlier studies and showed that participants using

Blaze is another tool recommending call graph elements.

explicit call graph navigation tools such as *Blaze* and *Stacksplorer* were able to solve software maintenance tasks much faster than using standard tooling. In a later study, Krämer et al. [2013] investigated what exactly made tools such as *Stacksplorer* and *Blaze* successful in supporting code exploration. The results of the study suggested that the prominent display of navigation recommendations and structural information lead to the success of the tools. In contrast, typical call graph implementations of IDEs tend to hide the information behind context menus and reactive tooling.

Smith et al. [2017] developed *Flower* in an attempt to bundle control flow and data flow navigation in one tool without taking up additional screen space. *Flower* functions as an Eclipse plugin and highlights on-screen references to highlighted code elements and adds links to the top and bottom of the editor that lead to off-screen references. No additional view in the IDE is needed to display the element's references. A user study gave mixed results as users struggled to navigate along more complex call graphs.

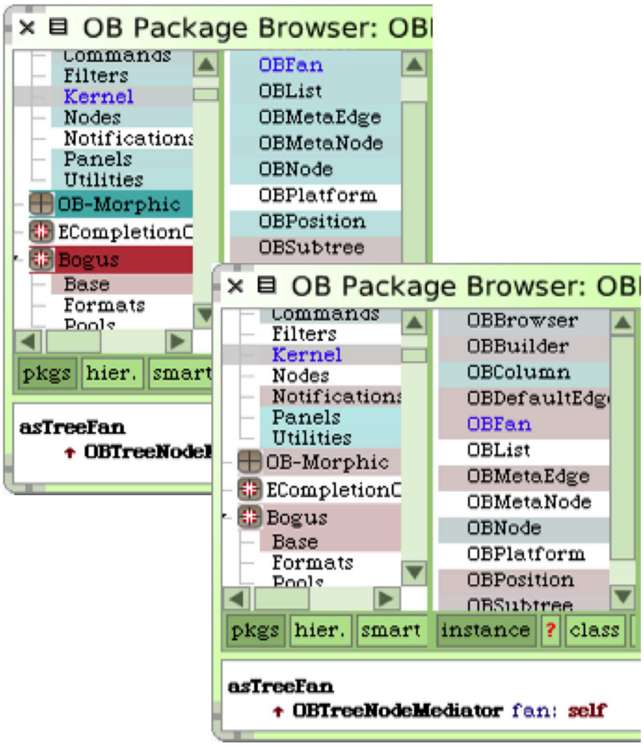
Prominent display of recommendations is important for navigation support tools.

Overall navigation support tools that recommend elements from the call graph seem to be promising approaches. Several studies confirmed improvements in code comprehension and navigation. The driving factor behind their success appears to be the prominent display of recommendations (Krämer et al. [2013]).

2.2.3 Other Navigation Support Approaches

Apart from new IDE paradigms and navigation tools that leverage structural-relationship, such as the call graph, other approaches toward navigation support have appeared over the years. This subsection will cover a few stand-alone works with original approaches before introducing another big group of developer support tools: recommendation systems in software engineering (RSSEs).

Rothlisberger et al. [2009] developed *HeatMaps*, a tool that gathers information from navigation, modification, and deletion of source code in the IDE. The gathered data can



```
asTreeFan
    ↑ OBTreeNodeMediator fan: self
```

Figure 2.1: The *HeatMaps* interface highlighting number of versions of source artifacts, top left, and recently browsed artifacts, bottom right (Source: Rothlisberger et al. [2009])

then be used to display heat maps that highlight certain software artifacts according to various metric values (as shown in Figure 2.1). The properties on which to base the heat maps can be configured by the developer to gain various different insights on interesting points in the project. Examples by authors included frequently or recently visited artifacts, the number of versions of a fragment, or the age of code pieces. Combinations of the metrics are possible as well. The heat maps work by highlighting the artifacts in colors that correspond to values on a scale. Even though no user study was conducted, the concept of introducing color highlights to visualize information on code artifacts displayed in the IDE should be interesting to pursue. Especially given the success of syntax highlighting in IDEs.

HeatMaps uses color highlights on certain elements in the IDE to guide the developer's attention.

Another approach to code navigation is through social tagging. *TagSEA* (Storey et al. [2006, 2007]) allows developers to append tags to waypoints in the code. In shared repositories, these tags are available to all developers in a team. These tags can then be navigated via keyword search. The shared collaborative knowledge of a team's annotation helped developers traverse between related artifacts.

2.3 Recommendation Systems for Software Engineering

Developer interactions can be data mined for rules of frequent behaviors and patterns.

Most of the efforts discussed so far have approached the navigation issue either by changing the way we interact with code in the IDE or by providing new ways of traversing structural relations between code elements in the IDE. Another branch of navigation support tools aims to provide smart navigation assistance by recommending potential points of interest to the developer which were learned from developer interaction behavior. Similar to how studies, such as Amann et al. [2016], Minelli et al. [2015, 2014], Gu et al. [2014], Damevski et al. [2017], have tracked developers' interaction data in order to analyze their behavior, the same kind of data can be used to mine patterns about developer activity in a code project. Histories of how programmers interact in the IDE give us detailed navigation paths that we can data mine for rules and patterns. Interaction patterns allow us to make statements like "Developers who worked on this code fragment often also changed...". These navigational patterns can reveal couplings and dependencies in the code that exceed but do not exclude structural couplings such as those from the call hierarchy. Common examples are semantically related code elements that are frequently changed together but are located in files written in different programming languages. Program analysis or topological tools fail to identify such relationships in code bases.

These rules can reveal otherwise hidden couplings between code elements.

We call these special dependencies between code elements evolutionary couplings:

EVOLUTIONARY COUPLINGS:
Evolutionary couplings are dependencies between code elements that are revealed by how a system evolves over time. Two code elements may be coupled evolutionary if they are frequently co-changed together or impacted by similar changes.

Definition: *Evolutionary Couplings*

An excursus on research on evolutionary couplings and how they can be mined from interaction histories can be read in Appendix A.

In this section, we will take a closer look at systems that recommend potential points of interest (POIs) to the developer. Most of those exploit evolutionary couplings to identify those POIs and use different techniques to mine those. We will also discuss the lack of human-centered research on these systems and suggest it as a possible reason why, despite continuous technological advances, these systems have not seen major adoptions in the landscape of modern IDEs.

We will look at RSSE approaches that use evolutionary couplings in the literature.

RSSEs are tools that support developers in seeking and finding relevant information to their current task (Robillard et al. [2010]). They support the developer by providing recommendations for certain actions or artifacts based on a context in which the recommendations are to be considered. The context can be seen as one part of the input data of RSSEs. The other part consists of the data that the recommendation engine crawls to compute the actual recommendations for the given context. While they can cover a wide range of applications such as finding reusable code, executing useful commands, or even tasks outside the IDE, we will focus on RSSEs that support developers inside the IDE during the programming task. Specifically, RSSEs that assist developers in navigating large code bases and recommending potentially interesting places in the code for further inspection or changing. Robillard et al. [2010] define three basic functionalities an RSSE architecture must implement:

We are mainly interested in RSSEs that recommend potential POIs.

There are three basic functionalities of RSSEs.

1. A data-collection mechanism to collect development-process data and artifacts in a data model;
2. A recommendation engine to analyze the data model and generate recommendations; and
3. A user interface to trigger the recommendation cycle and present its results.

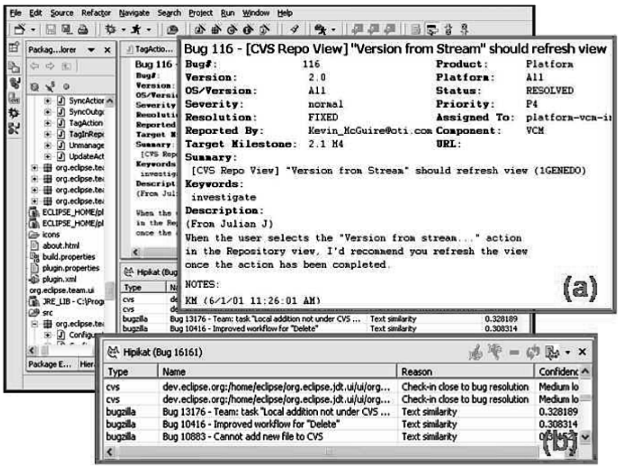
Robillard et al. also define three design dimensions of RSSEs:

There are three design dimensions of RSSEs.

- **The nature of the context:** RSSEs either require explicit context provided by the user through an interface or implicitly detect the current context for which to recommend things. Of course, a hybrid of both may be implemented.
- **The recommendation engine:** The additional data based on which recommendations are computed may originate from version histories, developer interaction data, bug reports... Typically, to produce recommendations from the data, a data mining algorithm is deployed. Finally, most RSSEs rely on a ranking mechanism to put mined recommendations into an order of relevance.
- **The output mode:** The basic possible output modes of RSSEs are push and pull modes. Pull modes provide recommendations after explicit user queries, while pull modes continuously provide recommendations proactively. Pull modes often require an implicit context.

Hipikat recommended a list of related artifacts for a current change task.

Hipikat (Cubranic and Murphy [2003]) is an early RSSE that takes artifacts of a software repository including versions, source code, bugs, related communication, and documentation as data. The tool links these artifacts based on keywords, meta information, and relationships. The artifacts are then translated into document vectors that allow for natural language querying by the user as context. Upon a query, a set of candidate artifacts is returned and


File Edit Source Refactor Navigate Search Project Run Window Help
(a)
(b)

Bug 116 - [CVS Repo View] "Version from Stream" should refresh view

```
Bug#:              116                   Product:      Platform
Version:            2.0                  Platform:     All
OS/Version:         All                  Status:       RESOLVED
Severity:           normal               Priority:     P4
Resolution:         FIXED                Assigned To:  platform-vcm-i
Reported By:        Kevin_McGuire@oti.com Component:   VCM
Target Milestone:   2.1 M4               URL:
Summary:
 [CVS Repo View] "Version from Stream" should refresh view (1GENEDO)
Keywords:
 investigate
Description:
(From Julian J)
When the user selects the "Version from stream..." action
in the Repository view, I'd recommend you refresh the view
once the action has been completed.

NOTES:
KM (6/1/01 11:26:01 AM)
```

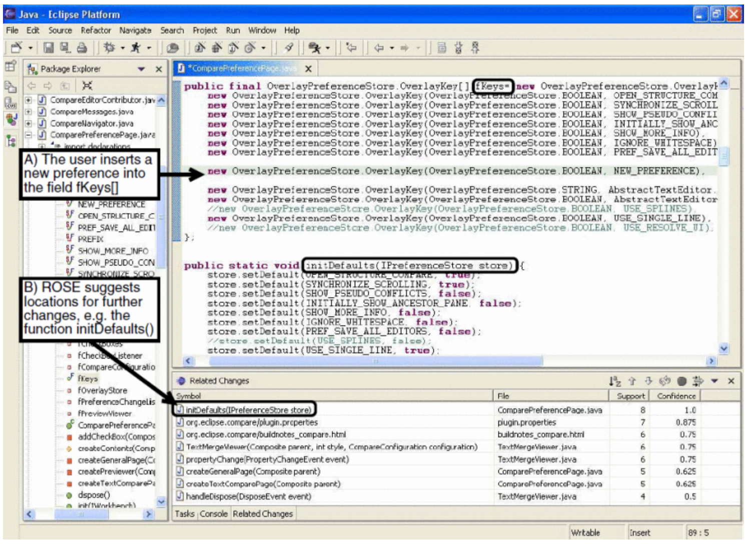
Hipikat (Bug 16161)

Type	Name	Reason	Confidenc
cvs	dev.eclipse.org:/home/eclipse/org.eclipse.jdt.ui/ui/org...	Check-in close to bug resolution	Medium lo
cvs	dev.eclipse.org:/home/eclipse/org.eclipse.jdt.ui/ui/org...	Check-in close to bug resolution	Medium lo
bugzilla	Bug 13176 - Team: task "Local addition not under CVS ...	Text similarity	0.328189
bugzilla	Bug 10416 - Improved workflow for "Delete"	Text similarity	0.308314
bugzilla	Bug 10883 - Cannot add new file to CVS	Text similarity	[illegible]

Figure 2.2: The *Hipikat* view in Eclipse, recommending different bug reports. (a) shows the current change task, (b) shows the recommended list of related artifacts. (Source: Cubranic et al. [2005])

formed into recommendations. A prototype tool implemented *Hipikat* in Eclipse, providing a list-based view for the recommendations (shown in Figure 2.2). The recommendations are enriched with reasons for the recommendation, such as text similarity or log occurrence, and can be clicked to open them in an editor. In a first evaluation study, it was found that in principle the recommendations could help developers in getting started with tasks. The effectiveness of the tool however was largely dependent on the quality and number of recommendations. A large number of recommendations was tough to filter for relevant artifacts and confused users. A later study showed that when using *Hipikat*, newcomers came closer to performing like experienced team members on software tasks (Cubranic et al. [2005]). Inspired by *Hipikat*, Malheiros et al. [2012] later created a similar tool that improved on the used textual similarity matching algorithm. The results were better recommendations and better overall performance.

Zimmermann et al. [2004] proposed a method to mine version histories and find associations between changed code


Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
*ComparePreferencePage.java
A) The user inserts a new preference into the field fKeys[]
B) ROSE suggests locations for further changes, e.g. the function initDefaults()
Writable
Insert
89 : 5

```
public final OverlayPreferenceStore.OverlayKey[] fKeys= new OverlayPreferenceStore.OverlayK
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, OPEN_STRUCTURE_COM
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SYNCHRONIZE_SCROLL
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_PSEUDO_CONFLI
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, INITIALLY_SHOW_ANC
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, SHOW_MORE_INFO),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, IGNORE_WHITESPACE)
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, PREF_SAVE_ALL_EDIT

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, NEW_PREFERENCE),

    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.STRING, AbstractTextEditor.
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, AbstractTextEditor
    //new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SPLINES),
    new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_SINGLE_LINE),
    //new OverlayPreferenceStore.OverlayKey(OverlayPreferenceStore.BOOLEAN, USE_RESOLVE_UI),
};

public static void initDefaults(IPreferenceStore store) {
    store.setDefault(OPEN_STRUCTURE_COMPARE, true);
    store.setDefault(SYNCHRONIZE_SCROLLING, true);
    store.setDefault(SHOW_PSEUDO_CONFLICTS, false);
    store.setDefault(INITIALLY_SHOW_ANCESTOR_PANE, false);
    store.setDefault(SHOW_MORE_INFO, false);
    store.setDefault(IGNORE_WHITESPACE, false);
    store.setDefault(PREF_SAVE_ALL_EDITORS, false);
    //store.setDefault(USE_SPLINES, false);
    store.setDefault(USE_SINGLE_LINE, true);
```

Related Changes

Symbol	File	Support	Confidence
initDefaults(IPreferenceStore store)	ComparePreferencePage.java	8	1.0
org.eclipse.compare/plugin.properties	plugin.properties	7	0.875
org.eclipse.compare/buildnotes_compare.html	buildnotes_compare.html	6	0.75
TextMergeViewer(Composite parent, int style, CompareConfiguration configuration)	TextMergeViewer.java	6	0.75
propertyChange(PropertyChangeEvent event)	TextMergeViewer.java	6	0.75
createGeneralPage(Composite parent)	ComparePreferencePage.java	5	0.625
createTextComparePage(Composite parent)	ComparePreferencePage.java	5	0.625
handleDispose(DisposeEvent event)	TextMergeViewer.java	4	0.5

Tasks Console Related Changes

Figure 2.3: The *ROSE* interface (Source: Zimmermann et al. [2004])

ROSE used association rules from change histories to recommend likely due changes in a list.

elements. Their approach used association rules to suggest and predict likely changes, and to detect evolutionary couplings undetectable by program analysis. They developed a tool called *ROSE* that recommends possibly due-to-change code pieces in a view in the IDE whenever a user changes some code element. For file types that supported a symbol outline (like C, C++, Java...) the tool worked on a fine-granular level, while other file types only invoked a file-level granularity of the miner. The recommendation user interface is implemented as a list view in Eclipse and is ranked by support and confidence measures of the association rules (see Figure 2.3). Besides navigation support, the goal of *ROSE* was to prevent errors or bugs from being introduced into the code by forgetting to change relevant fragments. To do so, *ROSE* implements a pre-commit warning when confidently related changes to the commit have not been made. The initial results showed promising navigational precision results in a quantitative test study using archived open-source repositories. *ROSE* correctly predicted a third of later to-be-changed items, while failing to prevent most errors. However, no user study was conducted on the tool and its IDE implementation.

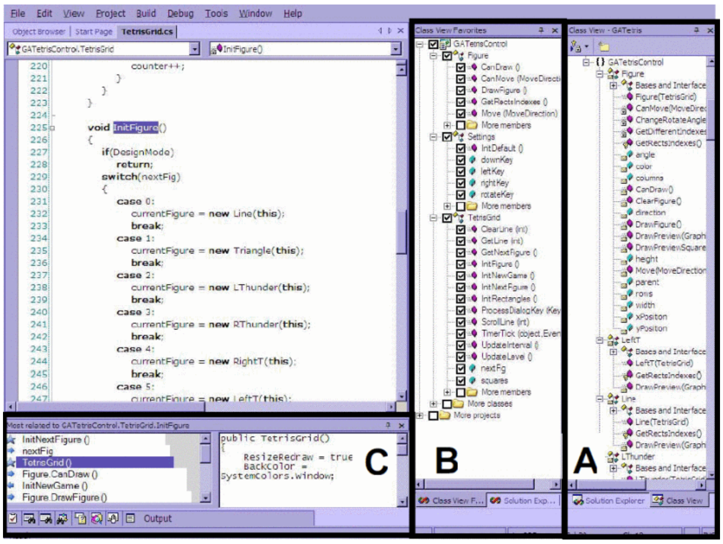
Ying et al. [2004] proposed a similar approach to Zimmermann et al. that also used CVS data to recommend source code that might be relevant to a code fragment. They used frequent pattern mining and achieved similar quantitative results to *ROSE*. However, they did not implement their approach in an IDE nor conduct any user studies.

Robillard [2005] proposed a technique to automatically propose and rank program elements that are potentially interesting to a developer to help better understand their current context or task. Their approach used the topology of structural dependency, including calls, called by, accesses, and accessed by, but did not consider change history. The tool proposed by Robillard was not proactive and required developers to drag and drop or select the input.

TeamTracks analyzes an entire team's navigation interactions and recommends items based on view frequencies.

DeLine et al. [2005b] proposed a method to track interaction history with code files and extract direct succession visits between classes and methods in that file. The view interactions are tracked for an entire team and evaluated as a whole. They then recommended "frequently visited next" elements in a list to users with frequency as the metric of relevance. The aim was to leverage the experienced team members' navigation expertise to help new team members in their tasks. The authors present *TeamTracks* (DeLine et al. [2005a]), a tool recommending the most related items based on tracked navigation data, using only view activities. The tool reorganizes the IDEs class view (see Figure 2.4 (A)) based on most frequently visited elements into a view called Class View Favorites, shown in Figure 2.4 (B). In addition, for the selected program element in the editor, the most frequently visited elements before or after visiting it are displayed in a Related Items view (see Figure 2.4 (C)) The authors conducted a usability study for the recommendation system yielding promising results and let participants use the tool for programming tasks. The results showed that the system worked especially well for new developers, and overall improved test task completion rates. However, the related items list was used far more frequently than the rearranged class view which was unpopular by comparison. An important result was that the results for code comprehension quizzes were significantly better with the usage of *TeamTracks* in the tasks. Tests on the pos-

TeamTracks significantly improved users code comprehension.



```
                counter++;
            }
        }
    }

    void InitFigure()
    {
        if(DesignMode)
            return;
        switch(nextFig)
        {
            case 0:
                currentFigure = new Line(this);
                break;
            case 1:
                currentFigure = new Triangle(this);
                break;
            case 2:
                currentFigure = new LThunder(this);
                break;
            case 3:
                currentFigure = new RThunder(this);
                break;
            case 4:
                currentFigure = new RightT(this);
                break;
            case 5:
                currentFigure = new LeftT(this);
```

Most related to GATetrisControl.TetrisGrid.InitFigure

InitNextFigure ()
nextFig
TetrisGrid ()
Figure.CanDraw ()
InitNewGame ()
Figure.DrawFigure ()

```
public TetrisGrid()
{
    ResizeRedraw = true
    BackColor =
SystemColors.Window;
```

☑ GATetrisControl
- ☑ Figure
  - ☑ CanDraw ()
  - ☑ CanMove (MoveDirecti
  - ☑ DrawFigure ()
  - ☑ GetRectsIndexes ()
  - ☑ Move (MoveDirection)
  - □ More members
- ☑ Settings
  - ☑ IntDefault ()
  - ☑ downKey
  - ☑ leftKey
  - ☑ rightKey
  - ☑ rotateKey
  - □ More members
- ☑ TetrisGrid
  - ☑ ClearLine (int)
  - ☑ GetLine (int)
  - ☑ GetNextFigure ()
  - ☑ InitFigure ()
  - ☑ InitNewGame ()
  - ☑ InitNextFigure ()
  - ☑ InitRectangles ()
  - ☑ ProcessDialogKey (Key
  - ☑ ScrollLine (int)
  - ☑ TimerTick (object,Even
  - ☑ UpdateInterval ()
  - ☑ UpdateLevel ()
  - ☑ nextFig
  - ☑ squares
  - □ More members
- □ More classes

□ More projects

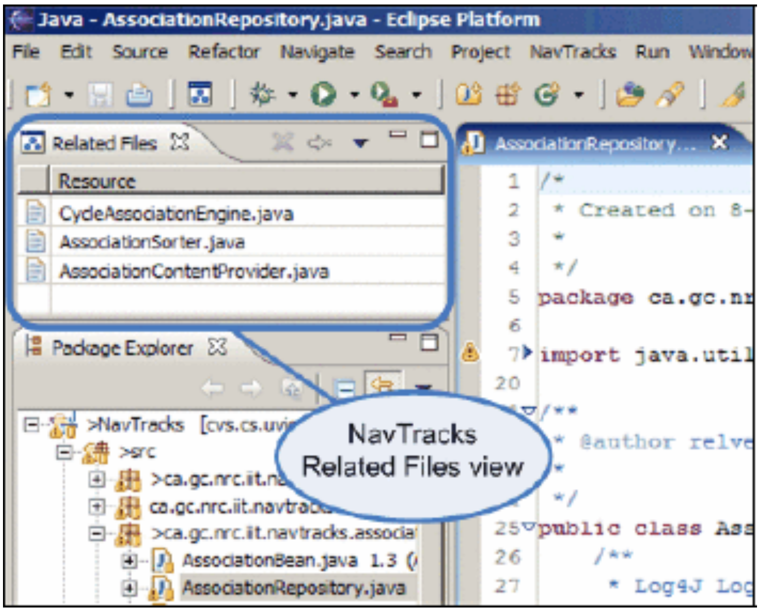
{} GATetrisControl
- Figure
  - Bases and Interface
  - Figure(TetrisGrid)
  - CanMove(MoveDirec
  - ChangeRotateAngle
  - GetDifferentIndexes
  - GetRectsIndexes()
  - angle
  - color
  - columns
  - CanDraw()
  - ClearFigure()
  - direction
  - DrawFigure()
  - DrawPreview(Graph
  - DrawPreviewSquare
  - height
  - Move(MoveDirection
  - parent
  - rows
  - width
  - xPosition
  - yPosition
- LeftT
  - Bases and Interface
  - LeftT(TetrisGrid)
  - GetRectsIndexes()
  - DrawPreview(Graph
- Line
  - Bases and Interface
  - Line(TetrisGrid)
  - GetRectsIndexes()
  - DrawPreview(Graph
- LThunder
  - Bases and Interface

Figure 2.4: The *TeamTracks* interface featuring a reorganized Class View (B) and Related Items (C) (Source: DeLine et al. [2005a])

itive effect of visual aids, such as different-sized icons for reasoning, were not conclusive. Some participants of the study remarked that they wished to have both the *TeamTracks* recommendations as well as recommendations from the selected elements call graph available in the tools views.

NavTracks used developer interaction data to create links between files based on navigation behavior from which recommendations are generated.

Singer et al. [2005] proposed *NavTracks*, which is an RSSE that keeps track of navigation histories and forms associations between related files. They argue that the hierarchical means of navigation and organization in an IDE are not the only meaningful relationship between files. Instead, similar to LaToza et al. [2006], the authors hypothesize that the paths developers take in a software project reveal their mental model of the project. They suggest that the file relationships uncovered by the navigation history should be consistent with the developer's mental model of the code. At the time, the novelty of their approach, compared to tools like *ROSE*, was that they did not use repository data for co-occurrences of changes but active developer navigation traces. In doing so, the authors hoped to gain a more up-to-date view of code dependencies as well as more navigation-directed developer support. *NavTracks* is imple-


Java - AssociationRepository.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project NavTracks Run Window
Related Files
Resource
CycleAssociationEngine.java
AssociationSorter.java
AssociationContentProvider.java
Package Explorer
>NavTracks [cvs.cs.uvi
>src
>ca.gc.nrc.iit.na
ca.gc.nrc.iit.navtrac
>ca.gc.nrc.iit.navtracks.associa
AssociationBean.java 1.3 (
AssociationRepository.java
AssociationRepository...
NavTracks
Related Files view

```
/*
 * Created on 8-
 *
 */
package ca.gc.nr

import java.util

/**
 * @author relve
 *
 */
public class Ass
    /**
     * Log4J Log
```

Figure 2.5: The *NavTracks* view (Source: Singer et al. [2005])

mented as an extension to Eclipse and meant to be complementary to the Eclipse Package Explorer. It proactively presents the top three related files to the developer's current context in a list view (seen in Figure 2.5). Similar to previous RSSE accuracy studies, the tool averaged around 35% successful next navigation predictions in a user-less study. The results of a usability study showed that it especially helped newcomers to projects by reducing their search and navigation times.

Kersten and Murphy [2005] developed *Mylar*, which is a tool that monitors programmer activity and calculates a Degree of Interest (DOI) based on selections, edits, and visits to program elements. *Mylar* uses colored shading to encode DOI levels and filters the class outline view and package explorer to show only elements of interest, hiding the rest. The remaining elements, deemed relevant enough to the current tasks are highlighted in color, based on their DOI, as shown in Figure 2.6. *Mylar* relies on the IDEs structure views to display interesting elements but does not include any markers of relationships between elements. In other words, no reasoning for an element's DOI is given.

Mylar/*Mylyn* uses color highlights on a gradient to highlight the current degree of interest for elements in the IDE.

The approach is task-based, highlighting elements relevant to the current task of the programmer, and requires task switching between tasks. A first evaluation showed that *Mylar* reduced the amount of navigation and especially misnavigation by developers during their tasks. However, no evaluation of the effects of the color highlighting itself was done. In a following paper by Kersten and Murphy [2006], a more advanced version of *Mylar* was tested, which provided professional programmers with DOI highlights in relation to the task context. Their user study suggested that providing developers with DOI highlights boosted their productivity.

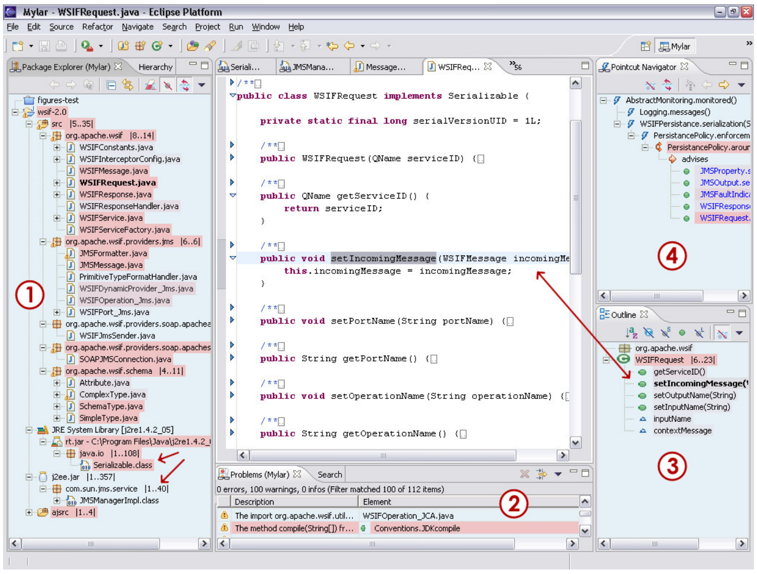
The *Mylar* tool later become known as the *Mylyn*[1] task management plugin for Eclipse that can be used to track issues or bugs. It still provides the DOI highlights of the task-focused interfaces and implements *Mylar*s original interaction tracker in Eclipse. It has been used in combination with Bugzilla[2] on a multitude of large open-source projects. As a result, it has accumulated a large collection of bug-report-related interaction data of those projects. Those interaction traces have since been used in a lot of the following works for tool validation.

Parnin and Gorg used working contexts to try and help developers return to tasks after interruptions.

Recency and frequency of visits were found to be valuable for navigation predictions.

Parnin and Gorg [2006] proposed building usage contexts during program comprehension by saving working contexts, including open windows and last executed commands, when programmers interrupt a task due to distractions or priority shifts. The context is then used as potential points of interest when a developer comes back to the task. Recommendations based on the context, using association rules, are also made and aim to reduce the amount of time developers spend recovering their last used work context. In a comparison of predictive models, they found recency of visits, and frequency of visits to artifacts to be the most valuable metrics to base predictions on. Motivated by this comparison of models, Piorkowski et al. [2011] further investigated the accuracy of certain prediction models for predicting programmer navigation. They too found that recency of navigation yields accurate navigation models, however only for click-based navigation – and on par

[1] https://www.eclipse.org/mylyn/ (accessed on May 11th, 2023)
[2] https://www.bugzilla.org (accessed on May 11th, 2023)



```
/**
public class WSIFRequest implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
    public WSIFRequest(QName serviceID) {

    /**
    public QName getServiceID() {
        return serviceID;
    }

    /**
    public void setIncomingMessage(WSIFMessage incomingMe
        this.incomingMessage = incomingMessage;
    }

    /**
    public void setPortName(String portName) {

    /**
    public String getPortName() {

    /**
    public void setOperationName(String operationName) {

    /**
    public String getOperationName() {
```

Problems (Mylar) Search

0 errors, 100 warnings, 0 infos (Filter matched 100 of 112 items)

Description	Element
The import org.apache.wsif.util...	WSIFOperation_JCA.java
The method compile(String[]) fr...	Conventions.JDKcompile

Figure 2.6: The *Mylar* views in the Eclipse IDE (Source: Kersten and Murphy [2005])

with code topology methods. For navigation within views, they found within-file distance of methods to be the most accurate metric for building navigational predictions. Interestingly, the study found Bug Report Similarity, even though commonly used in RSSEs up to that point in time, to perform significantly worse than Recency, Within-File Distance, topology, and mixtures of those.

At this point in the history of RSSE research, we see a decline in the actual implementation of tools and human-centered evaluation of approaches. Instead, the focus shifts towards more technical research and advances in recommendation engines, context data, and input data.

Few modern RSSE approaches have seen actual implementation or interface research.

So far, most of the approaches we have discussed collected data either from version histories or purely navigational interaction (or view-based) data of developers. The developer though, interacts more richly with the IDE than just through visits to certain files or code elements. The most basic interaction with source code is the edit. Edit interactions with artifacts provide another valuable information source that can be used to mine regularities from developer

Only recently have recommendation engines started considering edit interactions.

activity. Just as edits found in commits of version histories can be used to uncover evolutionary couplings, so can edits found in interaction data. Given that interaction data provides a vastly larger amount of recorded activities than version histories, fully leveraging the developer's interaction history seems promising to improve navigation support. Whereas the tools discussed so far often featured visual implementations but low recommendation accuracy, the newer approaches in the following reach much higher accuracy but fail to account for the usability measures of these systems.

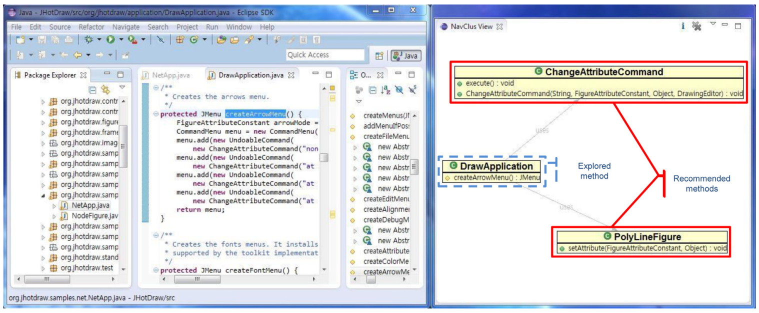
Interlude: Mylyn interaction data

It should be noted that most of the following works used *Mylyn* interaction data to evaluate their approaches. There exists research that suggests that *Mylyn* traces contain significant amounts of false edit events. As a result, the following accuracy measures could be lower than they should be as improvements are to be expected when evaluating with clean data. A more detailed interlude can be found in Appendix B.

Kobayashi et al. [2012] track read and write accesses to code artifacts on file and method level. Using the interaction data they create change sequences that are translated into a directed change guide graph.

NavClus tracks interaction histories and forms implicit task contexts. These contexts are later used to recommend POIs.

NavClus was developed by Lee and Kang [2013] to improve the accuracy the state-of-the-art navigation recommendation systems that mine programmers interaction histories. The tool segments interaction histories into sequences that are likely to be related to a task the programmer pursued. Program elements that are interacted with within one sequence are likely to be related to each other. The obtained sequences are then clustered by similarity to form task-related contexts. These contexts can later be recovered to predict interesting points in the code based on task similarity. An important advance over earlier task-related recommenders is that a task is implicitly defined by the features modified by the developer. No explicit task description or bug report is required. The tool was compared to *TeamTracks* (DeLine et al. [2005a,b]) in a study that evaluated both tools on archived bug reports and debug interaction data. The study showed that *NavClus* was able to



```
/**
 * Creates the arrows menu.
 */
protected JMenu createArrowMenu() {
    FigureAttributeConstant arrowMode =
    CommandMenu menu = new CommandMenu(
    menu.add(new UndoableCommand(
        new ChangeAttributeCommand("non
    menu.add(new UndoableCommand(
        new ChangeAttributeCommand("at
    menu.add(new UndoableCommand(
        new ChangeAttributeCommand("at
    menu.add(new UndoableCommand(
        new ChangeAttributeCommand("at
    return menu;
}

/**
 * Creates the fonts menus. It installs
 * supported by the toolkit implementat
 */
protected JMenu createFontMenu() {
```

Java - JHotDraw/src/org/jhotdraw/application/DrawApplication.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer
NetApp.java
DrawApplication.java
NavClus View
ChangeAttributeCommand
execute() : void
ChangeAttributeCommand(String, FigureAttributeConstant, Object, DrawingEditor) : void
uses
DrawApplication
createArrowMenu() : JMenu
Explored method
Recommended methods
uses
PolyLineFigure
setAttribute(FigureAttributeConstant, Object) : void
org.jhotdraw.samples.net.NetApp.java - JHotDraw/src

Figure 2.7: The *NavClus* tool and its UML navigation view (Source: Lee et al. [2013])

produce higher recommendation accuracy than *TeamTracks*. While the precision only increased from 49% to 52%, the average F-measure value for *TeamTracks* recommendations was 0.073 while *NavClus* achieved a value of 0.144. Furthermore, *NavClus* yielded a much higher edit-hit ratio, meaning that its recommendations included significantly higher percentages of later edited program elements. A graphical tool for *NavClus* was developed that outputs the recommended program elements in a UML class diagram (see Figure 2.7). The class diagram draws all previously navigated to program elements as well as recommendations, that can then be inspected. Later, a user study was conducted by Lee and Kang [2016] that aimed more towards identifying use cases of diagramming tools in software development rather than the efficiency or usability of the recommendation tool itself.

The *NavClus* recommendations were visualized in a UML diagram.

To answer the question whether view histories or edit histories are better for mining code recommendations, Lee et al. [2015] developed *MI*. *MI* works similar to *ROSE*, but instead of only utilizing edit histories (from commits) like *ROSE*, *MI* is able to consider both the view- and the edit histories of developers. The authors hypothesize that a combined context of both interaction types is the key to improving the recommendation accuracy of RSSEs. *MI* mines association rules from the contexts that are then later used to form the recommendations. To gain detailed insight into how impactful the different interaction events are on the recommendation accuracy, Lee et al. implement differ-

MI considers both edit and view interactions in the recommendation engine.

ent versions of *MI*. The versions differ in how strict they are in the matching of current contexts to the mined rules and in when they produce recommendations. In a quantitative evaluation, similar to the one performed by Zimmermann et al. [2004] on *ROSE*, they used the *MI* versions on archived open source repositories. Specifically, *MI* was trained and evaluated on *Mylyn* data. The results showed that combining view and edit interactions yields consistently higher recommendation accuracy on file-level than only considering edit or view activities. *MI* recommends files to edit with a 63% accuracy, a significant improvement over *ROSE* with only 35%. The authors argue that breaking the 50% precision barrier is a reason to assume that RSSE tools may start to become viable for real-world use. The tool has not been implemented in an IDE nor has it been evaluated in a user study.

MI showed significant recommendation accuracy improvements over state-of-the-art approaches.

To further improve the accuracy of association rule based RSSEs, Rolfsnes et al. [2016] aggregate association rules into hyper rules. Hyper rules use evidence of multiple available rules to find better-suited recommendation candidates. Later evaluations by Rolfsnes et al. [2018] of aggregated association rules for artifact recommendation suggest that they can indeed provide recommendations with higher precision and increase the viability of existing RSSE techniques for real-world use cases.

Damevski et al. [2018] used the concept of topic models from natural language processing to model developer interaction data. Inspired by topic models' ability to abstract low-level data to higher-level concepts, the authors aimed to predict future IDE command usages relevant to the current task. While command recommendation is fundamentally different from POI recommendation, the work showed that NLP techniques could successfully be used to model the current context of developers based on their interaction data. An evaluation showed that the modeled contexts provided high-accuracy recommendations for the next set of interactions performed by developers.

Most recently, Lee et al. [2021] proposed a code edit recommendation method using a recurrent neural network (*CERNN*). Similarly to Damevski et al. [2018], the approach takes a step away from association rule mining and towards the modern techniques of machine learning. *CERNN* aims to improve on the authors' previous tool, *MI* (Lee et al. [2015]), by transforming the context into vector embeddings and using those as input for an RNN. The network is trained on interaction data by using a sliding window approach, with a set of past interactions as training input and the next developer interaction as a desired output label. The tool was evaluated in the same manner as *MI* was, on *Mylyn* data. The comparative study showed that *CERNN* was not able to produce higher recommendation accuracy than *MI*, a result the authors attributed to the large amount of recommendations produced by *CERNN*. When analyzing a version of *CERNN* that stopped producing recommendations in the simulation if the first recommendation was false, the model performed slightly better (increase in f-measure of 5%) than *MI*. It is debatable, however, how valid of an evaluation this represents since it rules out the fact that the model generates lots of bad recommendations. In real-world applications, it is not possible to tell whether a first recommendation was bad and therefore it could be argued that the *CERNN* model provides no real improvement for actual implementations over *MI*. Additionally, given the high execution times of RNNs, they are less applicable to real-time online learning applications than association rules. For one evaluated open source project, *CERNN* took over 45 days to build the model. Obviously, this rules out any real-time learning and improvement of the model. Especially, since the authors found the model to perform worse when performing training incrementally.

CERNN trains an RNN with vectorized interaction data to predict developer navigation.

Though the accuracy of *CERNN* is promising, the training times of the network make it less viable in practice.

Despite the significant improvements in the technical approaches in terms of recommendation accuracy, none of the more recent tools have seen actual implementations in IDEs or user studies on their effectiveness or usability. We will take a look at the implications in the next section.

2.4 Summary and Takeaways

Studies on developer behavior and how software engineers use IDEs have shown that navigation and program comprehension make up the two most time-consuming tasks in the day-to-day work of developers (Ko et al. [2006]). Navigation and comprehension are not distinct from one another.

Code navigation and comprehension are closely intertwined concepts and represent two of the biggest issues developers face.

Research shows that the two are closely linked and improving one can significantly impact the other (LaToza et al. [2006], LaToza and Myers [2010]). A good understanding of the inner workings of a project enables the developer to efficiently navigate between code elements. Vice versa having a good sense of where things are in a project and how to traverse along their relationships is fundamental to understanding how the code functions. Given that the two biggest bottlenecks of developer activity appear to be of the same nature, it is not far-fetched to assume that they can be tackled with a single solution.

There have been promising approaches to support developer navigation, however, they have not seen adoption in IDEs.

The standard work environment for developers is the IDE. The modern IDE provides lots of developing support but still results in the identified navigation and comprehension issues. Tools that aim to assist developers in code navigation and comprehension, therefore try to change the file-based IDE paradigm altogether or enhance the IDEs interface. Among the most successful navigation support tools are those that leverage the structural information among code elements found in the call hierarchy. Tools like *Stacksplorer* (Karrer et al. [2011]) and *Blaze* (Krämer et al. [2012]) have shown to significantly improve developer efficiency by prominently displaying available call graph information based on the current context. Research also found promoting call hierarchy information to improve code comprehension (LaToza and Myers [2011]). A study on call hierarchy visualization tools suggests that the prominent display of navigation recommendations and structural information is what enables the approaches to improve developer efficiency (Krämer et al. [2013]). However, modern IDEs do not integrate these insights into their standard interfaces. Instead most call hierarchy visualization tools have to explicitly be triggered without proactive context-based usage options.

Another promising, but fundamentally different, approach we have discussed is that of navigational RSSEs. These systems utilize version histories or developer interaction data to mine regular patterns and turn these into recommendations based on the current context. Data-driven approaches can reveal what is called *evolutionary couplings* in code that are hidden from program analysis. While early works on the topic still struggled to generate accurate recommendations, they developed actual implementations of their tools in IDEs. Early user studies showed promising usability results and benefits, especially for newcomers to software projects (DeLine et al. [2005a], Cubranic et al. [2005]). More recent approaches by Lee et al. [2015, 2021], Damevski et al. [2018] have seen significant improvements in the recommendation accuracy, but have not provided actual implementations of the proposed tools for IDEs. As a result, using RSSEs for navigation support has reached a technical level of viability but close to no human-centered research has been conducted. The higher theoretical recommendation accuracy is one part, but research still remains to be done on how and if these techniques can be implemented into tools that efficiently support developers in their code navigation and comprehension.

RSSEs have seen significant improvements on the technical side but little to no implementation, adoption, and HCI research.

In the next chapter, we will outline a conceptual approach to an IDE extension that is based on the insights from our literature review and will help us answer some of the remaining open design questions of navigational RSSEs.

Chapter 3

Concept and Design Choices

Based on the insights from the previous chapters we conceptualize a recommendation framework.

This chapter will present the conceptual approach of our developed extension as well as the choices that were made in the design process. In order to meet the goals defined in Section 1.3 a closer look at the key takeaways from the motivation Section 1.1 and the previous work presented in the previous Chapter 2 was required. We identified a promising approach to tackle the arising code navigation challenges, namely recommendation systems for software engineering. The technical advancements in the field have made these tools viable but a lack of user-centered integration into developers' work environments may have prevented them from seeing practical use so far. Research on call-graph navigation tools however has shown what can make code navigation tools efficient: prominent and proactive display of the information.

Our approach, therefore, focuses on providing a framework for state-of-the-art recommendation engines to be efficiently integrated into modern IDEs.

There are no clear guidelines on the interface design of RSSEs.

Research has seen surveys on the collection and processing of interaction data in RSSEs (Maalej et al. [2014]), surveys on context extraction in RSSEs (Maki et al. [2015]), and even some summarizing overview of recommendation output and presentation possibilities (Proksch et al. [2015], Gašparič and Janes [2015]). However, no clear guidelines have been formulated on the interface and presentation design questions of RSSEs. As Proksch et al. [2015] summarize, key design questions for the output of RSSEs are the mode of interaction (*proactive*, *reactive*), style of presentation, reason for recommendation, content of recommendation, and finally the integration of the system into a given toolset such as the IDE.

We try to answer these three main questions.

In line with Robillard et al. [2010], our design process can be divided into three major questions that needed to be addressed before any actual implementation could take place.

1. **What** to recommend?
2. **Where** to display the recommendations?
3. **How** to present the recommendations?

The following sections will discuss these questions in detail and make points for the individual design choices that were made in the process.

3.1 What to Recommend to the Developer?

We want to recommend interesting source code elements and files (artifacts).

In an in-depth survey on RSSEs by Gašparič and Janes [2015], the authors scan existing literature and classify the proposed RSSE tools into categories based on what they recommend to the developer. The different outputs range from links to helpful web resources over useful APIs to bug reports and many more. By far the most popular type of recommendation however is software artifacts: source code elements and files. Given that we aim to build an extension that aids developers in their code navigation and comprehension, it is clear that our desired type of output for recommendations should be references to other points in the code the user can navigate to: code artifacts.

To aid developers in their navigation through code projects, we want to provide them with dynamic, proactive recommendations of potential points of interest (POIs). In order for any kind of recommendations to be dynamic and proactive they need to be situation- or context-dependent (implicit). An e-commerce shop could always recommend its top-selling items to all customers but that would be far from a dynamic recommendation system. A RSSE that always recommends the most frequently visited or edited sections of a project might be helpful for new developers to identify just those sections, but would hardly be of any navigational help in specific maintenance or debugging tasks. Instead, POIs should be recommended to the developer based on her current work context, as we have seen in most of the related approaches in Section 2.3. A context typically describes the task a developer is currently working on and can be described by the set of artifacts she has recently worked on. Based on these, potentially interesting points in the code are other code artifacts. Code artifacts can be described on different levels of granularity. The highest level being a file or document itself. Most file types relevant to programming and developer work however expose a certain hierarchy and outline of contained blocks or symbols. In typical code file formats such as scripts and documents, we can identify classes, interfaces, methods, functions, variables, and many more. These, as a context, form the most common input type of RSSEs, especially for those assisting in navigation (Gašparič and Janes [2015]).

We want recommendations to be generated proactively. This means an implicit context should be formed.

Recommendations should, if possible, refer to code elements on the method level.

In our scenario, contexts are responsible for two important fundamental aspects of RSSEs. First, they are the input data that trigger recommendations. And second, contexts are tracked and logged to form the interaction history of developers. These interaction histories can then be mined using different techniques for regularities in developer activity within the code. As we are primarily interested in a conceptual framework to be used with – theoretically – any navigational recommendation engine that outputs related code artifacts, we do not decide on any particular mining technique, as that should be irrelevant to the framework. What we are most interested in is the output of recommendation engines. The output we are dealing with is mostly evolutionary couplings that resulted from matching the ob-

The framework does not specify a specific recommendation engine. Rather, it should be compatible with any recommendation engine that runs with interaction data as input.

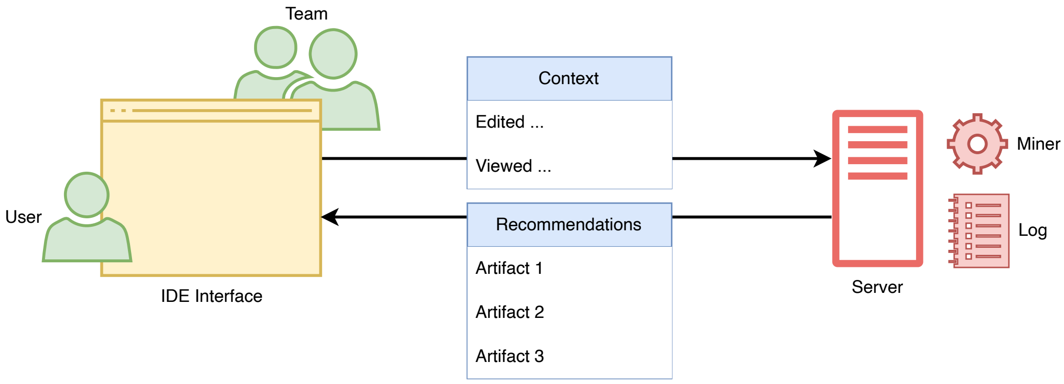


Figure 3.1: The basic cycle of an RSSE recommending software artifacts based on the developer's current context. The current context of the developer is detected in the IDE, packaged, and sent to the server (recommendation engine) that matches the context against mined rules and past observations. The resulting set of potentially interesting artifacts is returned to the developer.

served user context against a set of observed interaction behaviors from the past. The output of those engines is code artifacts of file-level or symbol-level granularity.

As newcomer developers are at the center of our thesis goals, collaborative insights from whole teams should be used.

The framework should allow for organization-wide collaborative use.

In order for the system to benefit newcomers to existing development teams, developers working on unfamiliar open source code, and new developers altogether, it should leverage the insights gained from multiple people interacting with a project and distribute the gained insights to everyone who works on it. Inspired by *TeamTracks* (DeLine et al. [2005a]), this can be achieved by tracking developer interactions locally and aggregating the interaction data on a collaborative server or repository. For open-source projects, logged interaction histories could be committed along with the source code changes to the project repository. In closed-source operations, like company intern projects or client-based work, the benefits of such a system could still be used by managing the repository's privacy settings, or by deploying an organization-wide server that centrally logs all development activity in real-time and also hosts the data miner that utilizes the logged information. The same server could then serve recommendations for a given context as input.

To achieve the collaborative goals, our framework should feature an activity tracker that works on a fine-grained method-level. To support different confidentiality restrictions of collaboration and data storage, the framework should allow for modular integration of a server architecture that interaction data can be pushed to and recommendations can be requested from. A conceptual architecture diagram of the system can be seen in Figure 3.2.

3.1.1 Merging Evolutionary and Structural Recommendations

We want to merge recommendations from the call graph with evolutionary coupling recommendations.

The approach so far takes a developer's current context to recommend related code pieces that were learned from other people working on the same project. In other words, it leverages evolutionary couplings to recommend code elements that have frequently been changed or visited in similar contexts in the past. These discovered dependencies may sometimes include structurally related elements, however, their main goal is to uncover the very opposite: dependencies that are not visible in the topological structure of code and therefore not detectable by program analysis tools. Navigational and exploratory support should not only include points in the code that are often changed together but also other structurally or semantically related artifacts. Especially for efficient code comprehension, it is important to gain a good overview of control-flow, data-flow and generally the complete neighborhood of related code elements. In the study on *TeamTracks* conducted by DeLine et al. [2005a], some participants of the study remarked that they wished to have both the TeamTracks recommendations as well as recommendations from the selected elements call graph in the recommendation view.

Studies on RSSEs and call graph tools have independently shown that including other types of couplings could be valuable.

To support developers in their code navigation we want to provide them the most complete set of POIs. This should include a merged set of structural information and evolutionary/collaborative recommendations. Studies on call graph navigation tools such as *Stacksplorer* (Karrer et al. [2011]) and *Blaze* (Krämer et al. [2012]) have already proven to improve developer efficiency in maintenance tasks and

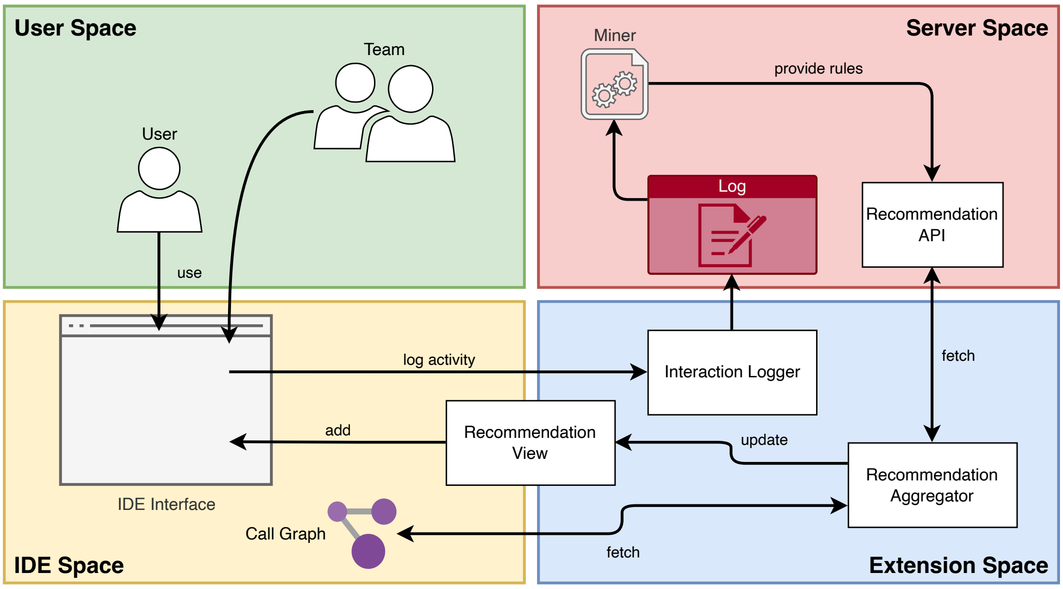


Figure 3.2: The conceptual architecture of the recommendation framework. The extension logs a team's interaction histories that are stored and mined on a server. The extension contributes a recommendation view to the IDE that is fed with data from the Recommendation Aggregator. The aggregator fetches, merges, and ranks recommendations from both the call graph and the data miner on the server.

to reduce the navigation overhead involved in code comprehension. Karrer et al. [2011] also suggest that visualizing and recommending relationships besides call hierarchy structures could further improve the approaches' beneficial impact. We hypothesize that the merging of structural and evolutionary recommendations could lead to a more complete picture of a context's related elements. In order to maximize developer support for code exploration and navigation we want to implement such a combined approach into our framework (see Figure 3.2).

To utilize both evolutionary and structural couplings for our recommendations we define the granularity of our evolutionary recommendations and the interaction logger to also be on method-level rather than file-level. By having both types of recommendations at a symbol-level in the code, we can later merge and rank them together in our recommendation interface. We will discuss this later in Section 3.3

In conclusion, we plan on providing both structural recommendations from the call hierarchy as well as evolutionary couplings from a recommendation engine in the same interface. As both approaches have shown promising results in supporting developer navigation, we aim to increase both approaches' applicability by merging them.

3.2 Where? A Home for the Recommendations

The framework should be integrated into an IDE.

This second conceptual question might seem like an obvious one. However, it entails more than one realizes at first. Developers spend approximately one third of their time working in Integrated Development Environments (IDEs), making it their most used application in a workday (Sillitti et al. [2012]). A study[1] from 2019 found out that – unsurprisingly – only 3% of developers do not use any form of IDE at all. As we can see, the general target environment to display recommendations to developers in is undoubtedly the IDE.

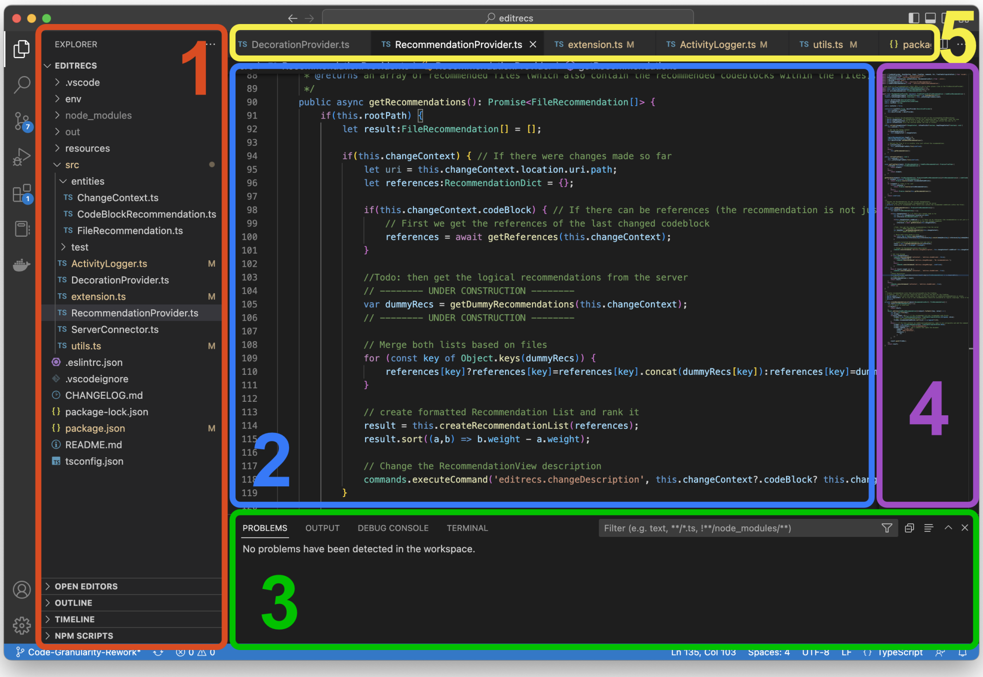
We decided on VS Code as the second biggest IDE and on account of its extension API.

Among the multitude of available IDEs, Visual Studio Code (VS Code) stands out as the fastest-growing and second most popular one [2]. VS Code is a code editor that is especially popular for usage with interpreted languages such as JavaScript, TypeScript, Python, and many more. Through the use of extensions, VS Code can be customized both in its looks and functionalities. Extensions for most languages and runtimes can be found in its big extension marketplace. For plugin developers, VS Code provides an extension API for the development of custom extensions.

The VS Code API allows us to do three important things. It allows us to track the developer's activity in the IDE, contribute new views to the IDE's GUI, and modify the decoration of file-related information in existing UI elements such as the explorer or the tab view. Based on the popular-

[1] https://www.jetbrains.com/lp/devecosystem-2019/ (accessed on May 12th, 2023

[2] https://pypl.github.io/IDE.html



```
     * @returns an array of recommended files (which also contain the recommended codeblocks within the files
     */
    public async getRecommendations(): Promise<FileRecommendation[]> {
        if(this.rootPath) {
            let result:FileRecommendation[] = [];

            if(this.changeContext) { // If there were changes made so far
                let uri = this.changeContext.location.uri.path;
                let references:RecommendationDict = {};

                if(this.changeContext.codeBlock) { // If there can be references (the recommendation is not ju
                    // First we get the references of the last changed codeblock
                    references = await getReferences(this.changeContext);
                }

                //Todo: then get the logical recommendations from the server
                // -------- UNDER CONSTRUCTION --------
                var dummyRecs = getDummyRecommendations(this.changeContext);
                // -------- UNDER CONSTRUCTION --------

                // Merge both lists based on files
                for (const key of Object.keys(dummyRecs)) {
                    references[key]?references[key]=references[key].concat(dummyRecs[key]):references[key]=dum
                }

                // create formatted Recommendation List and rank it
                result = this.createRecommendationList(references);
                result.sort((a,b) => b.weight - a.weight);

                // Change the RecommendationView description
                commands.executeCommand('editrecs.changeDescription', this.changeContext?.codeBlock? this.chan
            }
```

No problems have been detected in the workspace.

Figure 3.3: The Visual Studio Code User Interface. **1:** Primary Sidebar / File Explorer; **2:** Active Editor; **3:** Panel / Terminal; **4:** Secondary Sidebar / Minimap; **5:** Tab Bar

ity and extensibility of the IDE, we decided on VS Code as a suitable host environment for our framework tool.

The more pressing question however is where to display the recommendations *within* the IDE. In terms of the general UI layout, most of the popular IDEs do not differ too far from one another.

VS Code provides 3 general containers for custom views to be added to.

Figure 3.3 shows the general layout of VS Codes user interface. Marked by numbers and colors are the more prominent subparts or views of the UI. **1** marks the Primary Sidebar, the active View Container, and its child views. Available View Containers can be activated using the respective Icons in the Activity Bar to the left of the marked Container. View Containers are parent views in which views can be rendered. The active View Container in the figure is the File Explorer. The visible child views are the explorer itself and four more collapsed views: Open Editors, Outline, Timeline, and NPM Scripts.

2 marks the active editor, while **5** marks the open editors in the tab bar. As you can see, files that have any form of decoration – filename text color and badges: *FileDecorations* – display these decorations both in the file explorer **1**, in their respective tab in the tab bar **5**, and generally everyplace throughout the UI where the file is referenced including custom views.

3 shows the panel, typically hosting the terminal, debugging console, and other command-line-related tools.

Lastly, **4** marks the file minimap and – if extended – the area of the secondary sidebar. The secondary sidebar can be extended to host more rendered views in a similar fashion to the primary sidebar **1**.

1,3,4 are the locations in which custom views, contributed by extensions, can be rendered. A view can be used to display list-based data ranging from flat lists to deep, structured trees. Additionally, a view can provide view-actions in the form of buttons to trigger commands (see Figure 3.4).

Views can display welcome information, tree-structured data, and fully customized web content.

Custom views contributed by extensions can be initially placed in any of those locations. Views are not however fixed to their initial placement and can be dragged and dropped across the UI to any of the valid view containers **1,3,4**. There are three different types of content that can be displayed in a view. 1) A Welcome View that hosts information before or after data has been displayed in the view. As the name suggests Welcome Views are useful to present initial messages such as first steps or links to helpful resources after an extension has started up. 2) Tree Views as the standard way to display data in VS Code views. Their functionality ranges from shallow simple lists to deep trees and can be used to present hierarchical information like containment relationships among the displayed data. 3) Web Views that can be fully customized using HTML. These should only be used when the Tree View functionality is too limited to accommodate for the desired functionality. Typically this includes advanced web interactions, forms,

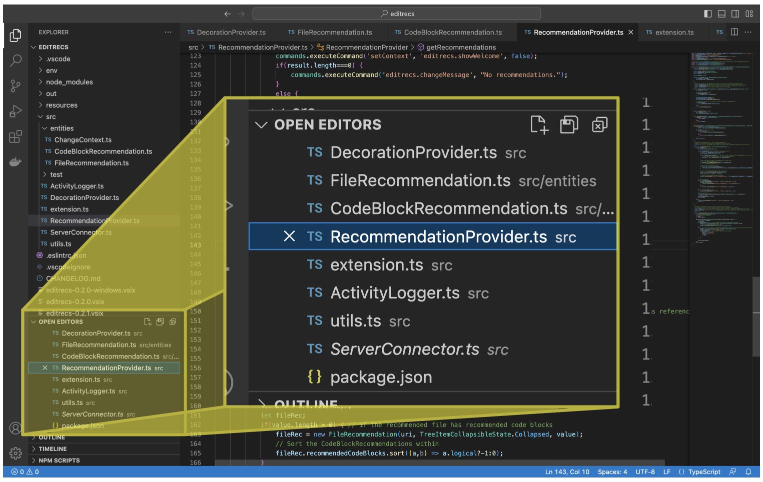

OPEN EDITORS
TS DecorationProvider.ts src
TS FileRecommendation.ts src/entities
TS CodeBlockRecommendation.ts src/...
TS RecommendationProvider.ts src
TS extension.ts src
TS ActivityLogger.ts src
TS utils.ts src
TS ServerConnector.ts src
{} package.json
OUTLINE

Figure 3.4: The Open Editors View in the VS Code interface. The view is rendered in the Primary Sidebar below the File Explorer. It can be collapsed and offers view-actions at the top.

authorization protocols, and other functionality that relies on more than just the display of text-based data.

Tree views are ideal for displaying recommendations both on file and method level.

As our approach for the extension does not feature extensive user input – as that would defeat the goal of proactiveness – or the presentation of hyper-complex or image-based data the usage of Tree Views will suffice. As discussed in the last section, the type of content we will be recommending to the user is code artifacts within the project. The Tree View will allow us to display file-level recommended items at the top of the tree with the specific recommended code blocks within that file displayed as children of that tree node. Additionally, a welcome view can be used to display initial information, useful links to documentation, and setting landings after the extension has started up. Welcome views are replaced with the actual tree view contributed by extensions as soon as the tree view has content ready for rendering.

Our tool should be customizable at least in terms of server usage as well as tracking behavior. Visual Studio Code has an extensive settings environment to which extension-specific configurations can be added. This will allow us not only to offer project-based settings but also to design the system in the modular fashion discussed in Section 3.1.

Our framework should enhance, not replace the file explorer.

By contributing a new view to the existing VS Code File Explorer UI instead of contributing a whole new View Container, we plan on enhancing the explorer instead of replacing it. This will be consistent with the design principle set by Proksch et al. [2015], that new RSSE tools should be seamlessly integrated into the existing work environments of the developer. The recommendation view will be the only additional screen space taken up by the extension.

3.3 How? Presenting Recommendations

So far our concept has covered what type of content our framework should recommend to the developer (code artifacts from collaborative recommendations merged with call graph neighbors) and where it should be situated (within the VS Code IDE as an extension providing a new Tree View). In this section, we will cover the design questions of how exactly the recommendations should be presented to the developer and when.

3.3.1 Design Choice: Proactive

Recommendations should be generated proactively with minimal user input.

Robillard et al. [2010] and Proksch et al. [2015] agree that a major design dimension of RSSEs is whether the output mode is proactive or reactive. An RSSE is reactive if it requires user input to generate a set of recommendations. This may include formulating queries or making a certain call to trigger the recommendation engine. A proactive RSSE serves recommendations automatically based on the implicit current context. The developer is not required any further action of his own. Hybrid implementations are also possible, where recommendations are made proactively for the current context but can be reactively generated for other contexts. Gašparič and Janes [2015] found the large majority of developed RSSEs to be reactive. They suggest that the reason might be the ease of implementation of reactive systems and the reduced danger of the recommendations appearing as spam.

Proactive systems have support from the literature.

However, both literature surveys (Robillard et al. [2010], Gašparič and Janes [2015]) call for more proactive systems and suspect more value and usability behind that design choice. We are inspired by the successes of call graph navigation support tools like *Stacksplorer* and *Blaze*, both of which proactively visualized the recommended information to the developer. As a study by Krämer et al. [2013] showed, the prominent display of the information was indeed what made the tools more successful in comparison to similar tools that acted reactively or hid their functionality behind calls and menus.

We as well suspect proactiveness to be a key design choice in an effective navigation support system. Especially for scenarios in which newcomers to a software system want to use the system for first steps and tasks. A prerequisite for choosing our system to be proactive is that its integration into the IDE has to be seamless and subtle so as to not appear distracting or overwhelming. As the planned view of the extension utilizes the familiar API of the IDE without providing new Web Views or external interfaces, the integration into the IDE should prove to satisfy those conditions. Additionally, the APIs Tree View is used to display textual data which is minimally obstructive when integrated into existing Container Views.

For our approach, being proactive means that the interface always automatically displays recommendations for the developer's current context, unprompted.

3.3.2 Design Choice: Tree View

Going beyond flat lists as the output mode is important to us.

Simple List-based outputs are by far the most common form of representation in RSSEs (Gašparič and Janes [2015]). However, given the lack of HCI research around the output of RSSEs and the simplistic interface implementations of existing tools (compare Figures 2.2, 2.3, 2.4, 2.5) the state-of-the-art is not necessarily the way to go forward.

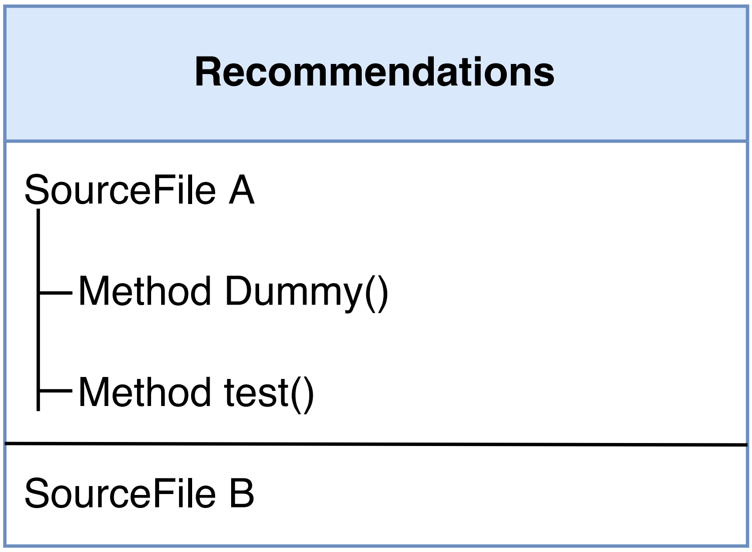


Figure 3.5: A conceptual illustration of how a tree view can be used to aggregate recommended code artifacts by documents.

Instead, it might be part of the reason why navigational support tools have not seen the adaption into modern IDE interfaces that their capabilities would suggest. Gašparič and Janes [2015] suggest that more effort in research on other forms of representation could be beneficial.

Tree Views enable the aggregation of multiple method-level recommendations from different sources in the same file.
Clicking recommended elements should take the user to them.

Instead of compiling all generated recommendations in a flat list, we plan on utilizing the capabilities of VS Codes Tree Views. The hierarchical nature of Tree Views allows us to aggregate recommendations of different kinds by source document (see Figure 3.5). The top level of the tree will be documents in the project that contain any recommendations while the nodes children represent the specific code blocks that are being recommended. Clicking on any of the tree elements opens up the recommended code block (or file if no tree node children exist) in the editor of the IDE in a new tab. This preserves the currently active editor in a tab to be navigated back to. Elements in the tree view can be collapsed for a better overview of the displayed contents.

Reasoning

RSSE implementations lack reasoning behind their output.

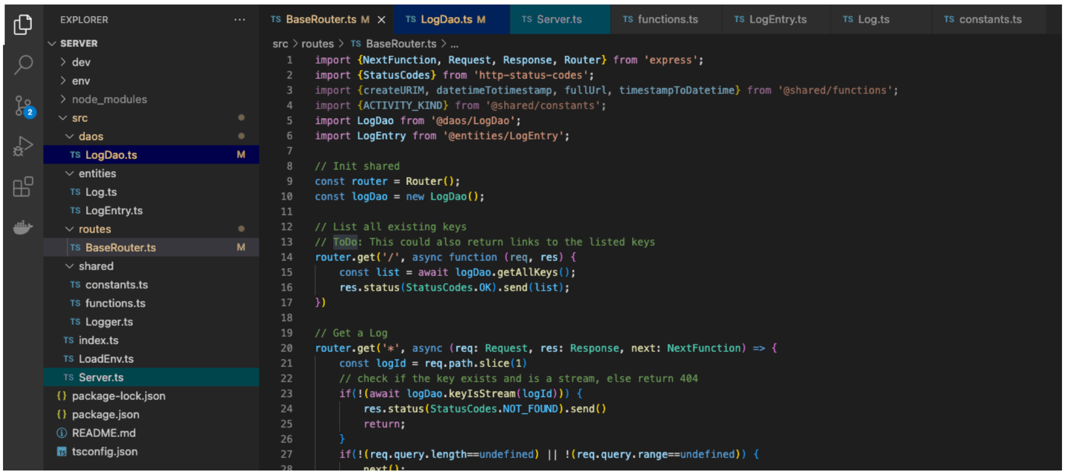
Another factor that Gasparic et al. hypothesize to be a reason why RSSEs are not implemented in common environments is their lack of reasoning. Most systems output artifacts in flat lists and expect the developer to either know or figure out why these artifacts are recommended or useful to their task. Providing reasoning for recommendations means giving explanations of some sort why a certain artifact is being recommended to the user. Especially in our case, as our system uses two different types of sources for recommendations it is important two clearly explain where a recommendation came from.

We want to tackle both issues 1) the uninspiring output of flat lists and 2) the lack of reasoning, with the same set of novel visual aids in the IDE interface.

3.3.3 Design Choice: Color Highlights

We want to use color highlights for reasoning and as an additional form of output visualization.

As we have discussed earlier in this chapter, a goal of the framework should be to extend the IDEs package explorer and general user interface. In our literature review, we have seen the interrelation of code comprehension and navigation. Developers struggle with both since often code bases grow up to many thousand files and millions of lines of code. Having a good mental model of a project also means knowing how to effectively navigate to desired code fragments. That also means knowing where to find them in the project folder/file structure. Especially for well-modularized and well-kept code repositories, the overall folder structure will represent a semantic model of the project. Code in files within close proximity can be expected to be somehow related. If related code is strewn across the project, in different directories or parts of the repository, evolutionary couplings as well as structural dependencies can help form the associations that are not visible in the project structure. However, simply providing hyperlinks in the form of artifact recommendations in the list does not give developers a good idea of where that link takes them and where the related code is actually located.



```
import {NextFunction, Request, Response, Router} from 'express';
import {StatusCodes} from 'http-status-codes';
import {createURIM, datetimeTotimestamp, fullUrl, timestampToDatetime} from '@shared/functions';
import {ACTIVITY_KIND} from '@shared/constants';
import LogDao from '@daos/LogDao';
import LogEntry from '@entities/LogEntry';

// Init shared
const router = Router();
const logDao = new LogDao();

// List all existing keys
// ToDo: This could also return links to the listed keys
router.get('/', async function (req, res) {
    const list = await logDao.getAllKeys();
    res.status(StatusCodes.OK).send(list);
})

// Get a Log
router.get('*', async (req: Request, res: Response, next: NextFunction) => {
    const logId = req.path.slice(1)
    // check if the key exists and is a stream, else return 404
    if(!(await logDao.keyIsStream(logId))) {
        res.status(StatusCodes.NOT_FOUND).send()
        return;
    }
    if(!(req.query.length==undefined) || !(req.query.range==undefined)) {
        next();
```

Figure 3.6: A very early mockup of the color highlight idea from our conceptual phase. It shows the idea to highlight recommended artifacts with colors in all places of the IDEs interface where the file descriptor is rendered. In the mockup, the file descriptors with blue backgrounds are highlighted for recommendation.

To reduce this *portal-like* feeling of just having recommendations as hyperlinks, we want to enhance the package explorer and general user interface of the IDE with visual aids to indicate locations of related code elements.

Inspired by tools like *Mylar* and *HeatMaps* we plan on implementing visual cues through color highlighting of interface elements. We plan on giving color highlights to all currently recommended artifacts in the project (see Figure 3.6). As the file descriptors for artifacts appear in multiple locations in the IDEs UI (package explorer, our recommendation view, search results, open tabs...) a recommended artifact could be highlighted with color where ever its file descriptor is being displayed.

Color highlights mark recommended elements wherever they occur in the IDE interface.

Color Highlights throughout the IDE UI should provide additional accessibility to recommendations and guide attention to recommend artifacts and their whereabouts. A recommended file could be visible and opened from our recommendation view or the package explorer or in some other view like the search results. Seeing highlighted elements in the package explorer could give users a feel for where in the project related code is located and how different parts of the project are connected. As shown in Figure

3.6 we can provide reasoning for the different kinds of recommendations by utilizing different color highlights based on whether a recommendation came from a mined evolutionary coupling or from the call graph analysis. This could provide an initial short learning curve but would afterward be an intuitive and quick source of reasoning for recommendations.

3.3.4 Design Choice: File Decorations

File decorations can be used as additional reasoning measures.

In addition to the Color Highlights, we want to provide more reasoning through other means than just color. Color Highlights could prove to be inaccessible for visually impaired people or be perceived as distracting or obstructive by some. In those cases, they should be optional or customizable and not be the sole source for their transmitted reasoning.

Therefore, other decorative elements could be added to the recommended artifacts file descriptors. These include file decorations like icons to be displayed next to the filename, badges, and descriptive tooltips explaining where a recommendation was sourced. Different icons and badges could signify that recommendations came from the evolutionary coupling miner or from the call graph. Tooltips can be used for purely textual reasoning.

3.3.5 Design Choice: Aggregation & Ranking

The merged recommendations will be grouped and ranked under their parent files.

Building a framework to host recommendations from two different sources poses another challenge in their presentation: Do we keep the differently sourced recommendations apart from each other or merge them into one display? Utilizing the concept of a Tree View we decided on aggregating recommendations of multiple sources found in one file into a single tree node in the recommendation view. We made the decision based on two reasons. 1) It reduces the amount of screen space allocated by the extension. Separating the two sources would mean that a single file that contains both types of recommendations would have to be recommended twice in the view. 2) Aggregation also provides a visual summary of the degree of interest that a recommended artifact holds. An aggregated list of contained related code elements shows the developer how many related fragments can be found in the file which should often be proportional to how interesting the file itself is in the current context.

The high-level nodes, representing files, also need to be ranked to order them based on relevance. In our conceptual approach, all recommendations on the method-level will be assigned weights. Evolutionary couplings will receive a higher weight ($w_{evolve} = 2$) than structural ones from the call hierarchy ($w_{call} = 1$). A recommended file's weight will be calculated based on all the recommended code blocks that it contains.

$$weight_F = \sum_{c \in F} weight_c$$

The highest level of items in the displayed Tree in the recommendation view will then be ordered based on descending values of weight.

Chapter 4

Implementation

In this chapter, we will outline the implementation of our conceptual approach for a user-friendly, collaborative RSSE framework presented in Chapter 3. The implementation is based on a client-server infrastructure with the option of running completely client-side at the cost of collaborative features. In the following sections, we first present the architecture of the system in Section 4.1 before going into more detail on the implementation of the client-side extension in Section 4.2. Finally, we will discuss the implementation of the collaborative recommendation server in Section 4.3.

4.1 System Architecture

We implement the recommendation system in a client-server infrastructure. The client-side is the VS Code extension.

The recommendation system implements multiple components in a client-server infrastructure, illustrated in Figure 4.1. The client-side is implemented as a Visual Studio Code extension using the VS Code Extension API[1]. It is written in TypeScript[2] and features several different components. These components serve different purposes during the context formation and recommendation presentation phases. The extension handles all data collection and visual out-

[1] https://code.visualstudio.com/api (accessed on May 11th, 2023)

[2] https://www.typescriptlang.org (accessed on May 11th, 2023)

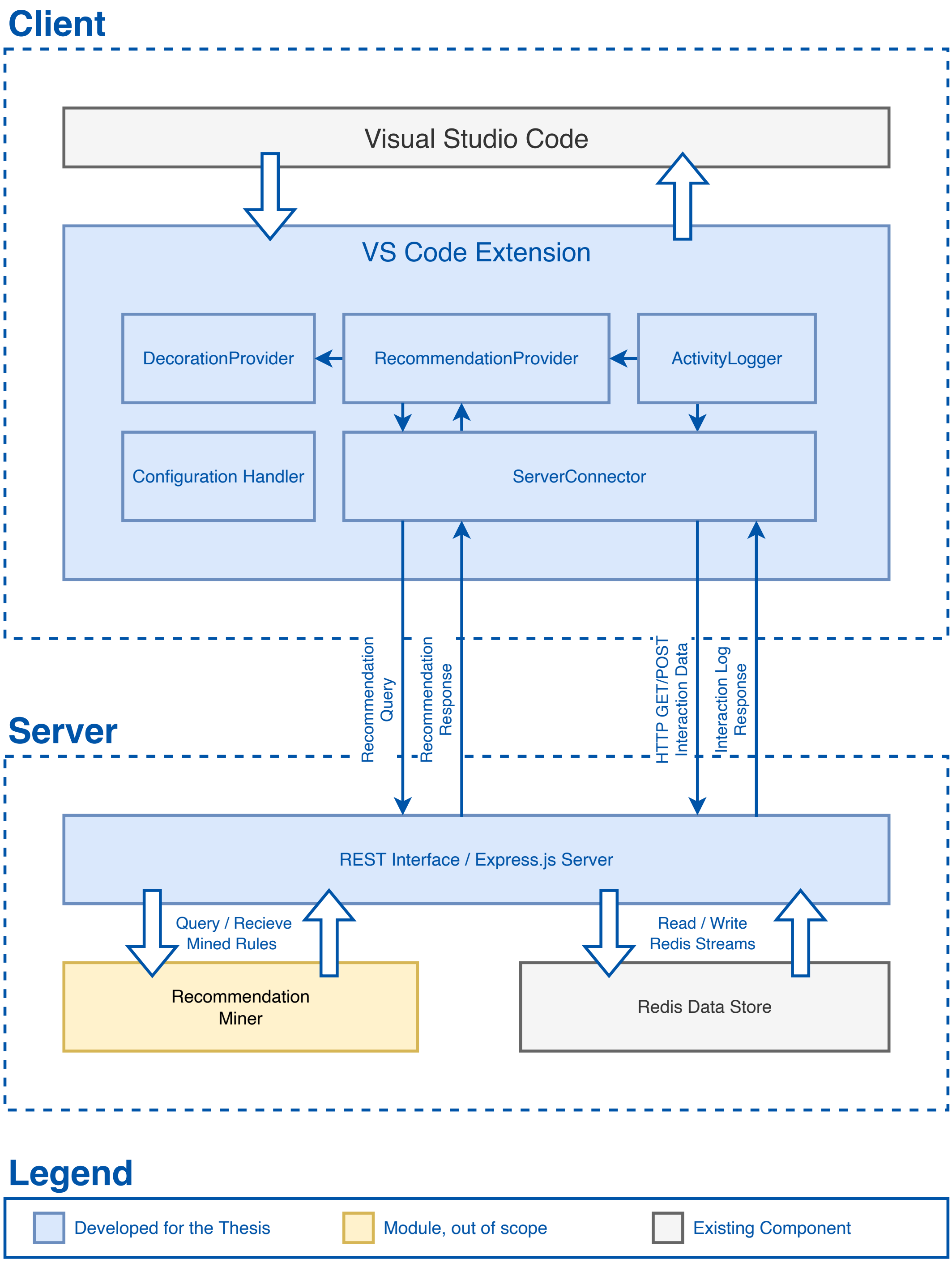


Figure 4.1: The architecture of the recommendation system. The blue parts represent components that were developed during and for the thesis. The yellow part indicates the recommendation miner, a module that can be swapped for various different techniques (out of the scope of the thesis). The grey parts describe existing components that are integrated into the system.

put of the system. A connector enables the extension to communicate with a server hosting collaborative interaction logs and data mining functionalities.

The server is implemented as an Express.js[3] server exposing a REST API. The server-side components fulfill data storage and querying tasks as well as the data-driven mining functionalities of the system. Incoming communication from clients via the REST interface is filtered and the queries are directed to the appropriate backend component, based on the type of request. For data storage, a Redis[4] data store is used with stream data. Queries concerning recommendations are sent to the recommendation mining module. The module can be implemented through various techniques and can be swapped based on usage requirements.

The server-side of the system acts as a REST interface handling recommendations and developer interactions.

4.2 Client Side: Visual Studio Code Extension

The client-side application is written as a VS Code extension in TypeScript. The VS Code Extension API lets us contribute new views to the IDEs interface, as well as commands that the developer can use inside the IDE. Most importantly, our extension contributes a new view called `RecommendationView` (placed by default in the explorer container) to display the generated recommendations. As illustrated in Figure 4.1, the extension features various components that handle the conceptual requirements formulated in Chapter 3.

4.2.1 Tracking Developer Interaction

In order to track the developer's interaction within the IDE, we implemented a tracker called ActivityLogger. An activity is of the kind:

The extension tracks developer interactions using the ActivityLogger.

[3] https://expressjs.com (accessed on May 11th, 2023)

[4] https://redis.io (accessed on May 11th, 2023)

Kind	Timestamp	Path	Symbol	User

VIEW activities are logged when the active editor is changed.

Where Kind is the activity type and either "VIEW" or "EDIT", Timestamp is a timestamp in the format returned by JavaScripts Date.now(), Path represents the project-intern relative path to the opened document, Symbol represents the name of a specific symbol within the documents outline, and User is a string identifier for people working on a project. The ActivityLogger component registers several event listeners for activities in the IDE. View activities are caught using the onDidChangeActiveTextEditor listener. The event is fired whenever a document is brought into the focus of the active editor. If the caught event references a different editor from the last viewed editor – and the path of the document shown in the editor is being tracked – then the following activity is logged:

Kind	VIEW
Timestamp	1673973400867
Path	/path/to/document
Symbol	
User	lmueller

Table 4.1: A VIEW Activity in the Interaction Tracker.

EDIT activities are logged when changes are made to a document.

When changes are made to a text document in the IDE, the onDidChangeTextDocument event is fired. The ActivityLogger catches the event and performs a number of checks on the registered edit event. First, a check is performed if the edited file is not ignored and is within the scope of the project workspace. Then, if the document has a symbol outline, the lines affected by the change are matched against the document's symbol outline. Up until a depth of 2 (e.g. methods and variables of containers such as classes), the symbol that has been affected by the change is computed. If the changed document has no symbol outline, only the document path is recorded. The ActivityLogger may log an activity of the form:

Interactions can be stored locally and on a configured server.

The ActivityLogger will always log interaction activity to a local interaction log, stored as a CSV file in the project

Kind	EDIT
Timestamp	1673973502753
Path	/path/to/document
Symbol	ClassSymbol{methodSymbol
User	lmueller

Table 4.2: An EDIT Activity in the Interaction Tracker.

workspace. If a collaborative server is configured and the extension settings are set to publish local interaction activity, the activities are also sent to the ServerConnector. The current context (a set (n, m) window containing the n-last viewed and m-last edited artifacts) is always exposed to the other components of the extension.

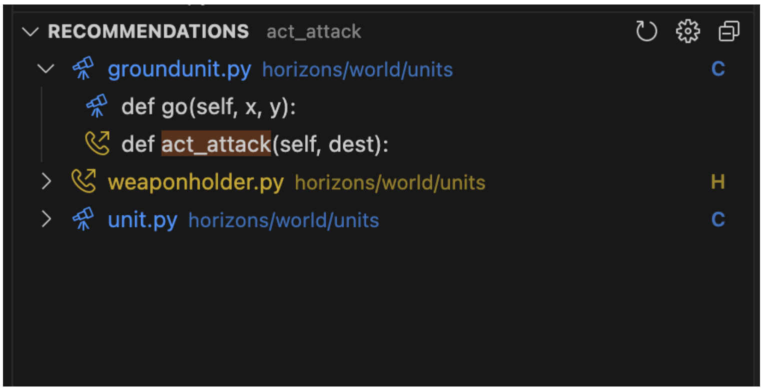
4.2.2 RecommendationProvider

The RecommendationProvider fetches, groups, and ranks recommendations.

The RecommendationProvider component handles the fetching, aggregation, ranking, and presentation of recommendations. It implements the `TreeDataProvider` interface from the VS Code Extension API. The interface acts as a data provider for a Tree View and implements functionalities such as fetching and refreshing content to render. In our case, the Tree View is the `RecommendationView` contributed by the extension.

Recommendations are generated automatically and formatted for the `RecommendationView`.

As soon as the developer interacts with the IDE, the RecommendationProvider gets the current context from the ActivityLogger. The context is then used to asynchronously query the collaborative server for recommendations, as well as fetching the immediate neighborhood of the call graph for the last edited elements. Both returned sets of artifacts are merged into one list, by aggregating the results by the files that they reside in. The result is a list containing `FileRecommendations`, each of which contains a list of child `CodeBlockRecommendations`. The merged list is formatted so that the VS Code Tree View recognizes it and is ranked by the weighting process defined in Section 3.3. An example can is shown in Figure 4.2.



RECOMMENDATIONS act_attack

groundunit.py horizons/world/units C

def go(self, x, y):

def act_attack(self, dest):

weaponholder.py horizons/world/units H

unit.py horizons/world/units C

Figure 4.2: The `RecommendationView` with three recommended files, containing multiple recommended code blocks.

A context indicator in the form of the last edited code block is also displayed in the `RecommendationView`. As shown in Figure 4.2, the `RecommendationView` features view actions for manual refreshing, collapsing all rendered `FileRecommendations`, and a shortcut to open the extension-specific settings. Clicking any of the rendered recommendations opens the target code block or file in a new tab as the active editor. The color highlights and additional decorations of the elements are implemented in the DecorationProvider. If no changes have yet been made in the session, or after start-up, a welcome view is rendered in the view.

4.2.3 DecorationProvider

File decorations are added by the DecorationProvider and based on the recommendation source.

The DecorationProvider component implements the `FileDecorationProvider` interface and is responsible for providing File Decorations to the project's file descriptors, based on their current state in the recommendation system. In the VS Code API, a File Decoration describes metadata that is rendered with a file in the UI. File Decorations consist of a badge, a color, and a tooltip (see Figure 4.2). Whenever the state of the context changes, the RecommendationProvider fetches and aggregates the new recommendations. Then, the DecorationProvider is notified of the changes and passed a list of URIs pointing to all files in the workspace that contain currently recommended code blocks. The DecorationProvider resets all File Decorations and updates them based on the passed list of URIs.

The DecorationProvider also handles the color highlights.

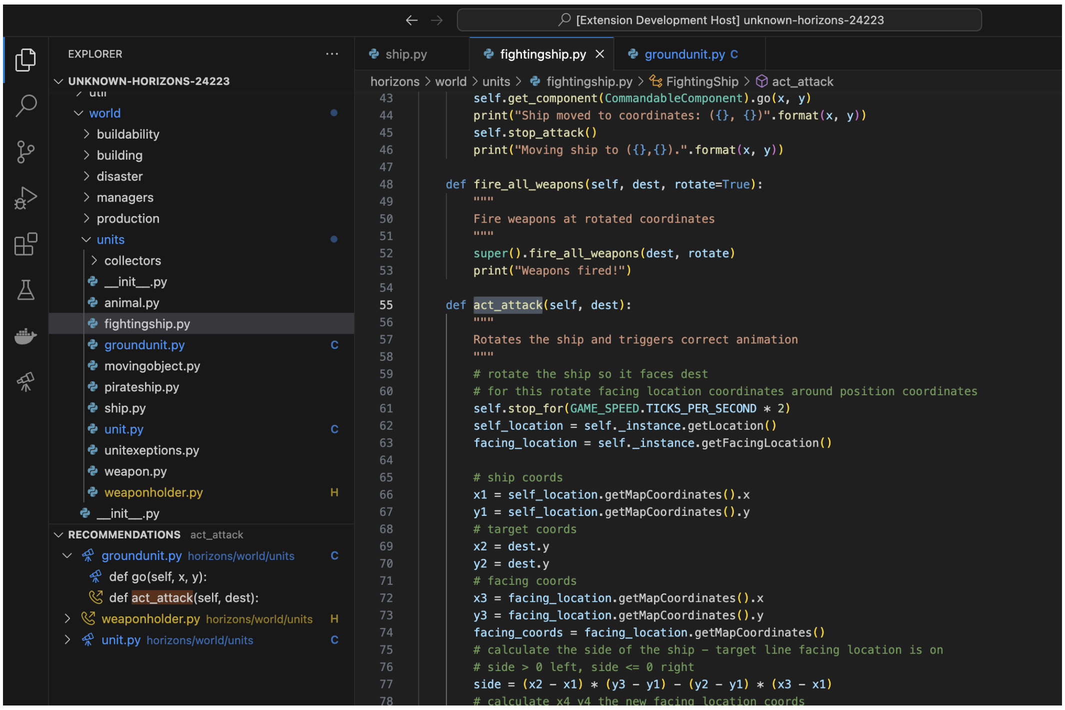
If a file only contains structural recommendations from the call graph, it is assigned a badge ″H″ (for hierarchical), the color *yellow*, and the tooltip ″Contains References″. These `FileRecommendations` as well as all structural `CodeBlockRecommendations` are displayed with a call graph icon. As soon as a file contains at least one recommendation from the miner module, it is assigned the badge ″C″ (for collaborative), the color *blue*, and the tooltip ″Often Changed Together″. These `FileRecommendations` as well as all evolutionary `CodeBlockRecommendations` are displayed with a telescope icon next to them.

The File Decorations are not only visible in the `RecommendationView` (as shown in Figure 4.2), but also in all other elements of the UI where file descriptors are rendered. Figure 4.3 shows how recommendations can be visible in the File Explorer, Search Results, and Tab Bar.

Color highlights propagate through folders in the file explorer.

Through the use of Color Highlights, recommendations are made visible in the project structure of the File Explorer. We suspect this to increase code comprehension significantly. When a folder in the explorer contains recommendations, it is also highlighted with color, propagating through parent directories up to the highest level of the explorer. The color highlights can then be followed even through a series of collapsed directories to the recommended files.

Caching Content: Views in the VS Code UI can be collapsed to hide their contents or be removed from the rendered interface elements altogether. In those cases, the RecommendationProvider utilizes a caching system that minimizes calls to the server. Hidden view states are cached for later usage. The extension however allows for the `RecommendationView` to be hidden but for the Color Highlights and other decorations to remain active and visible. In that case, the call for recommendations is still made but not applied to the hidden view.



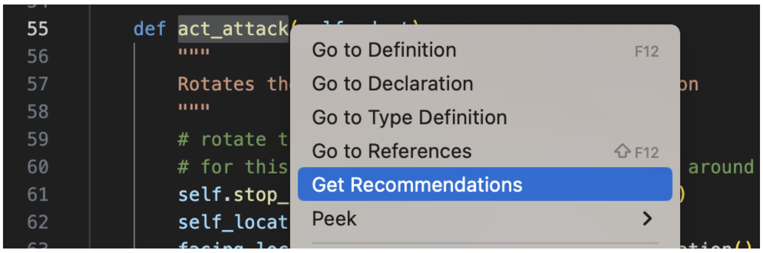
```
        self.get_component(CommandableComponent).go(x, y)
        print("Ship moved to coordinates: ({}, {})".format(x, y))
        self.stop_attack()
        print("Moving ship to ({},{}).".format(x, y))

    def fire_all_weapons(self, dest, rotate=True):
        """
        Fire weapons at rotated coordinates
        """
        super().fire_all_weapons(dest, rotate)
        print("Weapons fired!")

    def act_attack(self, dest):
        """
        Rotates the ship and triggers correct animation
        """
        # rotate the ship so it faces dest
        # for this rotate facing location coordinates around position coordinates
        self.stop_for(GAME_SPEED.TICKS_PER_SECOND * 2)
        self_location = self._instance.getLocation()
        facing_location = self._instance.getFacingLocation()

        # ship coords
        x1 = self_location.getMapCoordinates().x
        y1 = self_location.getMapCoordinates().y
        # target coords
        x2 = dest.y
        y2 = dest.y
        # facing coords
        x3 = facing_location.getMapCoordinates().x
        y3 = facing_location.getMapCoordinates().y
        facing_coords = facing_location.getMapCoordinates()
        # calculate the side of the ship - target line facing location is on
        # side > 0 left, side <= 0 right
        side = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1)
        # calculate x4 y4 the new facing location coords
```

Figure 4.3: The VS Code interface with visible `RecommendationView`, color highlights in the file explorer, and highlighted open tabs.

4.2.4 Configuration and Context Menu

Users can configure the extension with a collection of settings.

The VS Code Extension API allows extension developers to contribute configuration keys of their extension to be exposed in the VS Code Settings environment. At this point in development, users can configure the behavior of the extension by setting an output path for the ActivityLogger, a list of paths to be ignored by the ActivityLogger, the address of a collaborative server to be connected to the ServerConnector, a toggle to switch on/off the collaborative features, a toggle to switch on/off Color Highlights, and a personal identifier. In the future, additional configurations like custom colors for the highlights and custom icons for the `RecommendationView` could be added. However, to gain valid insights into the system's usability and efficiency, such configurations have not been implemented yet.



```
def act_attack
    """
    Rotates th                                on
    """
    # rotate t
    # for this                                around
    self.stop_                                )
    self_locat
```

Figure 4.4: The context menu item *Get Recommendations* can be used to generate recommendations for a symbol at the cursor. Using it does not require the symbol to be part of the current context or to be edited.

To allow the user to not only utilize proactive recommendations for the implicit context but also be able to request recommendations for other contexts, the extension contributes a menu item to the context menu in the editor (shown in Figure 4.4). *Get Recommendations* enables users to generate recommendations for the symbol at the cursor position in the editor. Activating the command is similar to performing a change to the symbol, by taking the symbol as the developer's current context. However, no change to the code block is required to receive recommended artifacts. The command is added in the navigational section of the editor context menu, below navigational commands like *Go to Definition*, *Go to Declaration*, *Go to Type Definition*, *Go to References*.

A context menu command, *get Recommendations*, is implemented that allows users to query for recommendations outside their current context.

4.2.5 ServerConnector

The ServerConnector is the component of the extension that handles communication with the recommendation server. It acts as an interface between the client-side extension and the server-side API. Other components can use its methods to request and receive information from the server's REST API. The ActivityLogger uses the ServerConnector to publish interaction data. The RecommendationProvider utilizes the component to query for context-based recommendations.

The Server Connector handles client-server communication on the client-side.

4.3 Server-Side

The server architecture handles collaborative interaction histories and recommendation generation.

As illustrated in Figure 4.1, the server-side consists of three main components. 1) A REST API implemented as an Express.js server; 2) A Redis Data Set; and 3) A Recommendation Mining module.

The recommendation mining module can be implemented as any recommendation engine that takes interaction data as input.

The implementation of the recommendation mining module is out of the scope of this thesis and will not be covered in much detail in this section. The module is designed to be easily changed to whatever recommendation mining technique is the desired requirement. For most state-of-the-art techniques like *MI*, the module will be a Python server exposing an API for communication and hosting a script implementing the miner. However, any mining algorithm that outputs sets of recommended software artifacts mined from interaction logs can be used in the module slot.

One server instance can manage arbitrarily many projects.

The server is designed to be instantiated once per organization that manages one or more software projects. An organization could be a company, a university chair, an open-source work group, or any other collection of individual developers working on shared repositories. One server can host interaction logs for arbitrarily many projects and manage input by arbitrarily many contributors.

4.3.1 Routing Client Requests

A `HTTP GET` to the server root will return a list of all tracked projects.

The interface component of the server-side handles all incoming requests from the client-side extension and routes them to the appropriate backend component. The REST API of the Express.js server takes `HTTP GET` requests for both interaction logs and recommendations. The interface exposes all projects for which the server manages interaction histories at the root domain `http://{server-url}/`. Issuing an `HTTP GET` request at the root will return a list of all keys for which the server stores an interaction log. A key usually represents a project ID and can be used to request the interaction history of the project with that ID.

For each managed project, the server exposes `http://{server-url}/pid`, where `pid` is the project ID. A `GET` request to that address returns the project's complete interaction log. Requesting to `http://{server-url}/pid?length` returns the number of log entries in that project's interaction log. By sending an `HTTP GET` request with the parameter *range*: `http://{server-url}/pid?range=X/Y`, where X, Y are timestamps and $X \leq Y$ one can request all log entries made between the time points X and Y in the interaction log.

Each project has a unique ID that also acts as its URI.

The client can send `HTTP POST` requests to a project's endpoint, with an interaction activity as payload, to append that activity to a project's log. On success, a 200 status code is returned. `HTTP PUT` requests can be used to put empty logs or to import existing logs from other servers.

Appends to a projects interaction log are done via `HTTP POST` requests.

Definitive documentation for requests made to the interface for querying recommendations is still under development. As the recommendation miner is supposed to be an exchangeable module, the required format of requests still depends on the implementation. In the future, however, the API should expose standardized endpoints with well-defined request requirements for recommendation querying. An endpoint like `http://{server-url}/pid/rec` could be defined for recommendation queries.

4.3.2 Storing Interaction Data

The system implements a Redis Data Set for efficient storage of interaction logs. Redis is an open-source, in-memory data store well suited for fast data access and stream-type data. For each new project to be tracked, a new Redis Stream[5] is added to the data store at the key corresponding to the projects ID. A Redis Stream is an append-only log that assigns unique identifiers to each entry. Each Redis Stream entry corresponds to a single developer interaction and has the interactions timestamp as its identifier.

Interaction Logs are stored as Redis Streams.

[5] https://redis.io/docs/data-types/streams/ (accessed on May 11th, 2023)

Chapter 5

Evaluation

In this chapter, we evaluate the results of this thesis both quantitatively and qualitatively. To do so, we conducted a user study on the implementation presented in the last chapter. The focus of the study was on *if* and *how* the implemented recommendation framework could help developers in navigating unfamiliar code projects. Section 5.1 will discuss the overall design of the conducted study as well as the goals we set out to meet. In Section 5.2 we will take a look at our participant group and the procedure of the study. Finally, we will present and discuss both the quantitative and qualitative results in Section 5.3 and Section 5.4

A study was conducted to investigate whether the framework can support developers in their navigation and comprehension of unfamiliar code.

5.1 Study Design and Goals

One of our main research questions is whether our proposed framework providing merged recommendations is able to improve code navigation and comprehension for newcomers to existing, large software projects. To accurately simulate the conditions that developers face when joining a new team we needed a target software project that the participants are not familiar with. We decided on the large open-source development project *Unknown Horizons*[1].

A large open source project was chosen to simulate programming tasks on.

[1] https://github.com/unknown-horizons/unknown-horizons (accessed on May 11th, 2023)

Unknown Horizons is a 2D real-time strategy game written in Python, in which the player builds a city on an island and manages different economical aspects of the resulting settlements. We decided on the project as it is written in a popular programming language and it is sufficiently large (132706 lines across 773 files). Additionally, the target project must not require much prerequisite domain knowledge so as not to skew the equality between participants. A game typically requires little to no domain knowledge to grasp its concepts and the little required background knowledge could be provided with a short video trailer and description of the game.

Participants should complete maintenance and debugging tasks that focus on navigation.

To evaluate the usability and effectiveness of the systems support we wanted to simulate software development tasks on the target project. The participants should perform software maintenance and debugging tasks that require them to navigate between various points in the code. In order to gain more insight into the participants' thought processes and decisions, we decided on performing the study as a think-aloud study (Jääskeläinen [2010]). A think-aloud study encourages participants to vocalize their thoughts during their performance and at critical points in the study run.

Evolutionary couplings for the open-source project were constructed realistically.

There exist no maintenance, debugging, or interaction histories for development work on the *Unknown Horizons* open-source project. Hence, we constructed evolutionary couplings between semantically related parts of the code. Since our research focus is on how we can integrate RSSEs into the developer workflow and IDE interface, it is sufficient to simulate scenarios with realistic recommendations. The evolutionary couplings we constructed result in the same kind of recommendations produced by state-of-the-art RSSE engines such as *MI*. The constructed evolutionary couplings are fed into the framework (as detailed in Chapter 4) as if they came from the recommendation miner module on the server-side. This way we can simulate the framework in a scenario in which meaningful recommendations from evolutionary couplings are merged with the structural recommendations from the call graph. The resulting system is a realistic simulation of a well-calibrated recommendation system that we can evaluate from a user-centered view rather than from a technological standpoint. In the future, tracking a real development team's work on a code project and mining evolutionary couplings from their interaction data could be material for another study that tests the applicability of the framework not for newcomers but for experienced developers of a project.

We compare four different conditions. Ground Truth and three versions of the framework.

Another important research question of this thesis is how well certain UI elements of the framework support the developers in their navigation and code comprehension. To get a better understanding of how valuable certain parts of our implementation are, we plan on evaluating different versions of the extension in a comparative study. We will treat four different conditions, each of which features a different recommendation display method:

C1: Ground Truth Basic Visual Studio Code

C2: Recommendation List Visual Studio Code *with* the `RecommendationView` available. No Color Highlights or other decorations are enabled.

C3: Color Highlights Visual Studio Code *with* Color Highlights and other reasoning decorations (badges, tooltips). The `RecommendationView` is disabled.

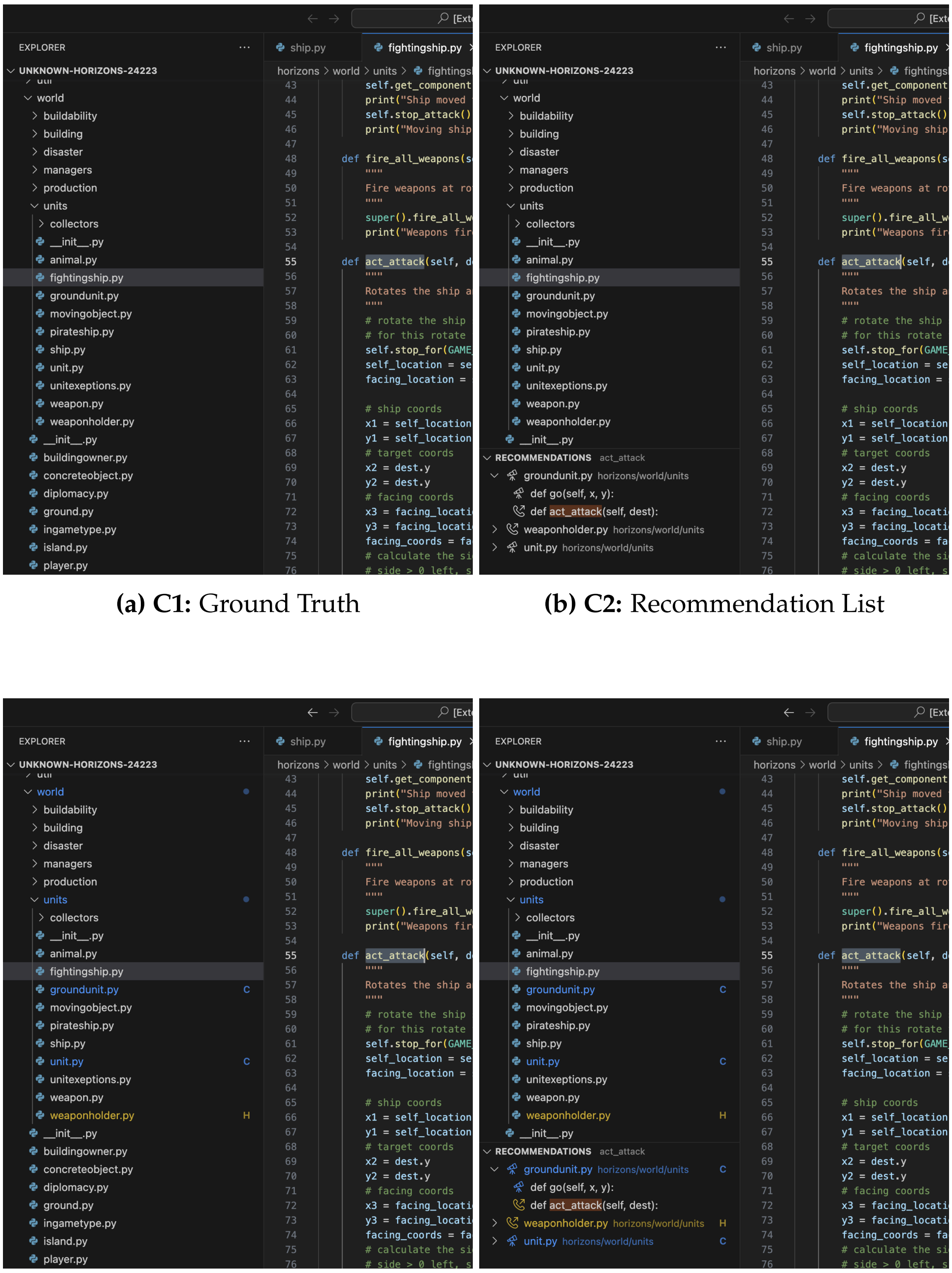
C4: Full Framework Visual Studio Code *with* the full extension activated.

The IDE interface in the four conditions can be seen in Figure 5.1. Each subfigure shows the IDE in one of the conditions within the same context. Conditions *Recommendation List* and *Color Highlights* were designed based on the most promising findings discussed in Chapter 2.

The conditions are randomized over four tasks.

To accurately represent performance and qualitative feedback across the four conditions – and to mitigate learning effects – we need to design four different task scenarios that the conditions can be randomized over. Each of the tasks should be approximately of the same difficulty and feature equally many subtasks. The subtasks should be evenly distributed over several optimal solution types (meaning the optimal way to solve that task). Since we are merging

Each task has three subtasks of different types.



```
            self.get_component
            print("Ship moved
            self.stop_attack()
            print("Moving ship

        def fire_all_weapons(s
            """
            Fire weapons at ro
            """
            super().fire_all_w
            print("Weapons fir

        def act_attack(self, d
            """
            Rotates the ship a
            """
            # rotate the ship
            # for this rotate
            self.stop_for(GAME
            self_location = se
            facing_location =

            # ship coords
            x1 = self_location
            y1 = self_location
            # target coords
            x2 = dest.y
            y2 = dest.y
            # facing coords
            x3 = facing_locati
            y3 = facing_locati
            facing_coords = fa
            # calculate the si
            # side > 0 left, s
```

RECOMMENDATIONS act_attack
groundunit.py horizons/world/units
def go(self, x, y):
def act_attack(self, dest):
weaponholder.py horizons/world/units
unit.py horizons/world/units

(a) C1: Ground Truth

(b) C2: Recommendation List

(c) C3: Color Highlights

(d) C4: Full Framework

Figure 5.1: All Four Conditions in the Same Context.

structural and evolutionary recommendations, we decided on designing three subtasks per task scenario. There are three types of subtasks:

- *None* - Subtask: A navigational task where the optimal solution path follows neither structural nor evolutionary links through the code
- *Structural* - Subtask: A navigational task where the optimal solution path features structural links
- *Evolutionary* - Subtask: A navigational task where the optimal solution path features evolutionary links

The tasks focus on navigational aspects.

The order of the subtask types in the four task scenarios should alternate. Overall, the tasks should not require much programming knowledge. Since we are only interested in the navigational aspects of the tasks, a low programming knowledge threshold ensures less skewed data caused by the different expertise levels of the participants.

We evaluate our thesis goals with various measures.

As defined in 1.3 "Thesis Goals", we aim to show that the framework is indeed able to support newcomer developers in their navigation (**G1**) as well as their code comprehension (**G2**). We will measure task success rates and task completion times for all conditions to evaluate whether we met **G1**. To check if we achieved **G2**, we will ask the participants code comprehension questions after each task scenario and compare the quality of answers across the conditions. For evaluation of the framework's adoption potential (**G3**), we will record all navigation tool usages of participants and how successful these were used. Additionally, Likert Scale questions on confidence and support satisfaction will be answered by participants after every single subtask. Observations on the perceived confidence and satisfaction with the navigation support will offer insights both into the performance aspects defined in **G1**, **G2**, as well as into the adoption potential of the framework (**G3**), and the color highlights (**G4**). The gathered results will allow us to compare the different conditions in terms of usability and efficiency (**G5**).

5.2 User Study

A pretest resulted in minor changes.

We conduct a user study with the aim of evaluating the extension against the goals defined in the previous section. The study started with a pretest phase during which two participants completed test runs of the study. The pretest resulted in minor changes to the survey's wording and clearer formulations of the comprehension questions. A video trailer of the open-source project was added to the scenario introduction because one pretest participant voiced confusion about the topic of the open-source project during the tasks. Also, a short tutorial for the navigation functions of the IDE and the ones of the extension was added before the actual study to guarantee an even knowledge of both. Participants that took part in the pretest did not participate in the actual study.

5.2.1 Participants

20 people, mostly from a CS background participated.

For the study, we recruited 20 participants (1 non-binary, 5 female, 14 male), 19 of which had a background in computer science. P2 came from a UX design background. The participants were aged between 22 and 51 years old (mean = 26.9, median = 25, stdev = 6.1), the second oldest participant was 31 (see Figure 5.2). 19 of the participants had experience coding in IDEs that ranged from 1 to 16 years (see Figure 5.3). Out of the 20 participants, 15 had worked with Visual Studio Code before. The group consisted of 4 research assistants (in computer science), 12 students (11 of which studied degrees of CS), 1 CS teacher, and 3 professional software engineers. Other demographic measurements on the participant group concerned their experience in various aspects of software development, visualized in Figure 5.4. All of the participants signed an informed consent form prior to the study that can be found at Appendix C.1.

Out of the 20 conducted runs, 5 were done remotely via Zoom. The other 15 were done in person. The location of the in-person runs had no impact on the study and could

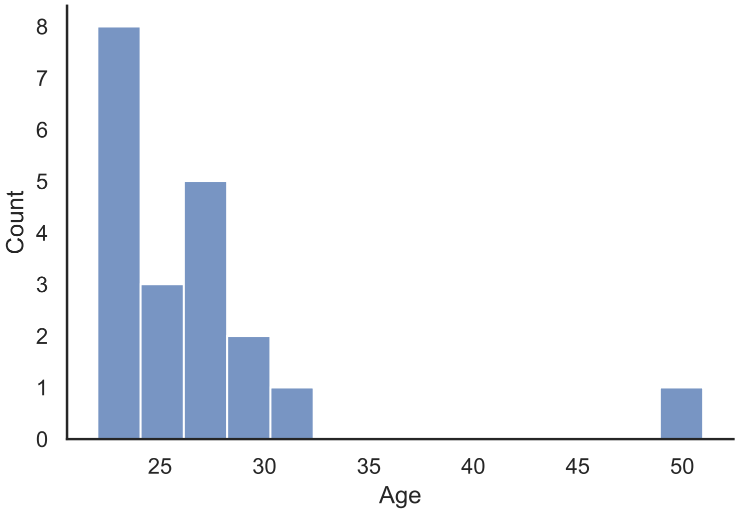


Figure 5.2: Histogram of the demographic measurement *Age*. $\overline{n} = 26.9, \sigma = 6.1$

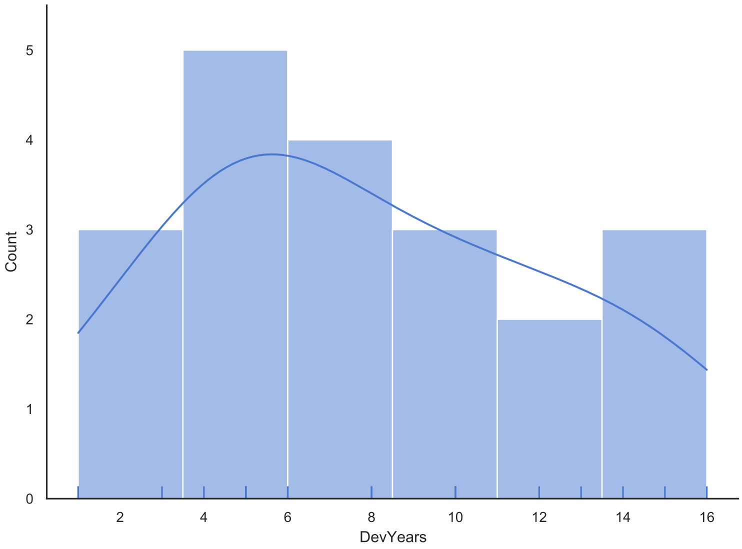


Figure 5.3: Distribution of the development experience in IDEs in years. $\overline{n} = 7.85, \sigma = 4.5$

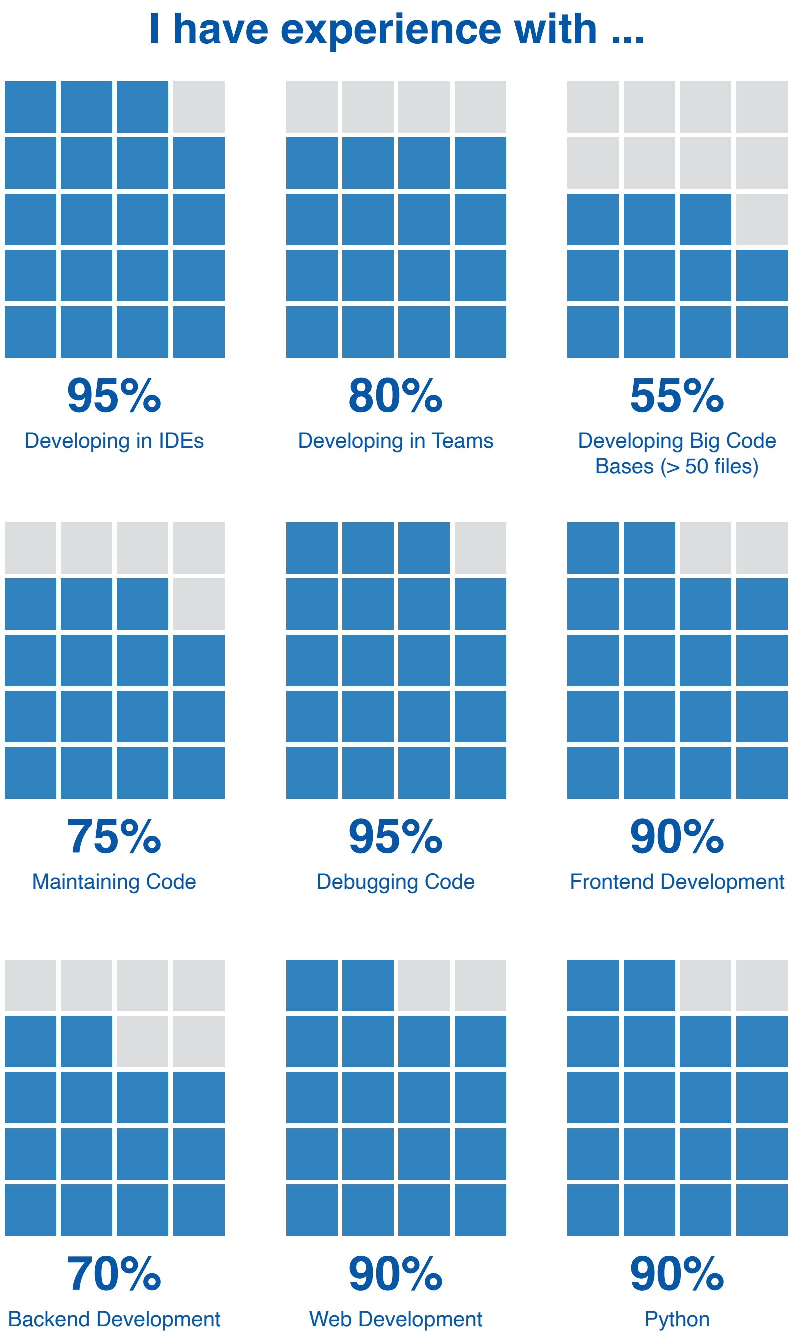


Figure 5.4: Software development experience measures of the study participants. Most had experience with development in IDEs, Python, and code debugging. Fewer were experienced with working on large code projects.

vary between participants. During all conducted study runs, the participants' screen and microphone audio were recorded using OBS for later analysis purposes. All recordings and collected data were stored anonymously and used only for analysis purposes.

5.2.2 Procedure

The study used a version of our extension adapted for the study. It ran on Visual Studio Code version 1.76, with the Pylance[2] extension for VS Code installed and enabled. The physical target system that participants performed the study on had no influence on the study or its results. In-person participants were offered the usage of a 2020 M1 MacBook Pro with either TouchPad or Mouse controls.

Participants were handed consent forms and surveys containing the tasks and questions.

After signing the informed consent form, each participant was given a short tutorial on how to interact with the VS Code IDE and its navigational tools such as *Go to Definition* or *Go to References*. In the same manner, a short introduction to the features provided by our extension was given. On a second screen, the participants viewed a survey (listed in Appendix C) that contained an introduction to the overall simulation setting and the open-source project at hand. The survey contained all four tasks and the respective subtasks, as well as all questionnaire items such as the Likert Scale questions and code comprehension questions. At the end of the four tasks, participants were asked to rank the encountered conditions by the amount of navigation support they gave them. Overall we performed 240 task trials: 4 Conditions × 3 Subtasks × 20 Participants. Finally, participants could give feedback in the form of appreciation, criticism, and suggestions on our framework.

[2] https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance (accessed on May 11th, 2023)

5.2.3 Minimizing Order Effects

In order to reduce the effect that the order of encountered conditions has on the participants' task performance, we randomized the order of conditions over the four tasks. The randomization was done based on the balanced Latin squares shown in Table 5.1. Each of the 20 participants encountered a unique order of conditions over the tasks. In ensuring so, we hope to minimize the order effect on task performances and the study results.

Task 1	Task 2	Task 3	Task 4
C1	C2	C4	C3
C2	C3	C1	C4
C3	C4	C2	C1
C4	C1	C3	C2
C4	C2	C3	C1
C2	C1	C4	C3
C1	C3	C2	C4
C3	C4	C1	C2
C3	C2	C4	C1
C2	C1	C3	C4
C1	C4	C2	C3
C4	C3	C1	C2
C1	C2	C3	C4
C2	C4	C1	C3
C4	C3	C2	C1
C3	C1	C4	C2
C1	C3	C4	C2
C3	C2	C1	C4
C2	C4	C3	C1
C4	C1	C2	C3

Table 5.1: The five balanced Latin squares used to randomize the four conditions over the four tasks.

5.2.4 Data Processing

We collected data through the means of video and audio recordings, as well as a survey, as described before. Task time data and tool usage data were extracted manually from the recordings and transferred into tabular form. Results from the various survey items were exported into tables as well. The data collected during the user study was processed in a semi-automated manner. First, we created standardized tables for all observations and measurements that were made. The tables, stored in CSV files, were imported into Jupyter Notebooks and analyzed using a mixture of Python libraries such as Pandas, Scipy, and Seaborn.

The study data was processed using CSV files and Jupyter Notebooks.

5.3 Quantitative Results

In this section, we will present and discuss the quantitative results from our conducted user study. We will analyze the collected data to find significant relationships and impacts of the different conditions on various performative aspects of the study. The results will give us indications if and to what extent we have met our research goals, and lead us into a general discussion of the findings.

5.3.1 Analyzing Success Ratios

A good indication of how much navigation support participants received from each of the conditions is the percentage of successfully completed subtasks. Highly supportive tool sets should enable more participants to correctly complete debugging/maintenance tasks in an unfamiliar environment. During the study, a cut-off time was implemented after which a subtask was stopped and filed as unsuccessful. The cut-off time was 10 minutes. A subtask is considered unsuccessful if it was done wrong, was given up by the participant, or took longer than the specified cut-off time.

The percentage of completed tasks gives us an idea of how supportive a condition was.

We analyzed the percentages of successfully completed subtasks both per task and per used condition. As for most of our measurements, comparing them based on used conditions is the most useful to answer our research questions. However, looking at the data grouped by the tasks that

We analyzed success ratios both per task and per condition.

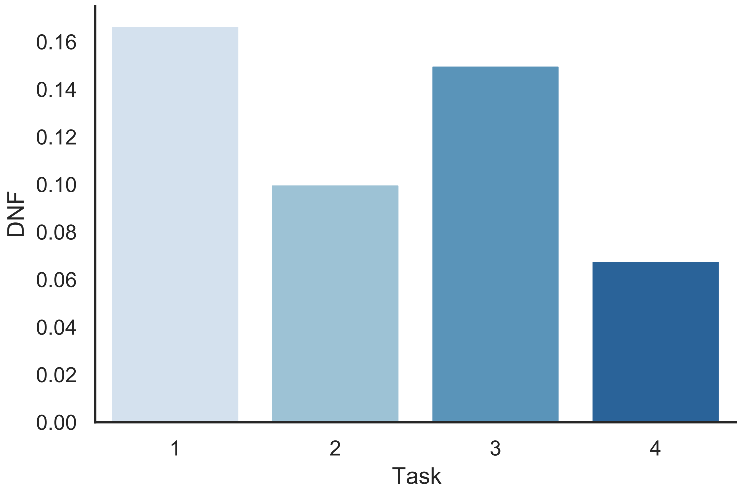


Figure 5.5: Ratios of unsuccessful subtasks per task. The X-Axis shows the four task scenarios and the Y-Axis shows the ratio of unsuccessful subtasks within the tasks (**D**id **N**ot **F**inish).

were performed gives us an idea of the tasks' perceived difficulty and the learning effects of participants during the study.

There appears to be a slight learning effect during the process of the study.

Figure 5.5 shows the ratio of unsuccessful subtasks for each of the four task scenarios. The data suggests a slight learning effect during the process of the study. This was to be expected since participants naturally became more acquainted with the code project and the nature of the tasks. While it is possible that Tasks 2 and 4 were slightly easier than Tasks 1 and 3, the order of the difference would not be too great. All tasks saw percentages of unsuccessful tasks between 6.7% and 16.7%.

The used condition had significant effects on the success ratio.

With respect to our research goals, we inspected the completion percentages grouped by used condition. The data for task completion was not normally distributed, which we confirmed with Shapiro-Wilk tests. Since it was not normally distributed, we chose non-parametric tests to check for significance in the effect of the condition on task completions. A Friedman test did indeed reveal significant effects of the condition on the likelihood that a task was com-

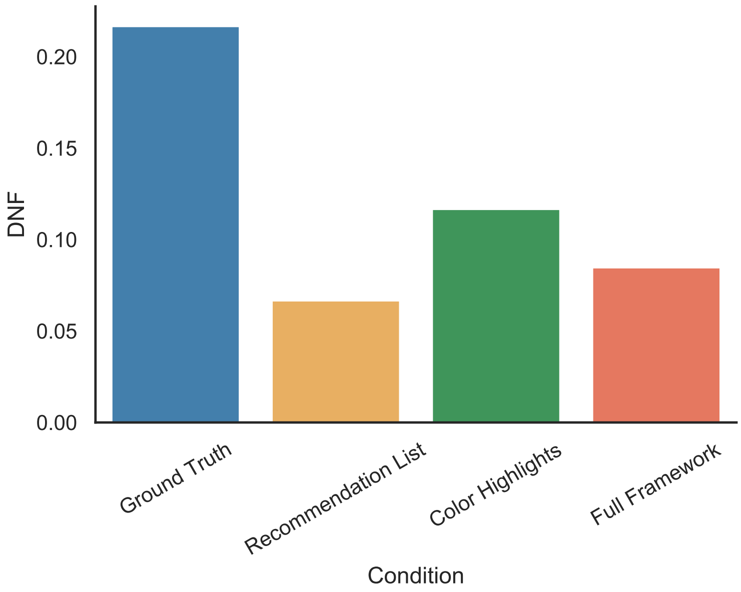


Figure 5.6: Ratios of unsuccessful subtasks per condition. The X-Axis shows the four conditions and the Y-Axis shows the ratio of unsuccessful subtasks (**D**id **N**ot **F**inish).

pleted ($\chi^2 = 11.32, p = 0.01$). We performed post hoc pairwise comparisons to identify significantly different pairs of conditions using Wilcoxon Signed-Rank tests. The results can be seen in Table 5.2.

Condition	Significance		Mean
C1: *Ground Truth*	A		0.217
C2: *Recommendation List*		B	0.067
C3: *Color Highlights*	A	B	0.117
C4: *Full Framework*		B	0.085

Table 5.2: Pairwise significance and mean DNF ratios for all conditions. Rows represent conditions. Two rows that are not connected by the same significance letter are significantly different ($p <= 0.05$).

As seen in Figure 5.6, the data paints a clear picture of which conditions helped participants to complete more subtasks successfully. Condition *Ground Truth* shows a

Conditions implementing our framework had the highest success rates.

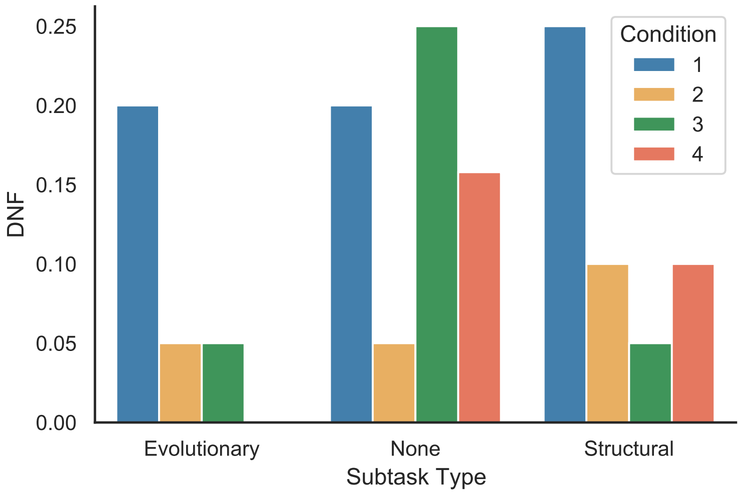


Figure 5.7: Ratios of unsuccessful subtasks per subtask type and condition. The X-Axis shows the three subtask types and the Y-Axis shows the ratio of unsuccessful subtasks (**D**id **N**ot **F**inish). The hue of the bars represents the conditions.

DNF ratio of 21.7%, almost twice as high as all other three conditions that used different versions of our framework. Even though the *Ground Truth* is not statistically significantly different from Condition *Color Highlights* ($p = 0.123$), the DNF percentage is notably higher. The percentage of failed tasks with *Color Highlights* (11.7%) is slightly higher than those of conditions *Recommendation List* and *Full Framework* (6.7% and 8.5%). This suggests that *Color Highlights* might be less supportive for newcomers to unfamiliar projects. It conveys less information than *Recommendation List* and *Full Framework* which both use the `RecommendationView` tree. This may be better compensated by developers that are already experienced with a project.

Framework conditions did outperform *Ground Truth* in all subtask types.

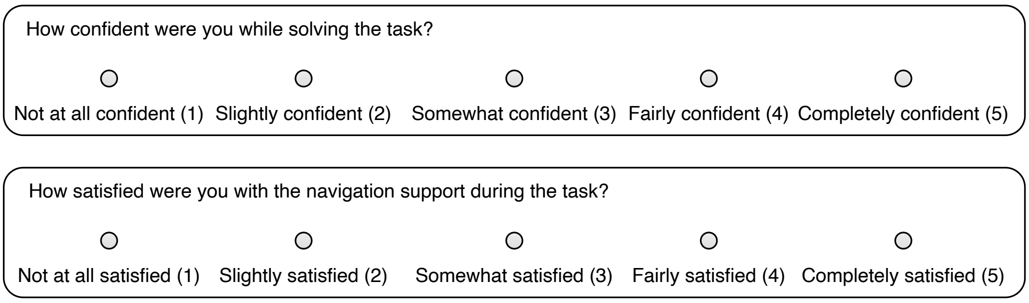
One threat to the validity of these results is that conditions 2,3,4 only outperform *Ground Truth* in the subtask type *Evolutionary*. To eliminate this threat, we analyzed the percentages of DNFs grouped by subtask type and condition (see Figure 5.7). As to be expected, conditions 2,3,4 performed significantly better on *evolutionary* subtasks. *Ground Truth* had no tools to support navigation between these code couplings. However, conditions 2,3,4 also show much stronger support for *structural* subtasks, suggesting that the prominent display of call graph information provided much more efficient navigation support than the state-of-the-art implementation in VS Code. This is in line with the findings of Krämer et al. [2013].

Participants displayed proficiency in using the *get Recommendations* command for exploration.

Figure 5.7 also shows that *Color Highlights* has the worst results for *None*-type subtasks. As explained above, *None*-type subtasks feature navigation and debugging tasks between code elements that have neither evolutionary nor structural couplings between them. They represent tasks that require active code exploration. *Color Highlights* may offer the least support for newcomers in these tasks due to its minimalist visualization of information and recommendations. In subtask *1.1*, a *none*-type, 60% of participants using *Ground Truth* failed to complete the task, while only 20% of participants using conditions 2,3 or 4 failed to complete it. Several participants displayed promising exploration strategies that deployed a mixture of the file explorer and our *get Recommendations* context menu function. During exploration of the code base they actively used the *get Recommendations* tool to request related points to certain code elements that they suspected could be relevant to the task. On several occasions, this led to useful recommendations (especially in the form of the `RecommendationView` in conditions *Recommendation List* and *Full Framework*) that allowed them to complete the subtask.

Color Highlights might be better suited for developers in familiar environments.

In general, we can see that the conditions *Recommendation List*, *Color Highlights* and *Full Framework* enabled participants to complete more tasks successfully than *Ground Truth* (as shown in Figure 5.6 and Figure 5.7. This suggests that the framework gave participants improved navigation support over the basic VS Code functionalities. With the exception of *Color Highlights* on *None*-type subtasks, our framework conditions performed better on all types of subtasks. We hypothesize that *Color Highlights* might be more useful to developers that are already familiar with the project they are working on. Since we only simulated newcomers in unfamiliar environments, however, this hypothesis remains to be investigated.



How confident were you while solving the task?

○	○	○	○	○
Not at all confident (1)	Slightly confident (2)	Somewhat confident (3)	Fairly confident (4)	Completely confident (5)

How satisfied were you with the navigation support during the task?

○	○	○	○	○
Not at all satisfied (1)	Slightly satisfied (2)	Somewhat satisfied (3)	Fairly satisfied (4)	Completely satisfied (5)

Figure 5.8: The two types of Likert questions posed to participants after each subtask. The corresponding numeric value to each answer is given in brackets.

Participants were able to solve up to 15% more tasks with the framework conditions.

The analysis of task completion percentages has shown that our framework does indeed support newcomers in their navigation of unfamiliar code environments. All conditions that used different versions of our framework performed better than the basic IDE condition *Ground Truth*. When using versions of our framework participants were able to complete 10-15% more tasks. We see this as an indication of having met our goal **G1**.

5.3.2 Confidence and Satisfaction

Likert questions on confidence and support satisfaction were asked after each subtask.

Overall, we collected 24 five-point Likert measures from each participant resulting in a total of 480 measurements. The Likert questions asked participants after each subtask how confident they felt during the task and how satisfied they were with the support provided by the currently used condition. The exact questions can be seen in Figure 5.8.

The conditions had significant effects on support satisfaction but not on confidence.

Although Likert data can be treated either as ordinal or interval data, we decided to treat it as interval data in this thesis. Our data was not normally distributed, so we chose non-parametric tests. Friedman tests showed that the condition had very significant effects on the participants' satisfaction with the navigation support ($\chi^2 = 22.32, p = 0.00005$), however no real significant effect on the participants' confidence ($\chi^2 = 4.42, p = 0.22$). The significance levels as well as mean and median confidence and satisfaction per condition are displayed in Table 5.3.

		Satisfaction		
Condition	Significance	Mean	Std Dev	Median
C1: *Ground Truth*		3.02	1.28	3
C2: *Recommendation List*	A	4.25	1.08	5
C3: *Color Highlights*		3.77	1.21	4
C4: *Full Framework*	A	4.24	1.13	5

		Confidence		
Condition	Significance	Mean	Std Dev	Median
C1: *Ground Truth*		3.15	1.41	4
C2: *Recommendation List*	None	3.87	1.16	4
C3: *Color Highlights*		3.65	1.27	4
C4: *Full Framework*		3.73	1.32	4

Table 5.3: Pairwise significance, as well as mean and median confidence and satisfaction values for all conditions. Rows represent conditions. Two rows that are not connected by the same significance letter are significantly different ($p <= 0.05$).

The median values for confidence give us a good idea of why there are no significant effects of the conditions on the participants' confidence. The median confidence value does not change between the conditions. Figure 5.9 shows the mean confidence per condition. Only slight differences are noticeable between the conditions *Recommendation List*, *Color Highlights*, and *Full Framework* implementing versions of our framework. The highest value nonetheless, was achieved by *Recommendation List*, surprisingly. Though, given that the differences are only very slight they may be neglectable. *Ground Truth* resulted in the lowest confidence values overall.

Recommendation List, *Color Highlights* and *Full Framework* have higher satisfaction values than *Ground Truth*

Conditions had a very significant impact on the participants' satisfaction with the navigation support they received. There are significant differences between all conditions except between conditions *Recommendation List* and *Full Framework*. Again, similar to the confidence values, *Recommendation List* and *Full Framework* show almost equally high satisfaction values with *Recommendation List* having a very slight edge. *Ground Truth* shows the lowest mean satisfaction value. *Recommendation List*, *Color Highlights* and *Full Framework* have higher median satisfaction values than *Ground Truth*. Mean satisfaction values are visualized in Figure 5.10.

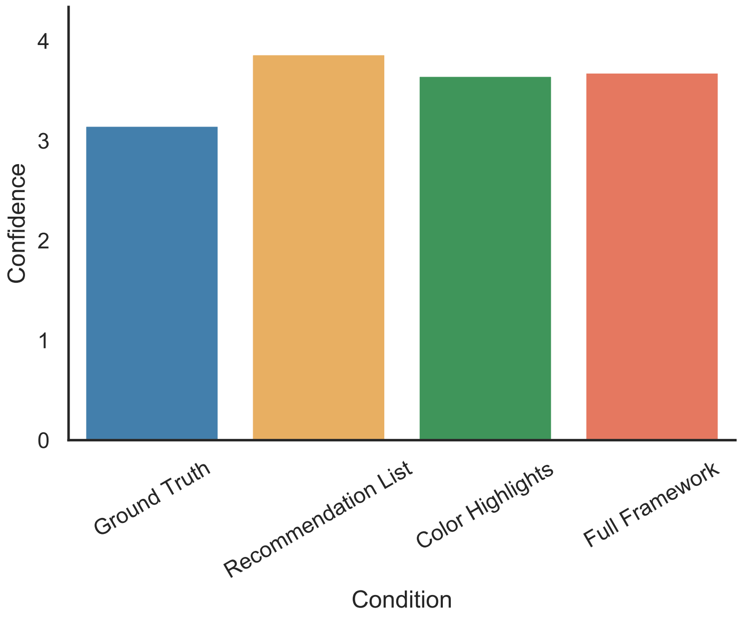


Figure 5.9: Mean confidence values per condition. X-Axis shows the conditions. Y-Axis shows the mean confidence value.

Figure 5.11 shows the effects of the condition on the distributions of confidence and satisfaction values in a histogram. While values for *Ground Truth* are barely clustered and are spread out widely, the values for conditions 2,3, and 4 show much higher concentrations and clusters in the higher ranges. *Ground Truth* shows a cluster around the (1,1) region, whereas the other conditions do not.

The tasks did not significantly impact the tool support satisfaction of participants.

We also analyzed the impact that the individual tasks had on confidence and satisfaction. Ideally, the tasks would have no significant impact on the perceived support satisfaction, as that would be a threat to the validity of our study. The data was not normally distributed, so again we opted for non-parametric tests. Friedman tests showed that there we no significant impacts of the tasks on satisfaction ($\chi^2 = 1.41, p = 0.7$). The tasks did however have significant effects on the confidence of participants ($\chi^2 = 12.03, p = 0.007$). Figure 5.12 shows the mean confidence values for each task.

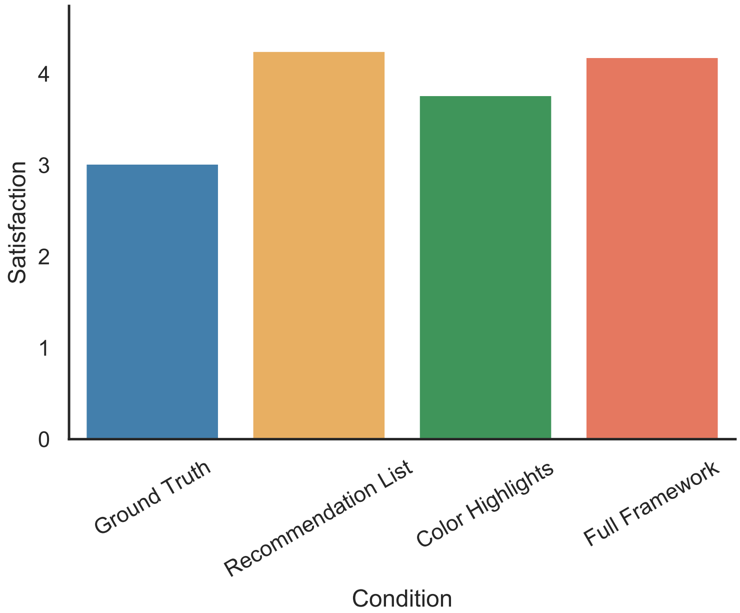


Figure 5.10: Mean support satisfaction values per condition. X-Axis shows the conditions. Y-Axis shows the mean satisfaction value.

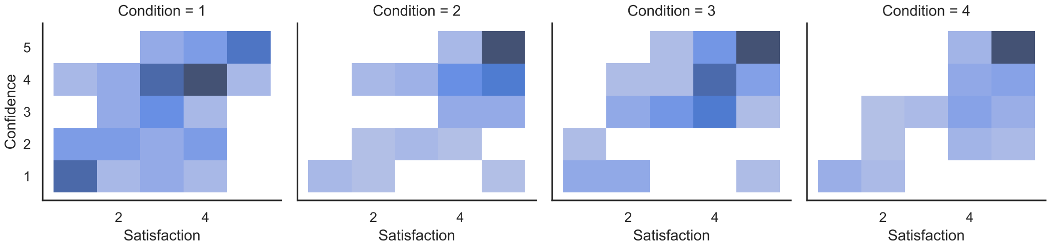


Figure 5.11: Distribution of confidence and satisfaction values per condition.

The significant effects of the task on participants' confidence can be interpreted as learning effects during the procedure of the study. Another Friedman test showed that the type of subtask (*Evolutionary, Structural, None*) had a slight effect on satisfaction ($\chi^2 = 5.86, p = 0.053$). This was to be expected since *None*-type tasks required the most active exploration and received the lowest mean satisfaction value ($\overline{s} = 3.45$). *Evolutionary* and *Structural* subtasks were close in mean satisfaction values and not significantly different from each other ($\overline{s} = 4.09$ and $\overline{s} = 3.87$).

Confidence values show a learning effect during the study.

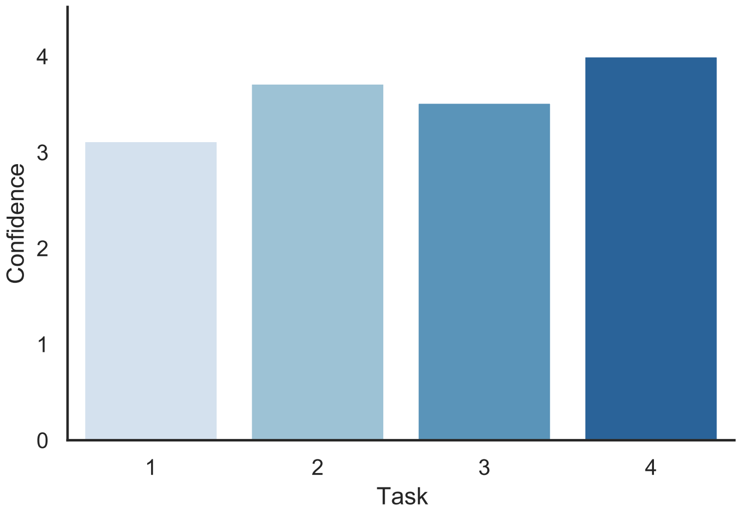


Figure 5.12: Mean confidence values per task. X-Axis shows the tasks. Y-Axis shows the mean confidence value.

Overall participants gave much higher satisfaction feedback for *Recommendation List*, *Color Highlights* and *Full Framework* than for *Ground Truth*. This confirms the framework's navigation support (**G1**) and promises a good adoption potential (**G3**).

5.3.3 Support Ranking

After having encountered all four conditions and worked through the tasks, participants were asked to rank the conditions based on how much support they felt they had received from the conditions. The average rank results – 4 being most supportive and 1 being least supportive – are shown in Figure 5.13.

While the ranking results are clear for conditions *Ground Truth* and *Full Framework*, conditions *Recommendation List* and *Color Highlights* were perceived as very similar in terms of the amount of support they gave to participants. Overall the ranking resulted in:

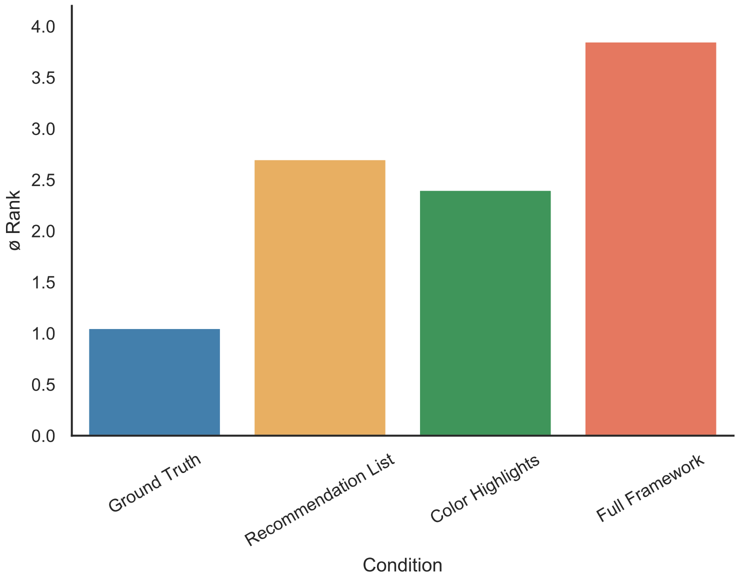


Figure 5.13: Average rankings in terms of perceived support. C1: $\overline{r} = 1.15$, C2: $\overline{r} = 2.6$, C3: $\overline{r} = 2.3$, C4: $\overline{r} = 3.95$.

1. **Full Framework**
2. **Recommendation List**
3. **Color Highlights**
4. **Ground Truth**

The first and last places in the ranking came as no surprise to us. The most fully featured version of the framework will be perceived as most supportive by most people. It is interesting to note though, that the ranking does not correlate with the perceived support satisfaction values discussed earlier. The mean satisfaction value for *Recommendation List* was slightly higher than that of *Full Framework*. In the ranking, *Full Framework* beats *Recommendation List* by a significant margin. The resulting order between *Recommendation List* and *Color Highlights*, even though very close, supports our hypothesis that richer visual information is useful for newcomers. However, the fact that the entire list-based paradigm of recommendation system outputs can be replaced by a minimal color highlighting approach and still

Participants ranked the *Full Framework* to be the most supportive in terms of navigation.

Recommendation List and *Color Highlights* are on close second and third position.

come close in perceived support, shows just how promising the approach really is. Again, we hypothesize that the benefits of the color highlights, as implemented in conditions *Color Highlights* and *Full Framework*, should become even more apparent in expert usage.

5.3.4 Completion Times

We recorded participants' task completion times.

As explained before, participants were timed during their task performances. Measurements were made for each subtask. Time measurements started with the participant's first interaction with the IDE and ended in either successful task completion, task failure, task cancellation by the participant, or time cut-off (10 minutes). On successful task completion, the measured time was recorded. In all other cases a DNF (**D**id **N**ot **F**inish) was recorded. In the following analysis of completion times, DNFs are treated as if hitting the cut-off time of 10 minutes.

We did not expect the task completion times to bear much significant insight into the navigation support given by the specific conditions. Since participants ranged widely in experience and expertise, we suspected the completion times to vary vastly and therefore be hard to compare meaningfully. Measurements like the discussed success ratio and confidence/satisfaction lend themselves much better for comparison across a diverse group of participants.

Due to a typo in the task description of task 4.3 that influenced Participant 5 in his performance, the time data from Participant 5 was excluded from the following analysis. The typo did not affect any participants before Participant 5 and later participants were informed of the typo beforehand.

The conditions had significant effects on the task completion times.

The completion time data was not normally distributed, so we chose non-parametric tests again. The Friedman test revealed that the used condition had significant effects on the completion time ($\chi^2 = 8.39, p = 0.038$). For post hoc pairwise testing, we performed Wilcoxon Signed-Rank tests. The results are displayed in Table 5.4.

	Completion Time					
	Significance			Mean	Std Dev	Median
C1: *Ground Truth*	A			236.7	220.5	124
C2 *Recommendation List*		B		130.1	146.0	85
C3 *Color Highlights*	A		C	180.6	180.3	115
C4 *Full Framework*		B	C	142.0	160.2	78

Table 5.4: Pairwise significance, as well as mean and median completion times for all conditions. Rows represent conditions. Two rows that are not connected by the same significance letter are significantly different ($p <= 0.05$).

Ground Truth is significantly different to conditions *Recommendation List* and *Full Framework*. *Recommendation List* is significantly different to *Color Highlights*, which marks a difference to the significance groups formed by the effect on DNF ratios.

Figure 5.14 visualizes the task completion times per condition. *Ground Truth* resulted in significantly higher task completion times. Conditions 2,3,4 that implemented versions of our framework all managed to significantly reduce the task completion times. In line with our previous analysis of DNF ratios and confidence and satisfaction, *Color Highlights* fares somewhat worse than conditions 2 and 4.

Ground Truth resulted in the longest task completion times. Conditions with `RecommendationView` lead to the fastest times.

We also analyzed if the tasks had an impact on task completion times. The results are the same as those of our analysis on task effect on DNF ratios (see Figure 5.5). Figure 5.15 shows the completion times grouped by task. Like in the earlier section on success percentages, the lower completion times as the study progressed can be attributed to the participants increasing familiarity with the software environment and the nature of the tasks.

Tasks had no significant impact on completion times.

Another interesting aspect of completion times is the effect that the different conditions had on completion times in the three subtask types. As shown in Figure 5.16, the conditions implementing our framework helped participants perform more efficiently in all three subtask types. One exception is marked by the efficiency of *Color Highlights* on *None*-type subtasks. We hypothesize that *Color Highlights* – due to having less visual information available – is harder to use for newcomers for code exploration in unfamiliar

Framework conditions made participants more efficient in all subtask types.

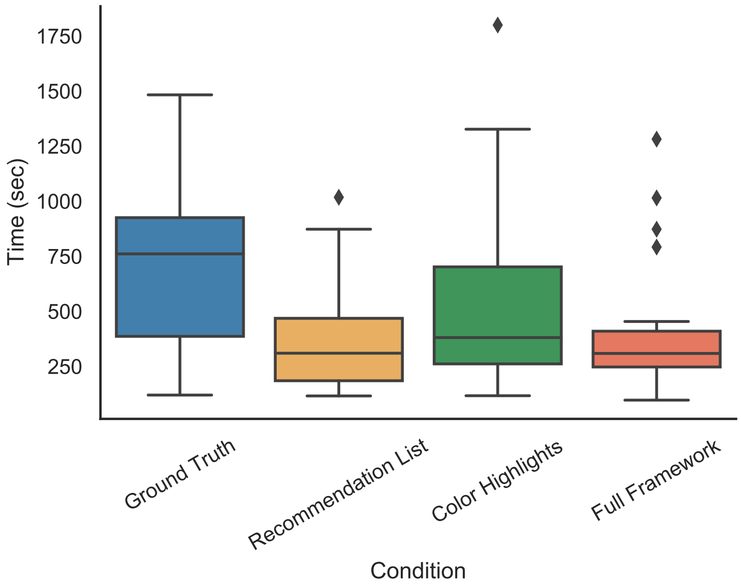


Figure 5.14: Task completion times per condition.

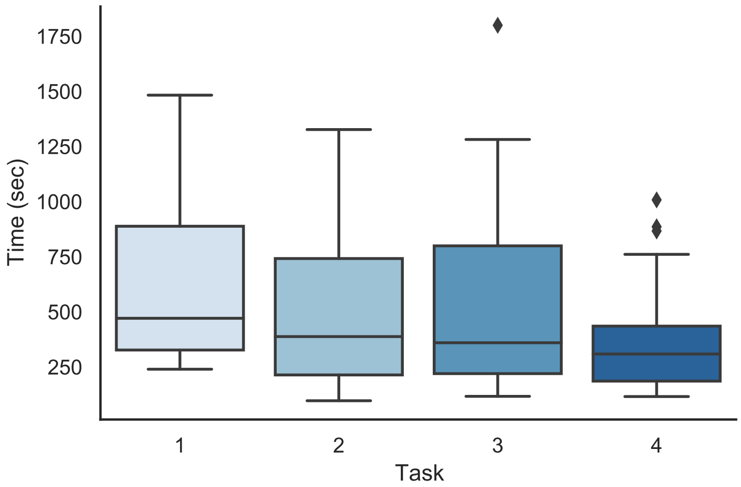


Figure 5.15: Task completion times per task.

projects. Expert users could potentially be able to explore code more efficiently using *Color Highlights*.

In conclusion, we found that the different conditions, contrary to our initial assumptions, significantly affected the

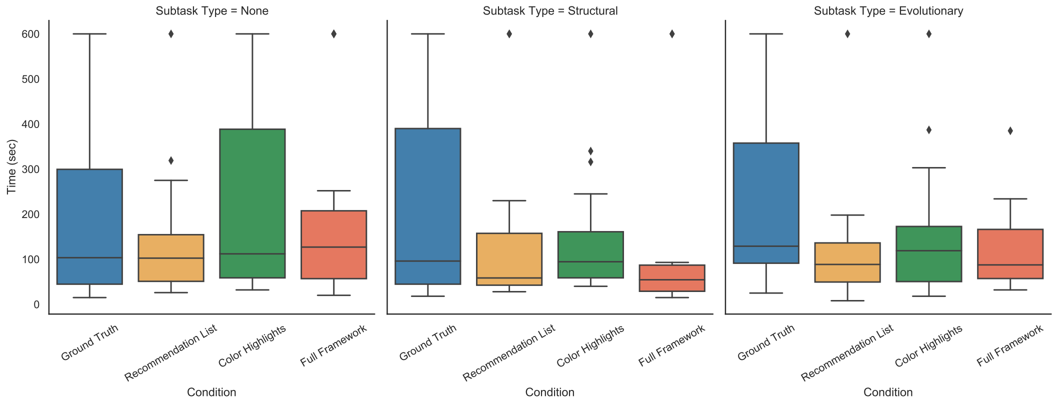


Figure 5.16: Task completion times per subtask type and per condition. Conditions 2,3,4 generally outperform Condition 1 by a big margin.

task completion times of the participants. Using the *Ground Truth*, participants took the longest to complete the tasks. All of the conditions implementing various versions of our navigation support framework enabled participants to complete tasks significantly quicker. The most efficient conditions were the tree view conditions *Recommendation List* and *Full Framework*. *Color Highlights* trailed slightly behind, which we hypothesize has to do with newcomers relying on aggregated visual information as provided by *Recommendation List* and *Full Framework*. Scanning the explorer for highlighted recommendations is a demanding task for newcomers. Given these results, in addition to the earlier discussion of success rates, we believe to have met our goal **G1**. The framework – in both fully featured and partly featured forms – is able to help newcomers solve software maintenance and debugging tasks much more efficiently.

5.3.5 Code Comprehension Questions

At the end of each of the four task scenarios, the participants were asked two comprehension questions. In total, we asked and recorded 160 questions and answers. All questions can be seen in Appendix C.2. The questions aimed to reveal whether participants had improved their mental map of the software project and their understanding of relationships and structures in the code. We mapped

Participants were asked code comprehension questions after using each condition.

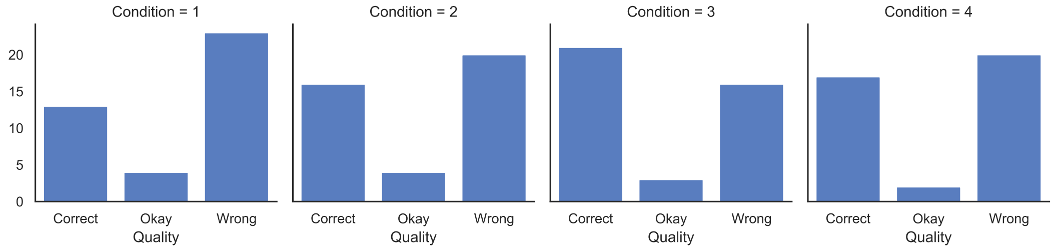


Figure 5.17: Distributions of *Correct*, *Okay*, and *Wrong* answers to the code comprehension questions per condition. *Color Highlights* is the only one with more than 50% correct answers.

all answers to three classes of answer quality: *Correct* (1), *Okay* (0.5), and *Wrong* (0). Figure 5.17 shows the distribution of answer qualities for each condition. The percentages of correctly answered comprehension questions are shown in Table 5.5.

Condition	% of Correct Answers
C1: *Ground Truth*	37 %
C2: *Recommendation List*	45 %
C3: *Color Highlights*	56 %
C4: *Full Framework*	46 %

Table 5.5: Percentages of correctly answered code comprehension questions for each condition.

Color Highlights was the only condition to achieve more than 50% correct answers.

We did not expect to find significant improvements during the developers' first few interactions with an unfamiliar code base. An indeed, Friedman tests did not reveal significant effects of the conditions on the code comprehension answer quality. However, we can see that *Color Highlights* did show strong improvements over *Ground Truth*. Participants answered 19% more of the questions correctly. Overall, given that the participants had never interacted with the code base before and only interacted with it for a short period of up to an hour, these results are very promising. Even within just the first four development tasks, participants were able to answer up to 19% more code comprehension questions using conditions that implemented versions of our framework. We expect these improvements to become all the more apparent once users spend more time working on a project with the framework.

The success of the color highlights can be attributed to the fact that they make developers more aware of the structure of the project. Since the recommendations are highlighted mainly in the file explorer, relationships between code elements in different parts of the project become visualized. This seems to have boosted the mental map of participants significantly. After using *Color Highlights* they showed a higher awareness of what certain parts of the project's functions were.

Color Highlights seems to improve developers' mental model of the project the most.

We see this as at least a partial fulfillment of our goal **G2** and definitive proof of color highlight value (**G4**). The framework did indeed improve participants' code comprehension through all conditions that it was implemented in. Our novel approach of using color highlights as the recommendation output mode showed the most promise. This good be a valuable insight for the further development and adoption of RSSEs and meets our goal **G3**.

5.3.6 Navigation Tool Usage

In order to get a better understanding of what tools participants liked to use and how successful they were with them we recorded their tool usage during the study. Not only can the participants' tools of choice tell us more about what tools are useful in their navigation tasks but also which tools they gravitate to when they are available to them. This is important for answering questions related to the adoption of RSSEs **G3** and the quantitative usability of tools **G5**.

We recorded what tools were used and how often they were used successfully.

First, we analyzed which tools in the IDE (and our framework conditions) were used by participants, how often, and on average how successful. We marked a tool in a subtask as successfully used if it was used in a purposeful way. That means if it directly or indirectly led to the successful completion of the subtask or contributed meaningfully to the participant's solution. A used tool was marked unsuccessful if it led participants to dead ends, led them astray, was disregarded after usage, or was used and the task was not completed successfully. It should be noted that participants were never encouraged to use any specific tooling in any of the conditions. They were allowed to use all the available functionalities of the IDE and the currently active conditions at all times.

The file explorer and global search were the most used VS Code navigation tools.

The collected data is visualized in Figure 5.18. The raw data can be seen in Table 5.6 From the data, it is obvious that the explorer is by far the most used navigation tool by newcomers in unfamiliar environments. This is in line with the research by Murphy et al. [2006]. The average success rate over all conditions however is only about 55%. The most popular group of tools are the search tools. The In-file text search was excluded from our analysis since it is not a true navigation tool for project-wide navigation. We still did record its usage and it was the most used tool with 273 uses. The most used search tool relevant to our analysis was the Global Search. It proved to be a go-to navigation tool for many participants – especially in condition *Ground Truth* – and had a success rate of 70%. One takeaway is that participants rarely ever used call graph-related navigation tools. *goToReferences*, *goToDeclaration*, and *goToTypedef* are all on the bottom end of recorded usages with lackluster success rates. Only *goToDefinition* was used moderately frequently with a 69% success rate.

All tools of our framework were used often and with very high success rates.

The tools exclusive to our frameworks conditions were *getRecommendations* (the context menu request), *ColorHighlight* (the color highlighted recommendations in conditions 3 and 4), *RecList* (the colorless `RecommendationView` in condition 2), and *ColorRecList* (the color highlighted `RecommendationView` in condition 4). All of our tool versions were used very frequently and successfully, as can be nicely seen in the cluster they form in Figure 5.18. This shows a promising willful and successful adoption of the framework's elements into the participants' workflow (**G3**).

We inspected whether the framework tools had positive or negative impacts on the usage of the standard navigation tools.

While tools like *ColorHighlight*, *RecList*, and *ColorRecList* are exclusive to certain conditions, most of the other used navigation tools were available to participants at all times. We analyzed the effect that the conditions had on the (successful) use of those tools. This is interesting, especially for condition *Color Highlights* that aimed to extend the IDEs existing navigation tools instead of implementing new views. Tools on which the effects of conditions were analyzed

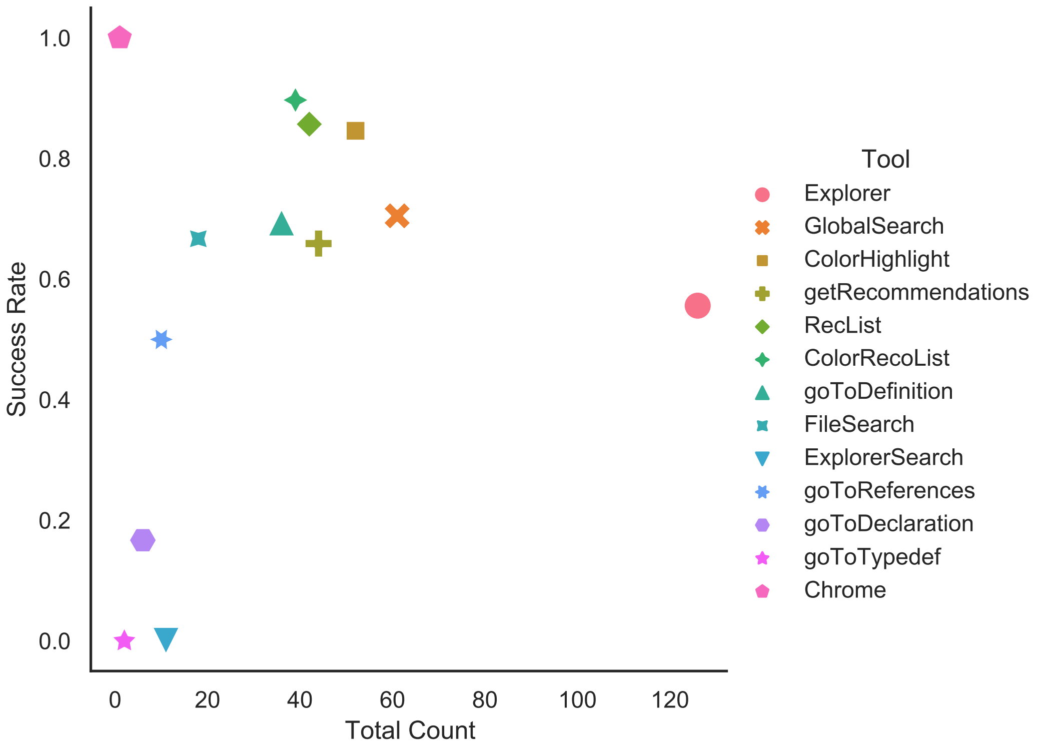


Figure 5.18: All tools that were used by participants during the study tasks. The X-Axis shows the total count of usages. The Y-Axis shows the rate of successful usage. Shape and color represent the various tools.

included *Explorer, GlobalSearch, getRecs, FileSearch, ExplorerSearch*, and *goToDef*. *getReferences* was excluded due to the fact that participants only used it in *Ground Truth*. In all other conditions, participants resorted to the framework's call graph visualizations and did so far more frequently and successfully (**G1**). *getRecommendations* was included because it was available in conditions *Recommendation List*, *Color Highlights*, and *Full Framework*, and frequently used.

Conditions had significant effects on the usage of some basic IDE tools.

Friedman tests showed that the used condition had significant effects on the successful usage of the *Explorer* ($\chi^2 = 35.6, p = 8.9e^{-8}$), the *getRecommendations* command ($\chi^2 = 20.12, p = 0.0001$), and the *FileSearch* ($\chi^2 = 1.34, p = 0.009$). No significant effects were found on *GlobalSearch*, *goToDefinition*, or *ExplorerSearch* (the last one being due to its 0% success rate).

Tool	Total Count	Success Rate
Explorer	126	0.556
GlobalSearch	61	0.705
ColorHighlight	52	0.846
getRecommendations	44	0.659
RecList	42	0.857
ColorRecList	39	0.897
goToDefinition	36	0.694
FileSearch	18	0.667
ExplorerSearch	11	0.0
goToReferences	10	0.5
goToDeclaration	6	0.167
goToTypedef	2	0.0
Chrome	1	1.0

Table 5.6: All used navigation tools and their respective usage counts and success rates. Ordered descending by total usage counts.

We performed pairwise post hoc Wilcoxon Signed-Rank tests to identify the different condition groups. The results, as well as all usage counts and success rates per condition, can be seen in Table 5.7.

Color Highlights significantly improved participants use of the file explorer.

For a better understanding of how the conditions impacted the tools usage count and success rate, we visualized the data in Figure 5.19. We can see that *Color Highlights* did indeed enhance and extend the explorer and its usability. Using *Color Highlights* participants used the explorer far more often and more successfully. *Recommendation List* and *Full Framework* reduced the explorer usage through the implementation of the `RecommendationView`. The *getRecommendations* command was also most successfully used by participants using *Color Highlights*. Because of how the color highlights boosted code exploration and comprehension, we assume that this has to do with the participants utilizing *getRecommendations* to request recommendations outside their current work context. *goToDefinition* was also most efficient when used in *Color Highlights*. This may be due to the lack of the extra `RecommendationView` and participants exploring this particular structural information in the way they are used to. The *GlobalSearch* proved

	Explorer				
Condition	Significance			Total Count	Success Rate
C1: *Ground Truth*	A			33	0.455
C2: *Recommendation List*	A	B		20	0.55
C3: *Color Highlights*				53	0.717
C4: *Full Framework*		B		20	0.3

	GlobalSearch				
Condition	Significance			Total Count	Success Rate
C1: *Ground Truth*				22	0.727
C2: *Recommendation List*	None			12	0.583
C3: *Color Highlights*				13	0.769
C4: *Full Framework*				14	0.714

	getRecommendations				
Condition	Significance			Total Count	Success Rate
C1: *Ground Truth*	–	–	–	–	–
C2: *Recommendation List*	A			11	0.636
C3: *Color Highlights*				20	0.8
C4: *Full Framework*	A			12	0.5

	goToDefinition				
Condition	Significance			Total Count	Success Rate
C1: *Ground Truth*				12	0.5
C2: *Recommendation List*	None			7	0.714
C3: *Color Highlights*				7	1.0
C4: *Full Framework*				10	0.7

	FileSearch				
Condition	Significance			Total Count	Success Rate
C1: *Ground Truth*				8	0.875
C2: *Recommendation List*	A			3	1.0
C3: *Color Highlights*	A			6	0.167
C4: *Full Framework*	A			1	1.0

Table 5.7: Pairwise significance of conditions on successful tool usage. Also, the total usage count and success rates per tool for each condition are shown.

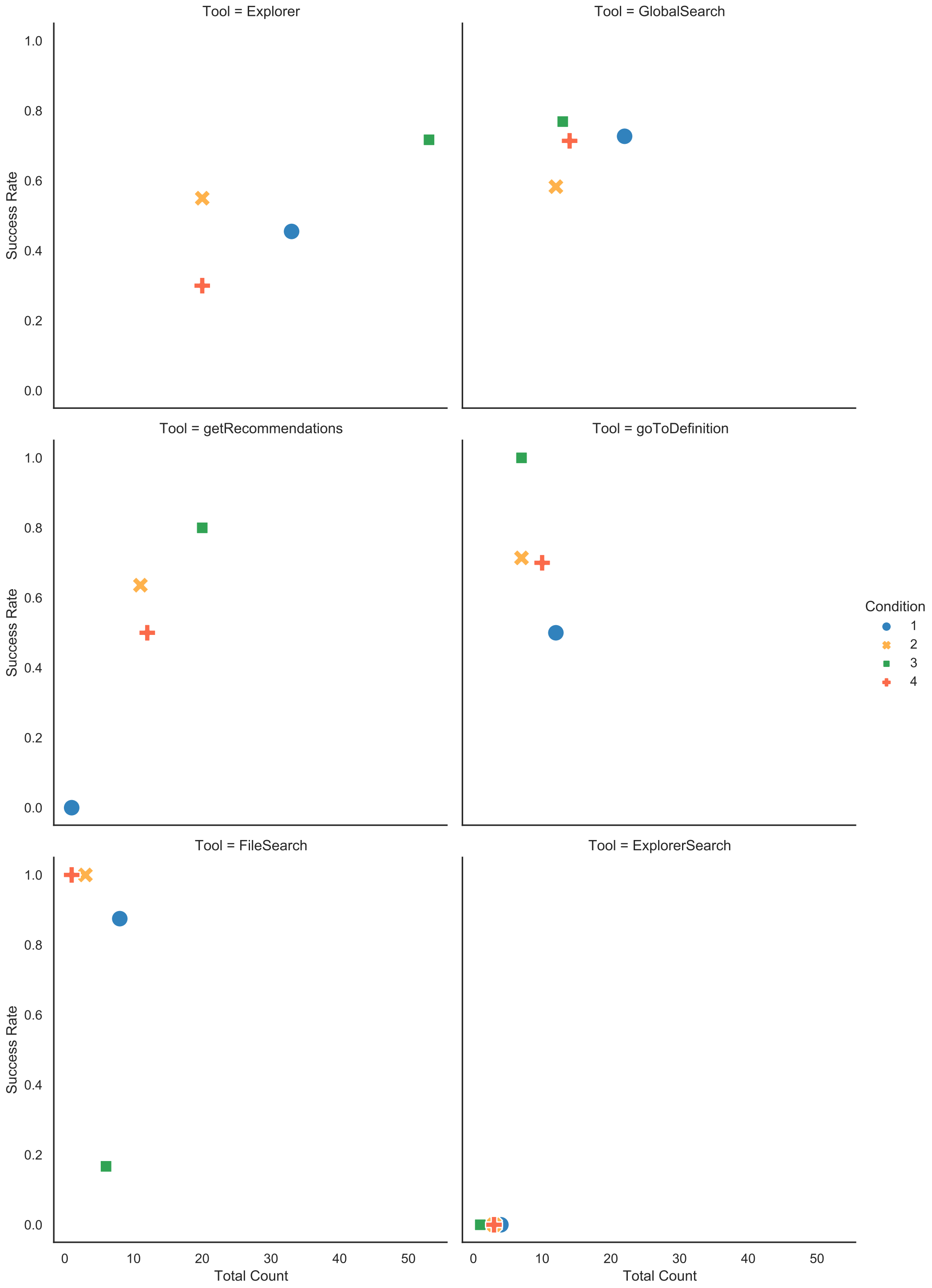


Figure 5.19: Relationship of total usage count and success rate per condition for each of the non-condition-specific navigation tools.

to be a useful tool regardless of condition. Both its usage count and success ratio barely changed between conditions.

The framework showed great adoption by the participants. *Color Highlights* proved to be a valuable addition for code exploration.

Overall, the analysis of used navigation tools in the study has shown that our framework (when available) has seen a very promising adoption by the users. In conditions *Recommendation List*, *Color Highlights*, and *Full Framework* the framework's tools were by far the most popular and most successfully used navigation tools. The conditions also had significant effects on the other navigation tools, commonly found in IDEs. *Color Highlights* enabled users to utilize the standard file explorer far more efficiently through the use of color highlighted recommendations (**G4**). The great adoption of the recommendation tools (both the `RecommendationView` and color highlights) shows the potential of well-integrated RSSE functionalities into modern IDE interfaces. We see this as clear proof of having met our adoption goal **G3**.

5.4 Qualitative Results and Feedback Discussion

In the following, we will discuss some of the qualitative results of the study. These include insights from the screen and audio recordings that are not represented in the survey data. Additionally, we asked participants for feedback on the framework. We aggregated common feedback points and observations, and discuss them in this section.

5.4.1 Active Code Exploration

Participants actively explored the code base with the *get Recommendations* command.

One point of concern that we had prior to the study was that participants might rely too heavily on the proactive nature of the recommendation system. Proactive recommendation generation takes away the need for an explicit user input but may give the impression that there is nothing besides the recommended content that could be of interest. Especially for the *None*-type subtasks, we thought that participants might get frustrated or give up when the initial proactively generated recommendations do not take them to the areas required in their tasks. However, participants showed great proficiency in using the *get Recommendations* context menu function to request recommendations for certain code elements. This allowed them to explore the codebase efficiently using the recommendations. A common observation was that participants scanned the code base (through the *GlobalSearch* or *Explorer* for example) for clues and code elements that could be related to their current task. Without having to edit anything in order to change their current work context, they began exploring recommendations of those elements with the *get Recommendations* function. This enabled them to solve *None*-type subtasks much more efficiently than in *Ground Truth* and showed very promising results in terms of improved code navigation and exploration.

Feedback

Some wished for keyboard shortcuts and peek functionality.

While the *get Recommendations* context menu command was very well received by all participants we obtained two points of feedback suggestions. P1 wished for a keyboard shortcut for the *get Recommendations* command. Similar to how in VS Code *go To Definition* can be executed by holding CMD and clicking a target symbol in the editor. P10 suggested the *get Recommendations* command to have a peek functionality similar to the native VS Code navigation tool *peek Call Hierarchy*. Both feedback points may be valuable in order to further integrate the framework seamlessly into the VS Code UI.

5.4.2 Undo Functionality

Some participants expected a way to retrieve previous recommendations.

During the study, several participants voiced confusion or slight frustration over the fact that there was no way to go back to previous recommendations in the `RecommendationView`. As soon as their work context had changed through edits the `RecommendationView` might contain different recommendations than before.

Some felt that they could be missing valuable information by not being able to go back and view the old recommendations again.

Feedback

The most common suggestion for this issue was the implementation of an *undo* or *back* button. P12 even expected the standard CMD+Z shortcut to also apply to the `RecommendationView`. A functionality to purely review old recommendations could be a useful addition to the framework. Having it bound to the undo functionality of the whole IDE, however, could prove to introduce complications with the recommendation engine's context window in actual use outside our simulated study. P1 and P6 both suggested a *back* button to browse through previous recommendation sets.

5.4.3 Jump to POI

Color Highlights was missing one important functionality: jumping to and marking the recommended code blocks.

Many participants remarked that they liked the `RecommendationViews` functionality of clicking a recommendation and jumping to the recommended code block in a new tab. As much this feature was appreciated in *Recommendation List* and *Full Framework*, it was missed by most participants in condition *Color Highlights*. When participants found useful recommendations through the color highlights in the explorer, they followed the highlights to the recommended document but could not see what was recommended within that file or where to look for it. They asked for a jump-to-POI functionality similar to when clicking on a recommended code block in the `RecommendationView`. This showed a significant shortcoming of our implementation in *Color Highlights*. Not only did *Color Highlights* show significantly less printed information on the screen but it also conveyed less information overall. Since no indication of what within a file is currently recommended was encoded in the color highlights, an entire dimension of the recommendation system was unavailable to participants.

Implementing line markers and jumps in *Color Highlights* could prove to make the condition more supportive.

This may be a possible explanation for why *Color Highlights* performed slightly worse than *Recommendation List* in many of the qualitative discussions we held earlier in this chapter. By finding a solution for how to visualize the missing dimension of method-level granularity through color highlights, the approach might produce even more promising navigation support results in future studies.

Feedback

The VS Code mini-map could be used to mark recommended code blocks.

Several participants had suggestions on how to improve the implementation of *Color Highlights* regarding the lack of method-level recommendations. P10 suggested that double-clicking highlighted files in the explorer could be used to jump to the recommended code block within that file in the editor. P3 and P5 remarked that the VS Code file mini-map could be used to highlight recommended code blocks in the files. We believe that this could be a very valuable solution to the issue. Additionally, it could prove to be viable for implementation since the same functionality is used in VS Code to highlight search results, warnings, and errors in the mini-map of documents.

5.4.4 (Colored) `RecommendationView`

The `RecommendationView` was very successful and liked by participants.

The recommendation view proved to be very intuitive for most participants. Participants were able to use it very efficiently very quickly for their exploration of related code elements. Our data discussed in Section 5.3 backs this up with the great results of conditions *Recommendation List* and *Full Framework* in almost all performed measures. The `RecommendationView` was used both often and successfully (compare Table 5.6) in the form of the standard *RecList* as well as the enhanced *ColorRecList*. Both versions had success rates of $> 85\%$. Participants remarked that having the prominent display of call graph data within the `RecommendationView` was much better than hiding it behind context menu commands.

Several participants said that they liked how the RecommendationView enhances and extends the file explorer rather than replacing it. Accordingly, most participants liked how it sat in the file explorer view container. P1 said that having it in the file explorer view container made it very easy for him to use the RecommendationView since he was already used to looking to that container for navigation purposes. P7, P10, P13, and P20 wanted to move the RecommendationView out of the file explorer view container. They moved it to the secondary sidebar on the right. Their motivation was that these participants were frequently using the *Global Search* functionality which is hosted in a different view container than the explorer and when activated replaces it. Hence, while using the *Global Search* they were not able to see the RecommendationView without moving it into the secondary sidebar. Being a VS Code extension contributing a view, our framework could accommodate for these different preferences in view placement.

Most participants liked that the view extended the explorer container.

A very common feedback point of participants was that they liked the compiled version of recommended items in the RecommendationView. They appreciated the fact that there was one single view to merge structural and evolutionary recommendations. Having a central point for related elements was perceived as way more useful than having it strewn across different interface elements in the IDE. P4 particularly said that having several related items to his current context allowed him to explore the code base even beyond his current task and get a good understanding of the project's structure. A general observation was that participants were able to browse through recommended elements without losing track of their current context. The RecommendationView enhanced their code exploration without distracting or confusing them with too much information. Only one participant, P7, said that having a new view in the IDE took considerable getting used to and therefore preferred condition 3 with color highlights only.

Merged and aggregated recommendation display received great feedback.

5.4.5 Color Highlights

Color highlights received mixed feedback in terms of attention-grabbing.

Color highlights were available in *Color Highlights* and *Full Framework*. While *Full Framework* featured them in addition to a colored `RecommendationView`, *Color Highlights* implemented only the color highlights to visualize recommendations in the IDE's existing views. Feedback on the color highlights differed based on the participants. Four participants (P1, P4, P9, P19) said that the color highlights did not immediately grab their attention to new recommendations. However, six participants (P2, P3, P5, P6, P13, P14) specifically remarked that the colors did grab their attention – more so than just having the contexts of the `RecommendationView` change. Many of the participants that felt that the color highlights were very helpful to them were participants that had a strong focus on navigating through the file explorer. Even though the highlights can also be seen if they are included in *Global Search* results, the highlights did not have a strong impact on participants using the search functions (as can be seen in Figure 5.19). The fact that the search results did include highlights however was positively received by participants. P3 found the color highlights to be less cognitively demanding than the `RecommendationView` since less information filtering was required. Another thing that was well received by many participants was how the highlights conveyed where clusters of related elements are located in the project by just looking at the explorer. The color highlight propagation through parent and child directories really helped give users a better idea of where certain code elements are located and of the general project structure. P7, a very experienced developer, was surprised by how helpful he found the color highlights in the explorer. His navigation strategy focused on explorer use anyway and he found the highlights to make his usual habits even more effective.

The colors made the explorer more usable and conveyed a better feeling of related file groups in the project.

Feedback

Color highlights could be out-of-view in the explorer.

One issue that some participants identified was that the color-highlighted recommendations can be out-of-view in the file explorer. Especially in big projects, users might have to scroll up or down to see highlighted files. Compared to the aggregated `RecommendationView` display, the information visualized solely through color highlights can be distributed over a larger area in the UI. This may lead to users potentially missing valuable recommended information. P3 suggested markers that show up at the top or bottom of the explorer to indicate that there are further recommendations in that scroll direction of the explorer. This however may prove to be impossible to implement in the current state of the VS Code extension API. For the future of the approach (and other minimally intrusive list-less RSSE approaches) however, it could be a valuable insight. P2 wished for the colors of the file decoration icons to match the color highlights of that file. That feature has since been implemented after the study finished.

Chapter 6

Conclusion

6.1 Summary, Discussion, Contributions

In this thesis, we have investigated *if*, *how*, and to what extent RSSEs can improve navigation and code comprehension for developers in unfamiliar code environments. We started by discussing the issue of source code navigation and presenting promising navigation support approaches. Based on our findings, we conceptualized and implemented a framework that integrates RSSE functionality into the VS Code IDE. The framework combines structural recommendations from the call graph with evolutionary recommendations from other developers' interaction histories with the code. Proactive recommendations of other potentially relevant code pieces in the project are generated for the developer, based on their current work context. We have hypothesized that two of the main reasons why RSSEs – despite their continuous improvement in recommendation accuracy – have not seen adoption into modern IDEs are 1) lack of HCI / UI-integration research and 2) insufficient flat-list output modes. Therefore, in addition to a tree-view recommendation output, we enriched the IDEs UI with color highlights to mark currently recommended documents. In order to evaluate whether our approach can improve code navigation and comprehension for newcomers to projects, we conducted a user study. The user study simulated participants' first software maintenance and debugging tasks on a previously unknown code base. To get a better idea of what exactly helps them with navigation and comprehension we tested three different implementations of the framework as well as basic VS Code.

6.1.1 Discussion

As we have presented in Chapter 5, our measures of *Success Rates*, *Completion Times*, *Support Satisfaction*, and *Code Comprehension* have shown promising results for our recommendation framework. All three conditions that implemented parts of the framework showed clear improvements over the basic IDE in all conducted measures.

The framework enabled users to complete more tasks and do so more efficiently.

On average, participants completed 10% more tasks successfully with condition *Color Highlights*, 15% with condition *Recommendation List*, and 13% with *Full Framework* (see Table 5.2) as compared to *Ground Truth*. Similar improvements can be seen in task completion times (compare Table 5.4). The average task completion times with *Ground Truth* were significantly slower than those for the other three conditions. Both measures of *Success Rates* and *Completion Times* have shown that the framework does indeed support newcomers to unfamiliar code projects in their navigation. Using the framework conditions the participants of our study were able to solve software maintenance and debugging tasks much more efficiently and successfully than using the standard VS Code IDE.

Color Highlights showed great improvments in code comprehension.

Regarding the *Code Comprehension* questions we asked, the participants were able to answer 37% of the answers after using *Ground Truth*, 45% of the answers after using condition *Recommendation List*, 46% of the answers using *Full Framework*, and 56% of the answers after using *Color Highlights* (compare Table 5.5). These improvements are very promising given that participants had only worked on the code project for the first time and for a very short amount of time. Overall, our framework has proven to improve participants' code comprehension and their mental map of the code project. Especially *Color Highlights* seemed to boost the users' awareness of the project structures by a lot.

We have seen improvements in *Support Satisfaction* when participants used our framework compared to when they used basic VS Code (compare Table 5.3). This further suggests the framework's ability to support newcomers in their navigation and attests to the approach's usability. It also shows that there are improvements to be made in the way IDEs handle support navigation and RSSEs can be part of those improvements. *Task Confidence* values however did not significantly improve based on what condition participants were using.

Users were much more satisfied with the navigation support they received from the framework.

Our analysis of *Tool Usage* showed that participants were happy to adopt our new tools and did so very efficiently (compare Table 5.6). The frameworks tools were used often and with high success rates. The color highlights also enhanced the way participants used the explorer. With color highlights enabled they were able to navigate successfully using the explorer much more often.

Our tools saw great adoption by the users.

6.1.2 Contributions

Overall, we make the following contributions with this thesis:

- We have proposed of a novel approach of merging structural recommendations from the call graph with evolutionary recommendations from collaborative interaction histories (**G0**).
- We have shown that by integrating such recommendations into the IDE we can support developers in their code navigation in unfamiliar environments (**G1**).
- Color highlights marking recommended files in the IDE enabled users to gain better understanding and mental maps of a new code base (**G2**, **G4**).
- Developers are willing to adopt RSSEs into their workflow if they are integrated well into the IDEs UI. Users are more satisfied with the received navigation support when RSSE elements are present in the IDE (**G3**).

6.2 Limitations

There are a few limitations to our study that might affect the general applicability of its results to real-world scenarios. The first one could be argued to not be much of a limitation but rather a direct implication of our research goals.

The study was limited to developers in unfamiliar code environments.

Naturally, the findings of this first study are limited to developers working on unfamiliar code bases. This was clearly formulated in the thesis goals and the study was designed and conducted accordingly. Issues in code navigation and comprehension are typical for but not exclusive to newcomers to projects. Still, it is important to note that the promising results of this study can not be applied to expert developers working in familiar environments. Further research and studies need to be conducted in order to investigate merged structural and evolutionary recommendations, as well as the *Color Highlights* RSSE approach in expert contexts.

A lab study does not substitute a future field study with real-world data.

Secondly, our study was a lab simulation and therefore bares some differences to real-world scenarios. Although participants were never asked to use any navigation tooling specifically, the study setting may have increased their willingness to adopt the framework's navigation tools. Also, the data used to generate the evolutionary recommendations was not real-world data. Even though it was constructed in a realistic fashion, inspired by actual state-of-the-art recommendation miners, it presupposed a rich interaction history and good recommendation results. Concerning our research goals, however, this should have no effect on our findings. We set out to investigate if a well-working RSSE engine combined with structural recommendations and a user-friendly interface can support developers in navigating and understanding unfamiliar code projects. Collecting actual interaction data and mining rule-based recommendations was out-of-scope for this thesis. Advancements on the technical side of RSSE research in recent years motivated our research and suggest that the missing HCI components could be one of the reasons we have not seen RSSE adoption in IDEs.

Lastly, the developed framework and our evaluation do not consider other possible IDE paradigms like those presented in Section 2.2.1. We have only investigated the effect of well-integrated RSSE functionalities, the merging of different recommendation sources, and color highlights in a standard, window, and file-based IDE.

No other IDE paradigms were considered in the study.

6.3 Future work

To conclude this thesis we will present some interesting future work that could follow this thesis and remains to be done. We think that the field of recommendation systems in software engineering has been neglected, especially from an HCI perspective, and offers exciting research opportunities. Especially with the recent developments in data-driven technologies – also for developer assistance – it is to be expected that the technical side and recommendation accuracy of navigation support systems will further improve. These advances call for more human-centered research on how these technologies can be made accessible and useful.

For the developed framework several things remain to be explored:

The user feedback should be incorporated into the framework.

- As often suggested by study participants, color highlights should also mark the recommended code blocks within the files and allow users to directly jump to them.
- Color Highlights could be extended to encode further information such as the relevance of recommended artifacts.
- Recommendations in the `RecommendationView` could be enriched with code summaries.

Finally, and most importantly, a field study should be conducted that deploys the framework in a real-world software development environment. By tracking the interaction histories of a development team and using a state-of-

A field study in a real work environment should be conducted.

the-art recommendation engine the framework can be evaluated for developers working on familiar code bases. Code navigation issues may often concern newcomers to projects but do not at all exclude experts. It would be interesting to see how experts perceive the navigation support of the framework compared to their regular work environment. We hypothesize that well-integrated RSSE interfaces could help reduce navigation times and errors in development. Additionally, we suspect that listless interface solutions (as condition *Color Highlights* in our study) could be even more beneficial for expert usage than for newcomers.

We are excited to see what future research on developer support through recommendation systems will bring and hope to see further research on how to efficiently integrate these systems into the developer's work environment.

Appendix A

Excursus: Evolutionary Couplings and Association Rule Mining

Logical or evolutionary couplings are dependencies between code elements that are revealed by how a system evolves over time Hassan and Holt [2004], Canfora and Cerulo [2005], Gall et al. [1998, 2003]. Insight into a system's evolution can be gained through the commit histories of a repository or through the developers' programming interaction with the system. Two code elements may be coupled evolutionary if they are frequently co-changed together or impacted by similar changes. These couplings differ from standard structural couplings that can be discovered using program analysis as they are not based on how system components are interconnected. As these couplings can escape structural program analysis such as call hierarchy tools, they can provide valuable information on code dependencies in software projects. Understanding and analyzing evolutionary couplings can help improve a project's maintainability by shifting focus to code fragments that are more reliant or more influential on other areas of code. Identifying a close evolutionary coupling between two elements means that changing one will likely affect the other one's behavior as well. Gaining insight into those often hidden dependencies can reduce the likeliness of bugs being introduced into the code. As a result, evolutionary couplings have played a large role in software change impact analysis and tools that aim to support developers in estimating these impacts (Briand et al. [1999], Hassan and Holt [2004], Canfora and Cerulo [2005], Canfora et al. [2010], Rolfsnes et al. [2016]).

Association Rule Mining

The most common way of mining evolutionary couplings is through the technique of association rule mining Agrawal et al. [1993]. Association rules are strong regularity links in a transaction data set based on observations of that data set. Mining association rules typically involves the counting and analysis of co-occurrences and the deduction of rules from co-occurrences of items in transactions. The order of items in the transactions is irrelevant, which marks the most important distinction to sequence mining. The most famous example is that of a supermarket tracking all purchases of customers. Each purchase can be considered a transaction containing all items purchased in that purchase. By analyzing the transactions we can determine regularities in the shopping behavior of customers and formulate rules such as "A transaction containing both beer and peanuts is likely to also contain chips.". This would translate to the association rule $Beer, Peanuts => Chips$. To make statements about association rules we can use the concepts of support and confidence. Consider we have two itemsets in $X = Beer, Peanuts, Y = Chips$ in our database Support indicates how frequently an itemset is observed in the data set.

$$support(X) = P(Beer \cap Peanuts)$$
$$= \frac{\text{number of transactions containing both Beer and Peanuts}}{\text{number of all transactions}}$$

The confidence of our rule $X => Y$ is the percentage of transactions that satisfy X that also satisfy Y.

$$confidence(X => Y) = P(Y|X) = \frac{support(X,Y)}{support(X)}$$

In summary, association rules can tell us about regularities in our data and how relevant and frequent they are. Association rule miners deploy algorithms to efficiently identify those association rules in a data set that satisfy given thresholds of minimum support and confidence. Applied to interaction data or version histories of software projects, association rules can reveal files or finer code elements that have frequently been changed together.

Approaches in mining evolutionary couplings can generally be divided into two groups. Ones that mine couplings from commits (version control systems) and the ones that mine couplings from developer interaction data Bantelay et al. [2013].

Besides change impact analysis, a popular application for evolutionary couplings are Recommendation Systems in Software Engineering (RSSEs). Given that evolutionary couplings can be mined from developer interactions in the IDE, it is not far-fetched to assume that they can in turn also guide developers in their interactions. New developers can benefit by navigating along evolutionary couplings in the code that were learned from experienced developers' interactions with it. And software tasks such as debugging and maintenance could be improved and less error-prone with tools that recommend code elements that are tightly coupled.

Appendix B

Interlude: *Mylyn* Interaction Data

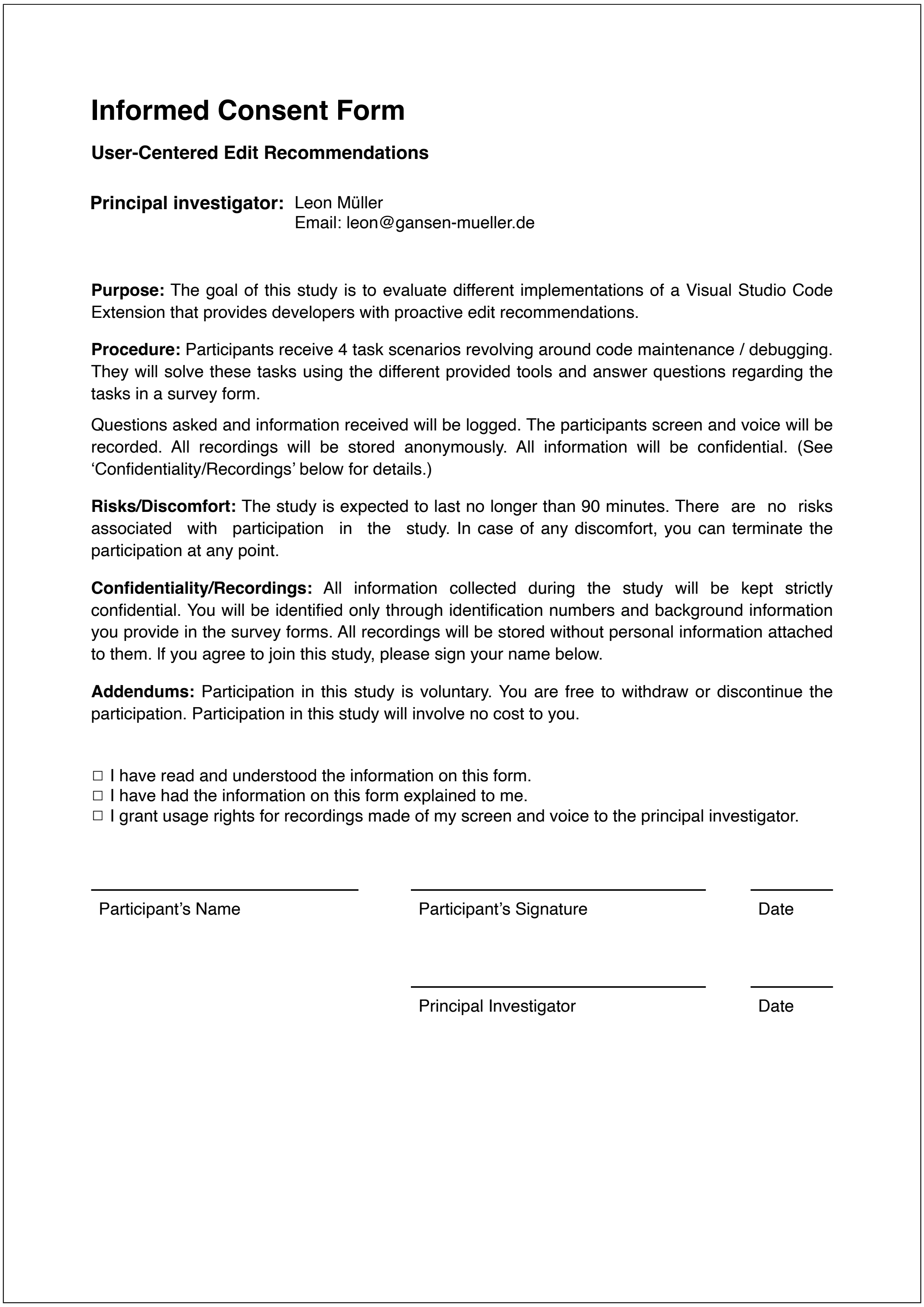
Mylyn (Kersten and Murphy [2005]) interaction traces contain interaction events of different kinds: *selection, edit, propagation, command, ...* A selection event is triggered upon file opening and tab switching. An edit event is triggered when either a textual or graphical edit has occurred (Kersten and Murphy [2006]).

Yamamori et al. [2017] argue that in actual fact, edit events are recorded in *Mylyn* whenever the cursor position is in the editor. Even when no actual change has happened, an edit event is logged in the interaction trace. In a study on noise in interaction traces, Soh et al. [2015] found that *Mylyn* traces miss an average of 6% of time spent on tasks and contain on average about 28% of false edit events. A later study by the same authors (Soh et al. [2018]) revised the percentage of false edit events in *Mylyn* traces to a staggering 75% to 85%. The authors discuss the effects on previous studies on developer behavior and the performance studies of RSSEs such as *MI*. To quantify these effects they propose a technique to correct *Mylyn* trace and clean out the false edit events. They reconducted the study performed on *MI* by Lee et al. [2015] both for noisy *Mylyn* traces and for corrected traces without the false edit events. The results showed that using the corrected data, *MI*'s recommendation precision, recall, and f-measure improved significantly with the cleaned traces (Precision increased by up to 56% for one particular project). The average precision increased to about 89%. This further supports the technical viability of navigational RSSEs. Future interaction loggers however need to address the noise issues identified in these studies.

Appendix C

User Study Documents

During the user study discussed in chapter 5, the participants were handed various documents. C.1 contains the informed consent form all participants read and signed prior to the study. The introduction to the study setting, as well as all task descriptions and survey elements filled out by the participants, is shown from C.2 to C.18.



Informed Consent Form

User-Centered Edit Recommendations

Principal investigator: Leon Müller
Email: leon@gansen-mueller.de

Purpose: The goal of this study is to evaluate different implementations of a Visual Studio Code Extension that provides developers with proactive edit recommendations.

Procedure: Participants receive 4 task scenarios revolving around code maintenance / debugging. They will solve these tasks using the different provided tools and answer questions regarding the tasks in a survey form.

Questions asked and information received will be logged. The participants screen and voice will be recorded. All recordings will be stored anonymously. All information will be confidential. (See ‘Confidentiality/Recordings’ below for details.)

Risks/Discomfort: The study is expected to last no longer than 90 minutes. There are no risks associated with participation in the study. In case of any discomfort, you can terminate the participation at any point.

Confidentiality/Recordings: All information collected during the study will be kept strictly confidential. You will be identified only through identification numbers and background information you provide in the survey forms. All recordings will be stored without personal information attached to them. If you agree to join this study, please sign your name below.

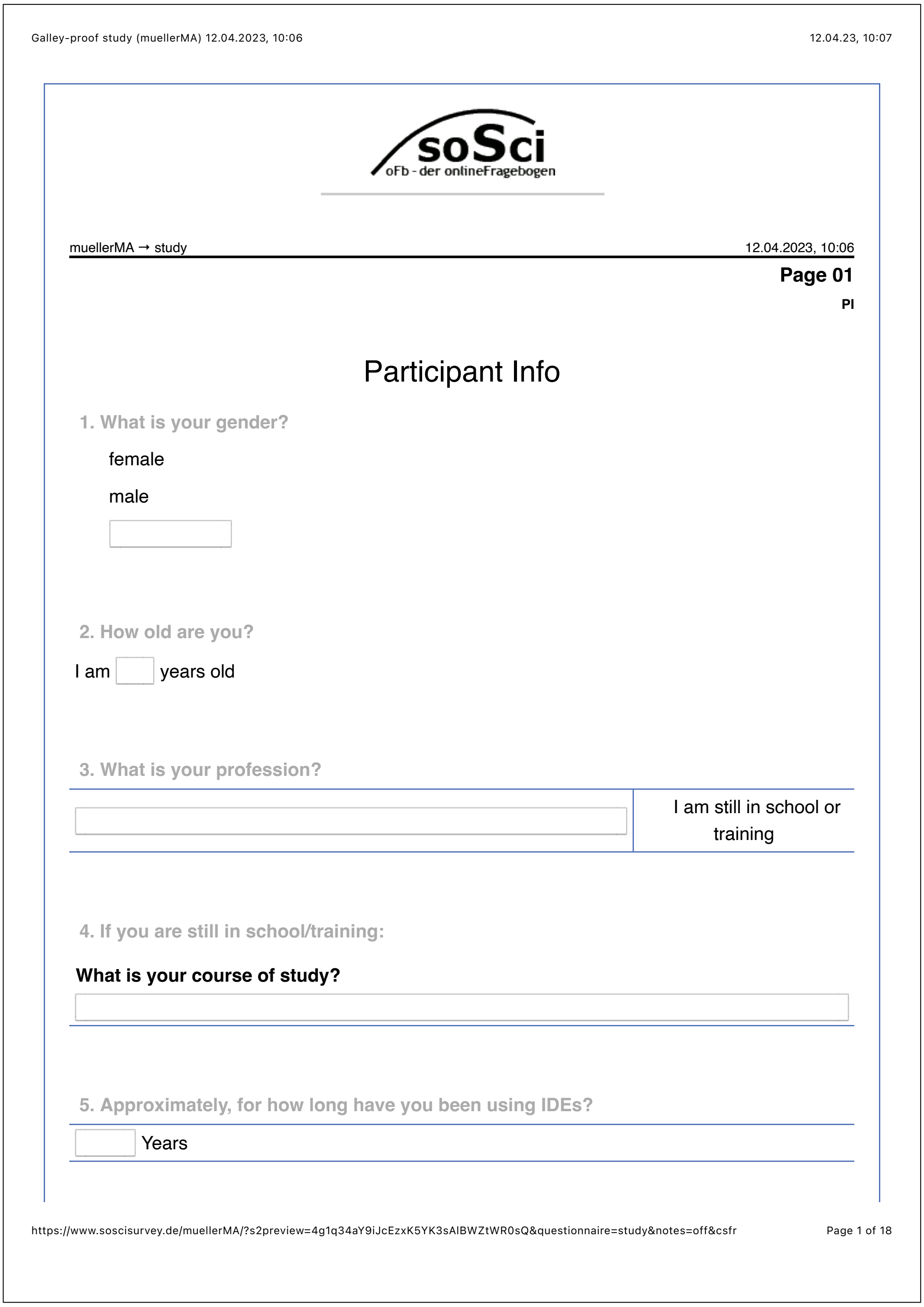
Addendums: Participation in this study is voluntary. You are free to withdraw or discontinue the participation. Participation in this study will involve no cost to you.

□ I have read and understood the information on this form.
□ I have had the information on this form explained to me.
□ I grant usage rights for recordings made of my screen and voice to the principal investigator.

Participant’s Name Participant’s Signature Date

Principal Investigator Date

Figure C.1: The informed consent form received by all participants.



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

muellerMA → study 12.04.2023, 10:06

Page 01

PI

Participant Info

1. What is your gender?

female

male

2. How old are you?

I am years old

3. What is your profession?

	I am still in school or training

4. If you are still in school/training:

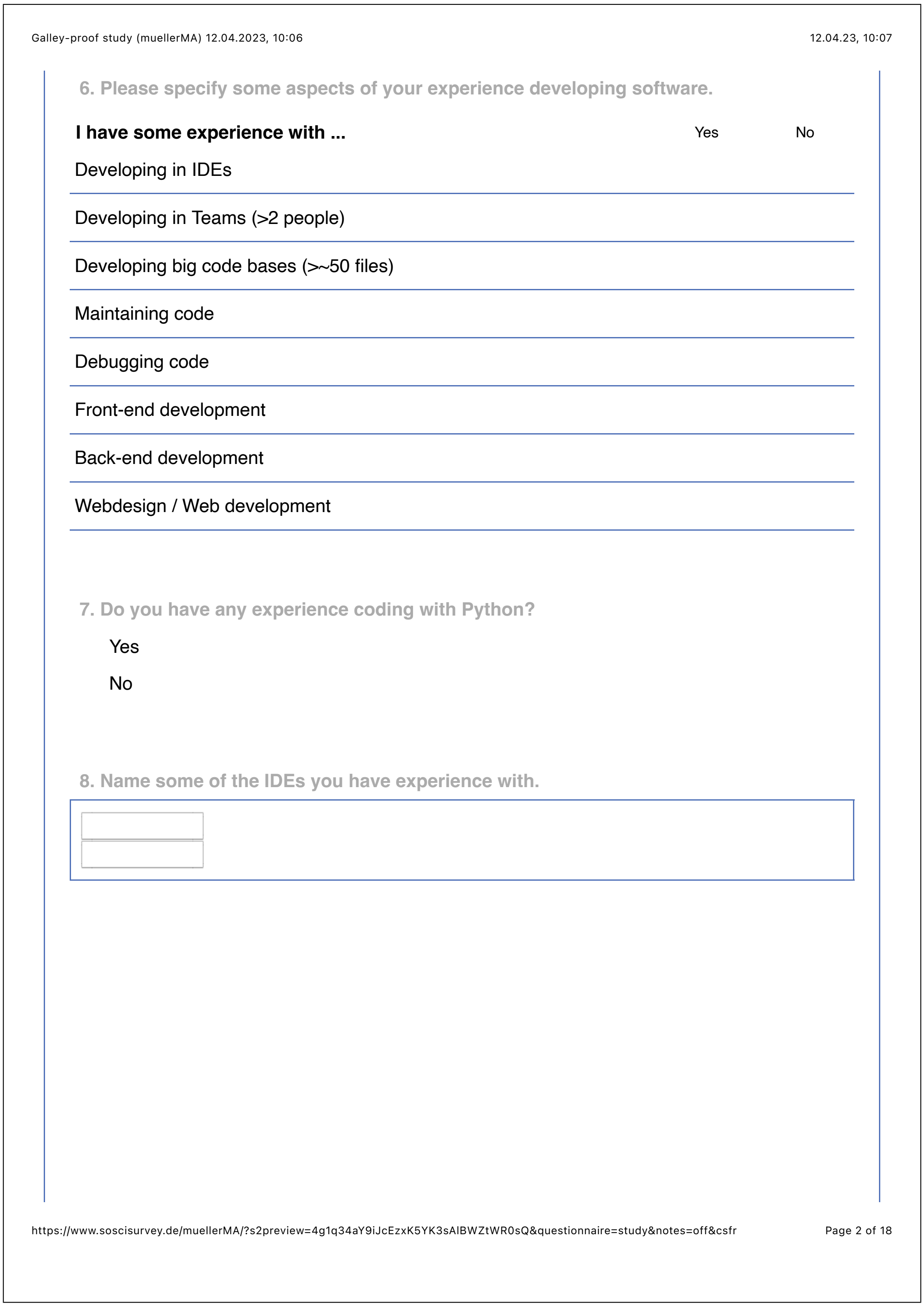
What is your course of study?

5. Approximately, for how long have you been using IDEs?

Years

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 1 of 18

Figure C.2: The survey handed to and filled out by all participants.



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

6. Please specify some aspects of your experience developing software.

I have some experience with ...	Yes	No
Developing in IDEs		
Developing in Teams (>2 people)		
Developing big code bases (>~50 files)		
Maintaining code		
Debugging code		
Front-end development		
Back-end development		
Webdesign / Web development		

7. Do you have any experience coding with Python?

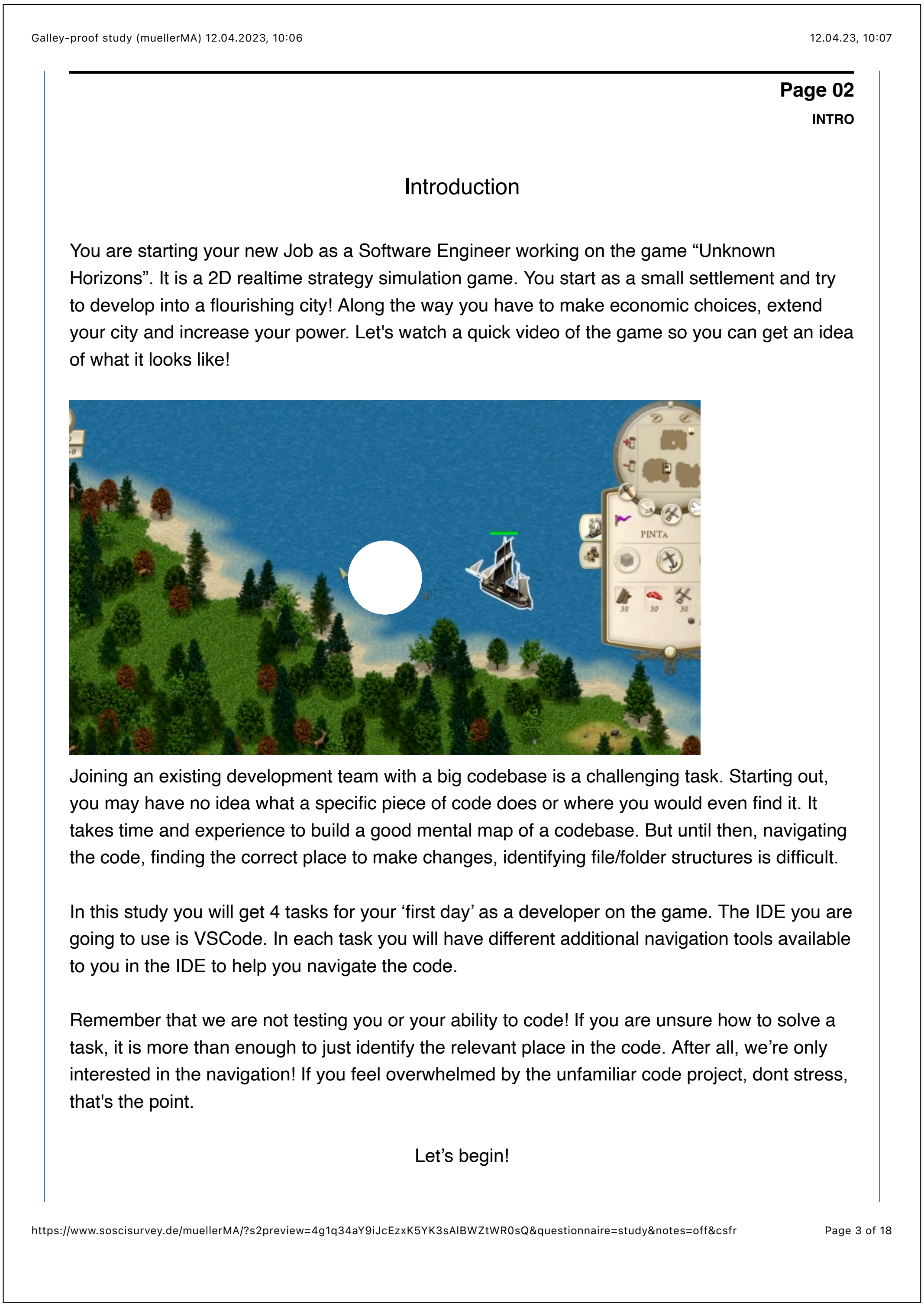
Yes

No

8. Name some of the IDEs you have experience with.

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 2 of 18

Figure C.3



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 02

INTRO

Introduction

You are starting your new Job as a Software Engineer working on the game “Unknown Horizons”. It is a 2D realtime strategy simulation game. You start as a small settlement and try to develop into a flourishing city! Along the way you have to make economic choices, extend your city and increase your power. Let's watch a quick video of the game so you can get an idea of what it looks like!

Joining an existing development team with a big codebase is a challenging task. Starting out, you may have no idea what a specific piece of code does or where you would even find it. It takes time and experience to build a good mental map of a codebase. But until then, navigating the code, finding the correct place to make changes, identifying file/folder structures is difficult.

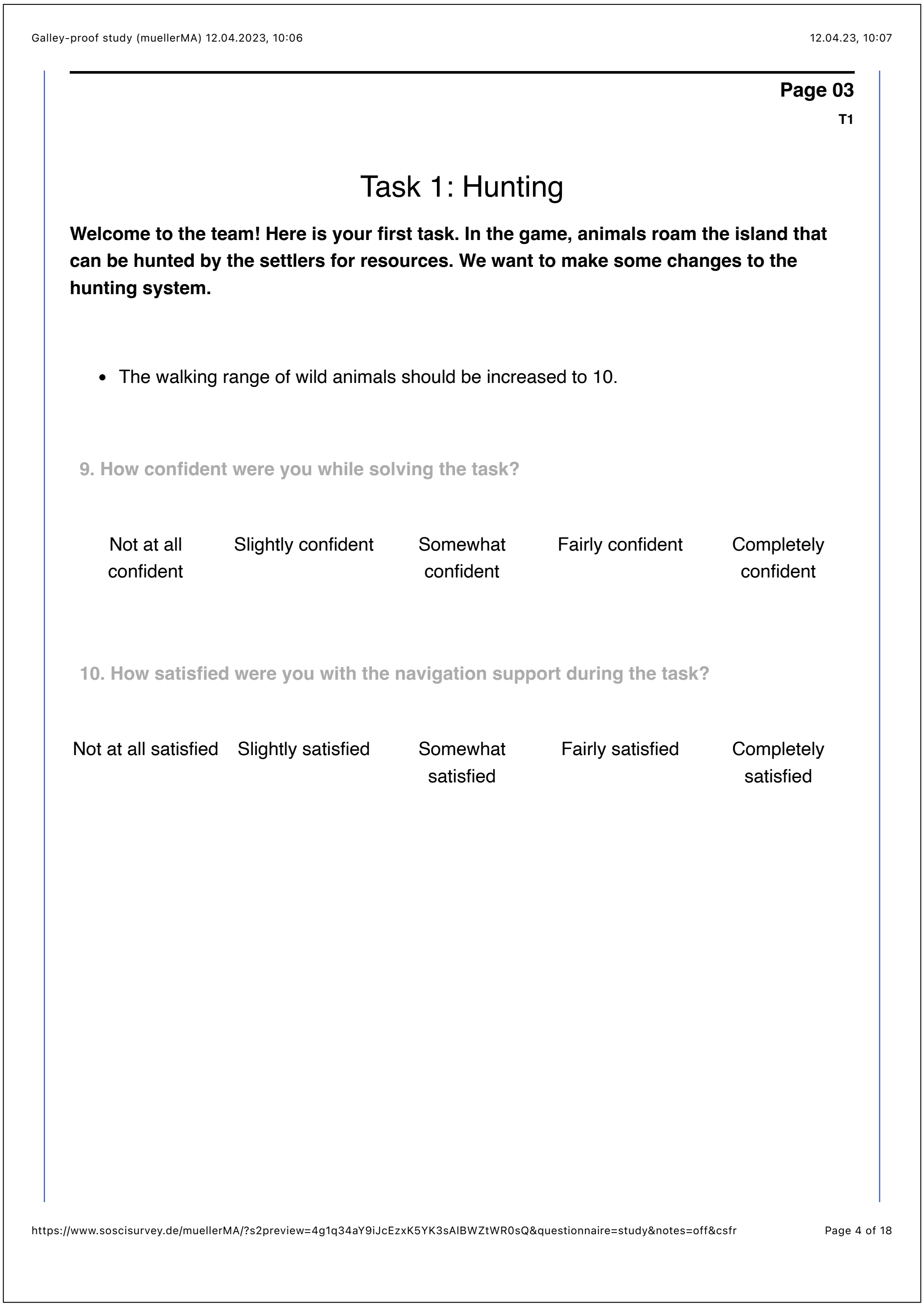
In this study you will get 4 tasks for your ‘first day’ as a developer on the game. The IDE you are going to use is VSCode. In each task you will have different additional navigation tools available to you in the IDE to help you navigate the code.

Remember that we are not testing you or your ability to code! If you are unsure how to solve a task, it is more than enough to just identify the relevant place in the code. After all, we’re only interested in the navigation! If you feel overwhelmed by the unfamiliar code project, dont stress, that's the point.

Let’s begin!

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 3 of 18

Figure C.4



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 03

T1

Task 1: Hunting

Welcome to the team! Here is your first task. In the game, animals roam the island that can be hunted by the settlers for resources. We want to make some changes to the hunting system.

- The walking range of wild animals should be increased to 10.

9. How confident were you while solving the task?

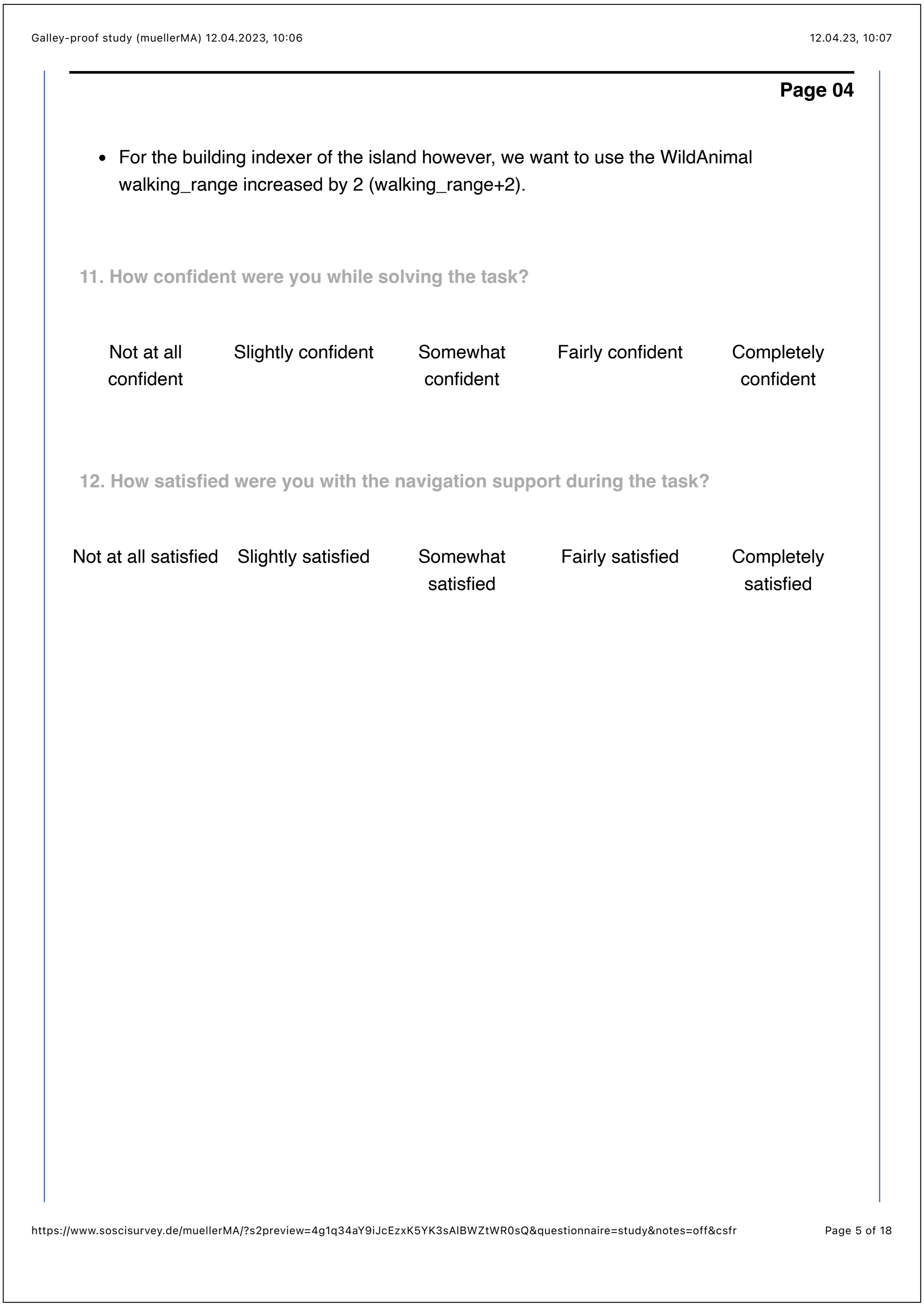
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

10. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 4 of 18

Figure C.5



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 04

- For the building indexer of the island however, we want to use the WildAnimal walking_range increased by 2 (walking_range+2).

11. How confident were you while solving the task?

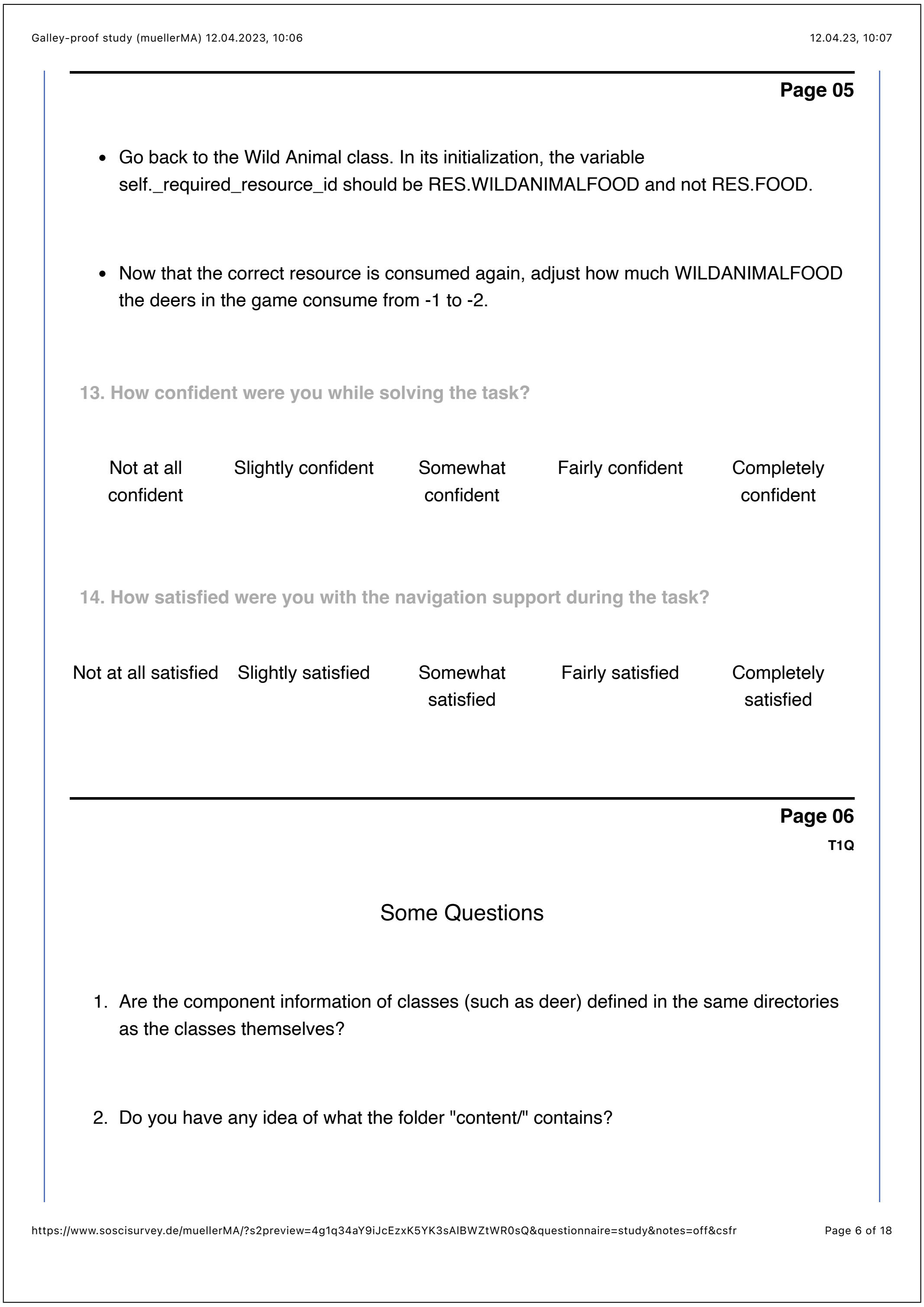
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

12. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 5 of 18

Figure C.6



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 05

- Go back to the Wild Animal class. In its initialization, the variable self._required_resource_id should be RES.WILDANIMALFOOD and not RES.FOOD.

- Now that the correct resource is consumed again, adjust how much WILDANIMALFOOD the deers in the game consume from -1 to -2.

13. How confident were you while solving the task?

Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

14. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

Page 06
T1Q

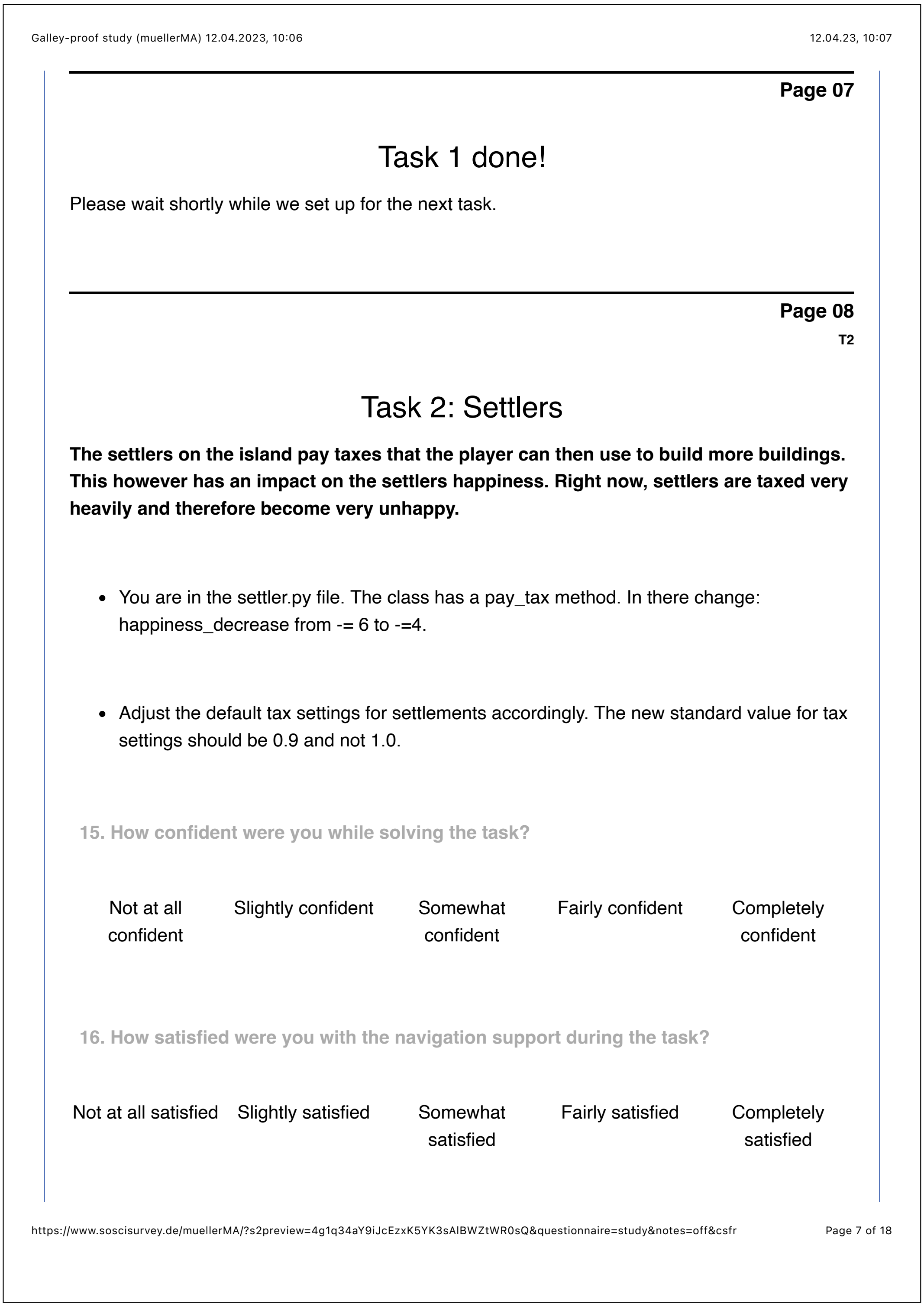
Some Questions

1. Are the component information of classes (such as deer) defined in the same directories as the classes themselves?

2. Do you have any idea of what the folder "content/" contains?

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 6 of 18

Figure C.7



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 07

Task 1 done!

Please wait shortly while we set up for the next task.

Page 08

T2

Task 2: Settlers

The settlers on the island pay taxes that the player can then use to build more buildings. This however has an impact on the settlers happiness. Right now, settlers are taxed very heavily and therefore become very unhappy.

- You are in the settler.py file. The class has a pay_tax method. In there change: happiness_decrease from -= 6 to -=4.

- Adjust the default tax settings for settlements accordingly. The new standard value for tax settings should be 0.9 and not 1.0.

15. How confident were you while solving the task?

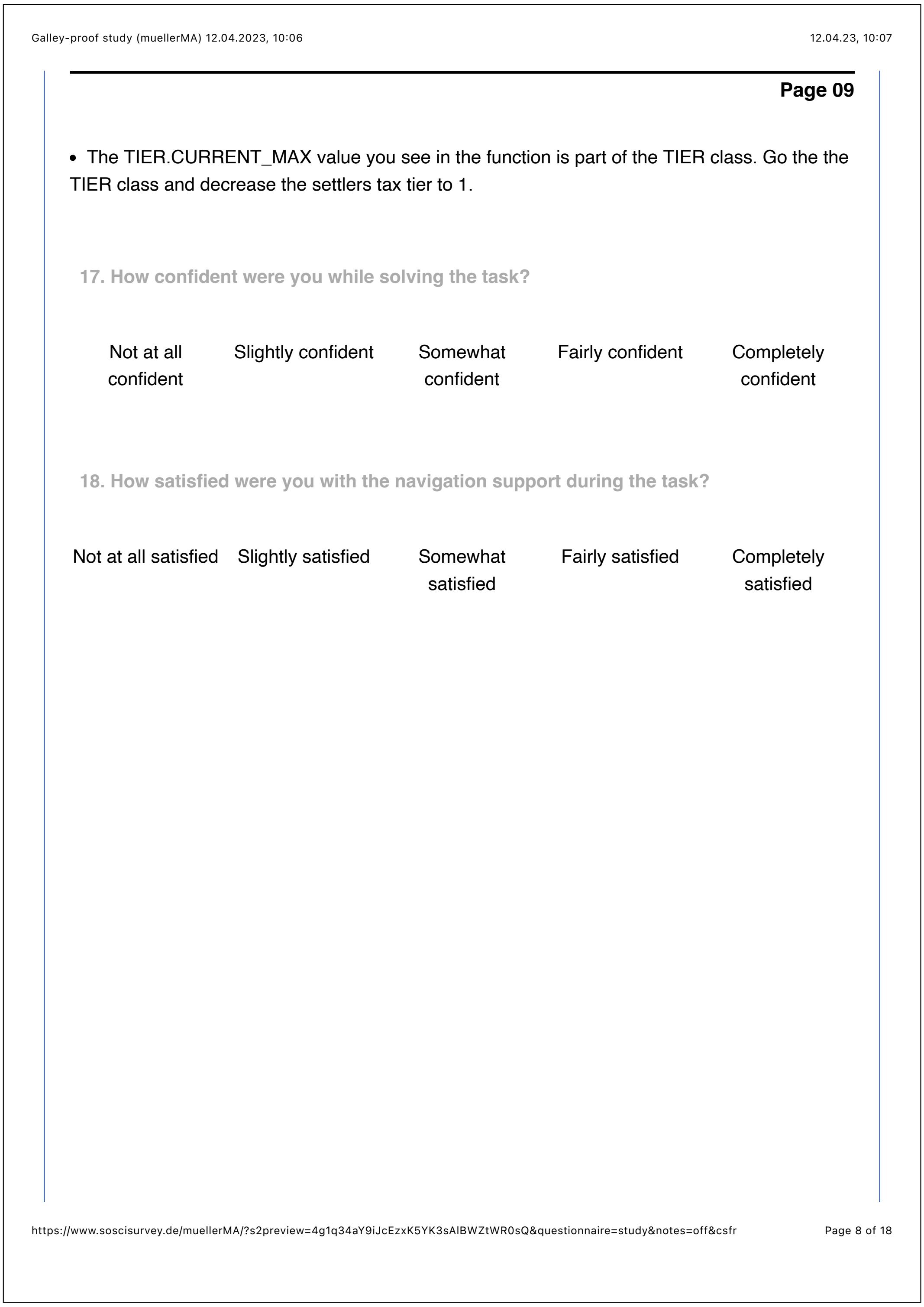
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

16. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 7 of 18

Figure C.8



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 09

- The TIER.CURRENT_MAX value you see in the function is part of the TIER class. Go the the TIER class and decrease the settlers tax tier to 1.

17. How confident were you while solving the task?

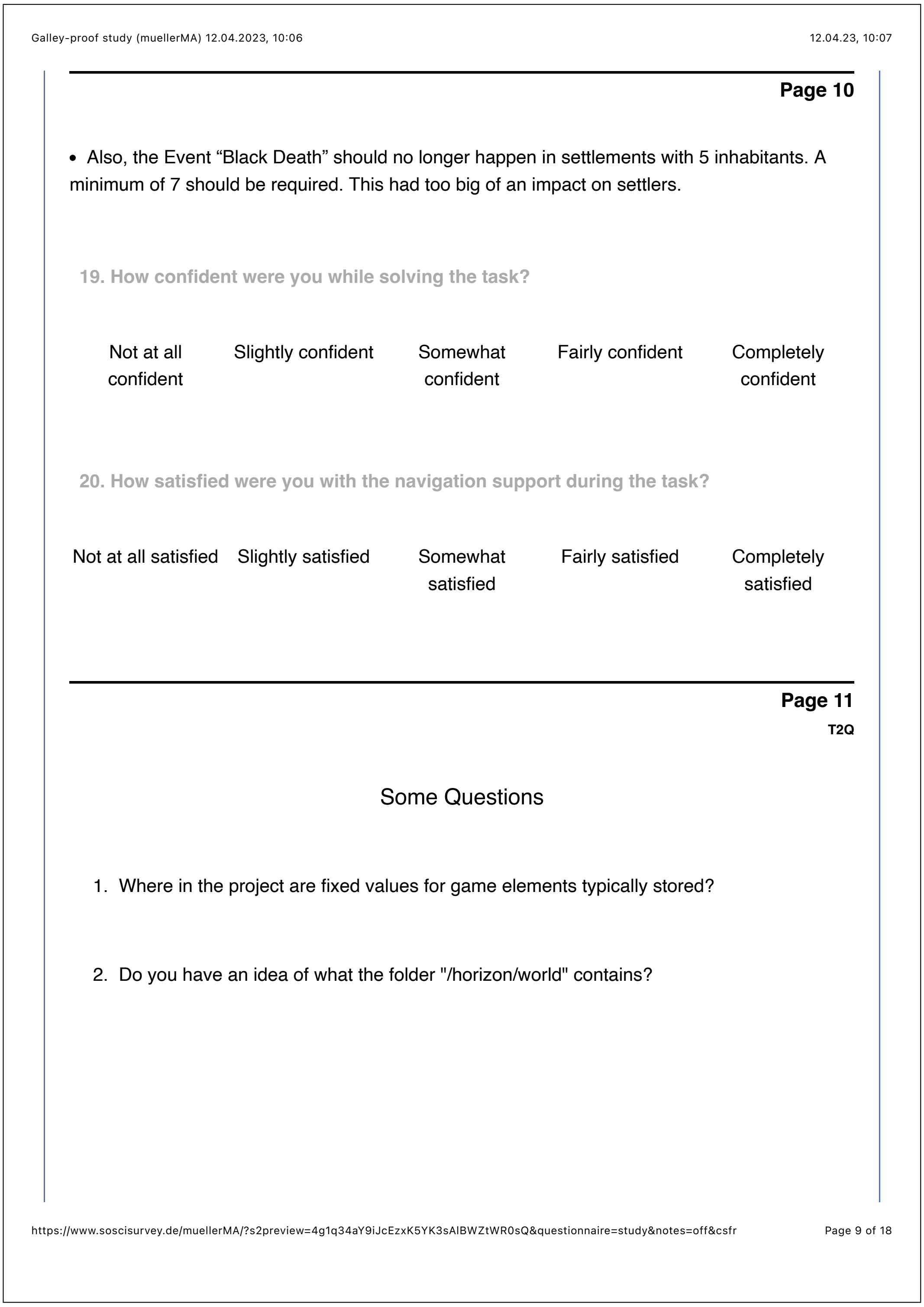
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

18. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 8 of 18

Figure C.9



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 10

- Also, the Event “Black Death” should no longer happen in settlements with 5 inhabitants. A minimum of 7 should be required. This had too big of an impact on settlers.

19. How confident were you while solving the task?

Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

20. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

Page 11

T2Q

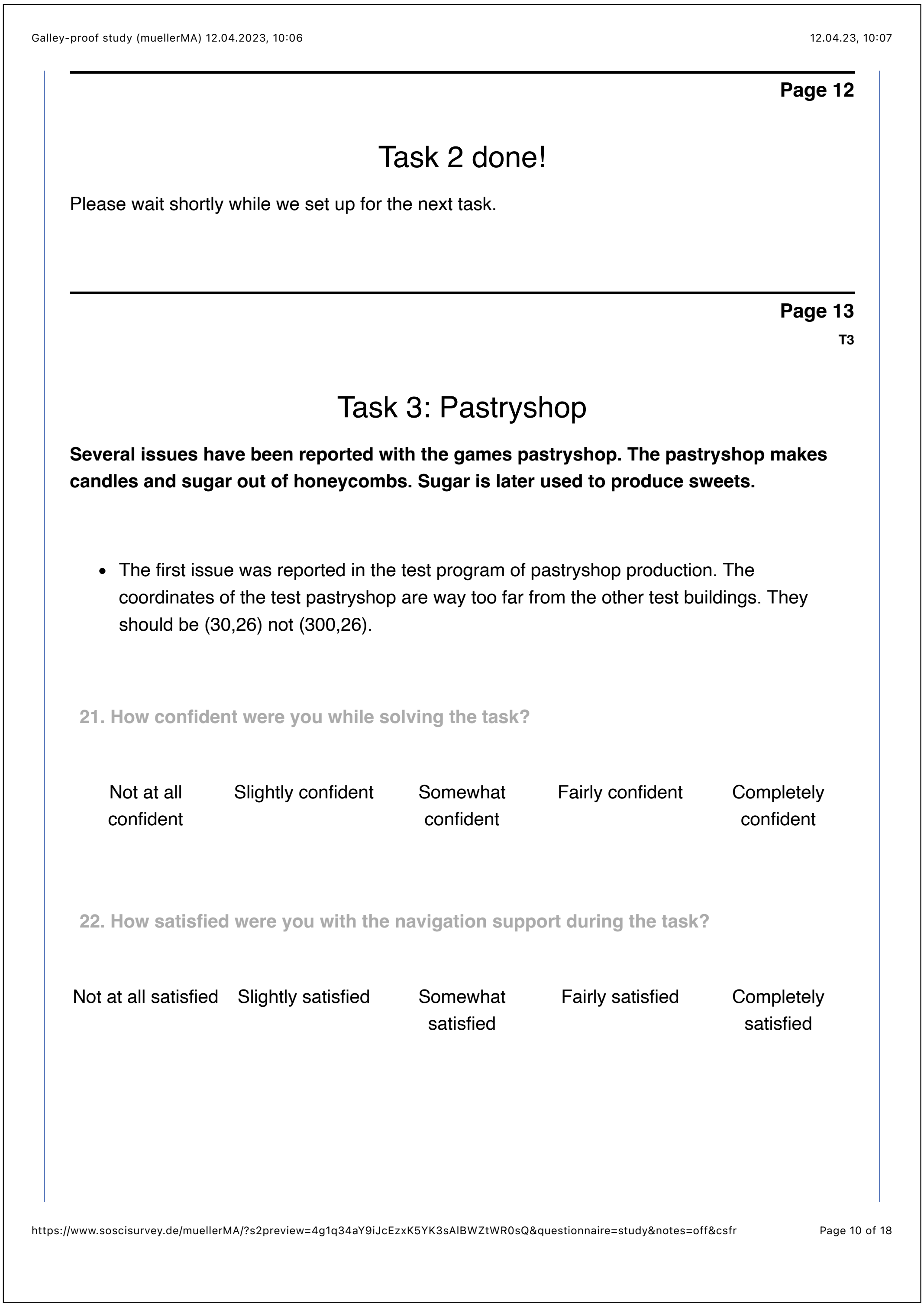
Some Questions

1. Where in the project are fixed values for game elements typically stored?

2. Do you have an idea of what the folder "/horizon/world" contains?

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 9 of 18

Figure C.10



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 12

Task 2 done!

Please wait shortly while we set up for the next task.

Page 13

T3

Task 3: Pastryshop

Several issues have been reported with the games pastryshop. The pastryshop makes candles and sugar out of honeycombs. Sugar is later used to produce sweets.

- The first issue was reported in the test program of pastryshop production. The coordinates of the test pastryshop are way too far from the other test buildings. They should be (30,26) not (300,26).

21. How confident were you while solving the task?

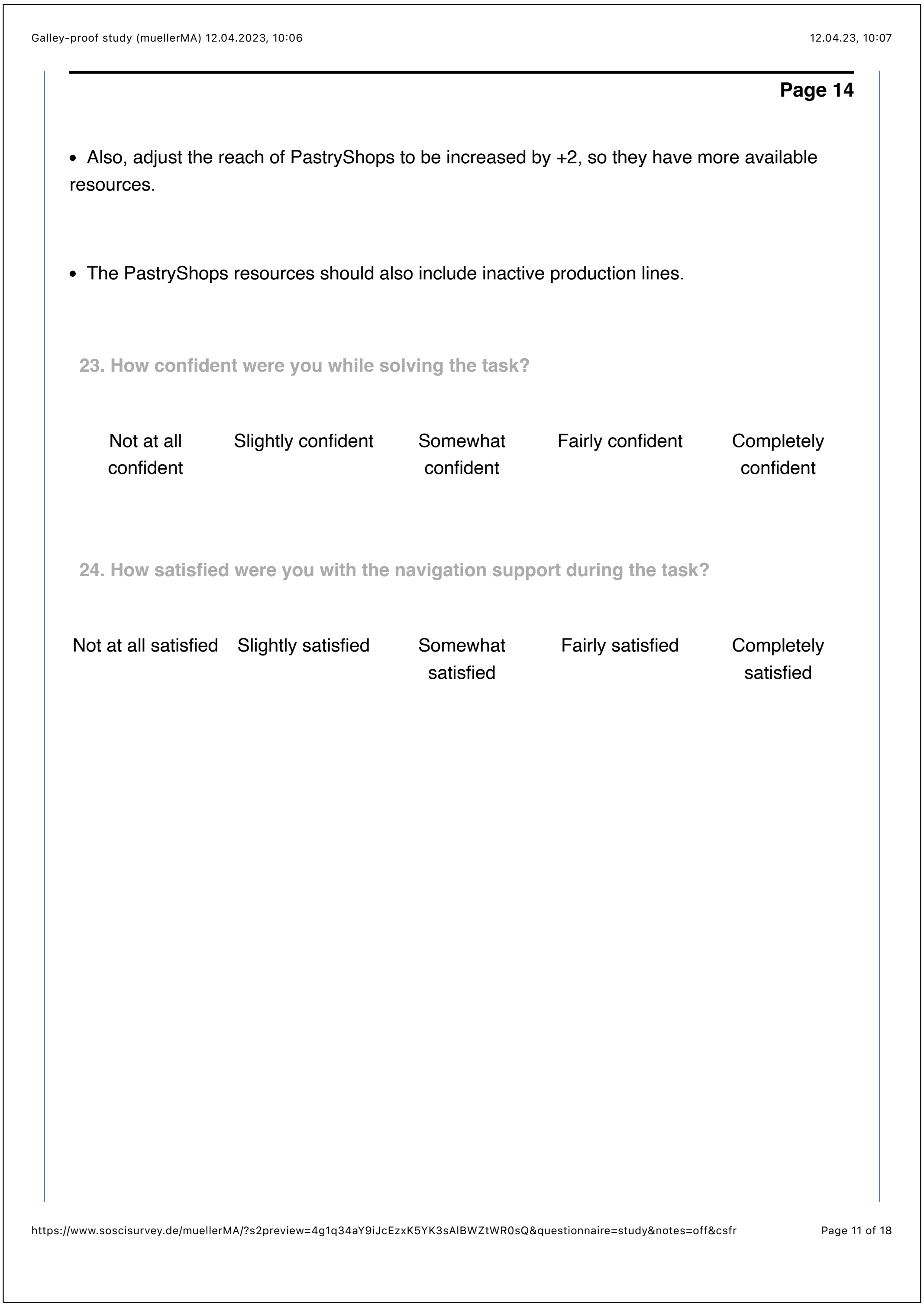
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

22. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 10 of 18

Figure C.11



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 14

- Also, adjust the reach of PastryShops to be increased by +2, so they have more available resources.

- The PastryShops resources should also include inactive production lines.

23. How confident were you while solving the task?

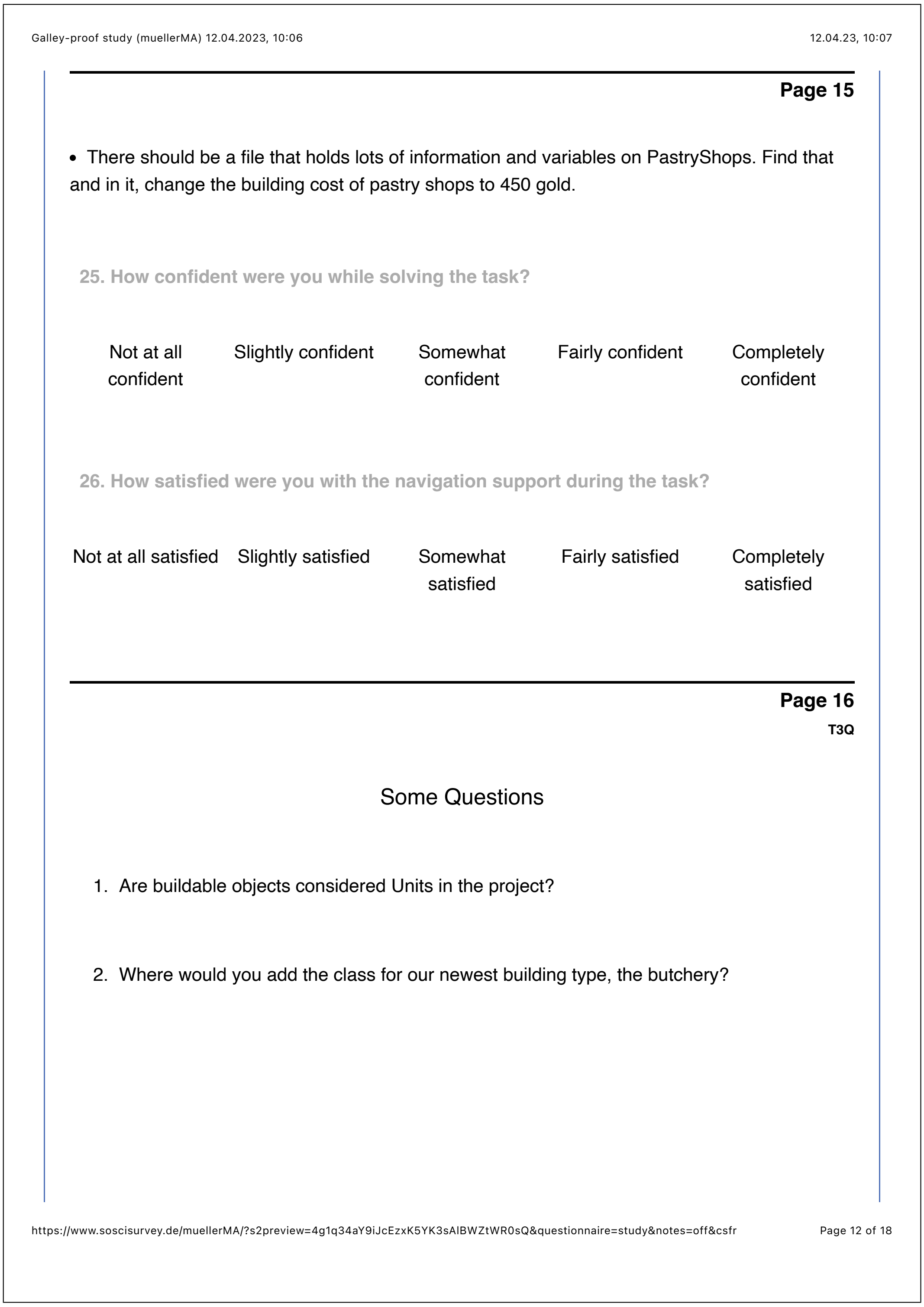
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

24. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 11 of 18

Figure C.12



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 15

- There should be a file that holds lots of information and variables on PastryShops. Find that and in it, change the building cost of pastry shops to 450 gold.

25. How confident were you while solving the task?

Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

26. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

Page 16

T3Q

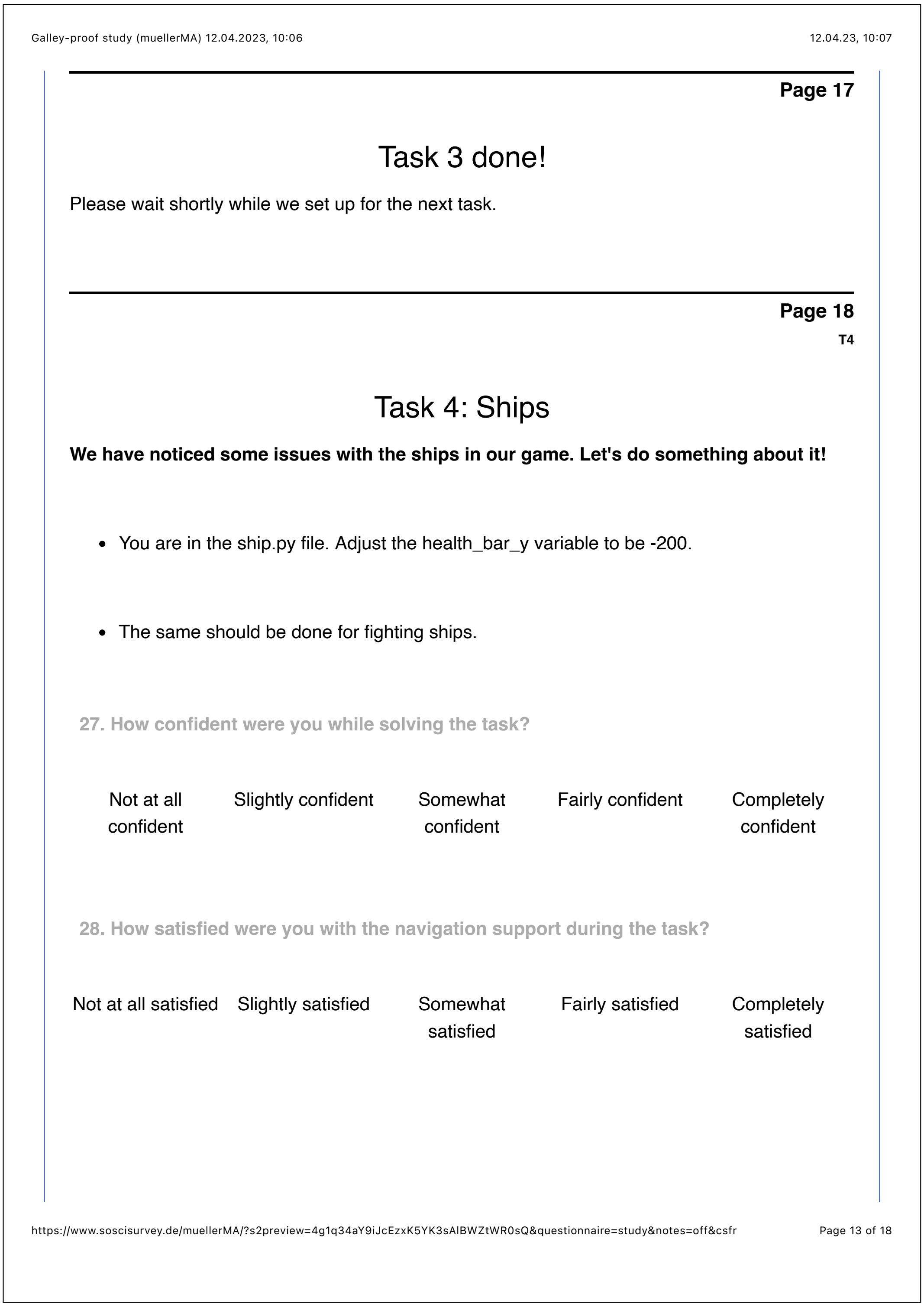
Some Questions

1. Are buildable objects considered Units in the project?

2. Where would you add the class for our newest building type, the butchery?

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 12 of 18

Figure C.13



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 17

Task 3 done!

Please wait shortly while we set up for the next task.

Page 18

T4

Task 4: Ships

We have noticed some issues with the ships in our game. Let's do something about it!

- You are in the ship.py file. Adjust the health_bar_y variable to be -200.

- The same should be done for fighting ships.

27. How confident were you while solving the task?

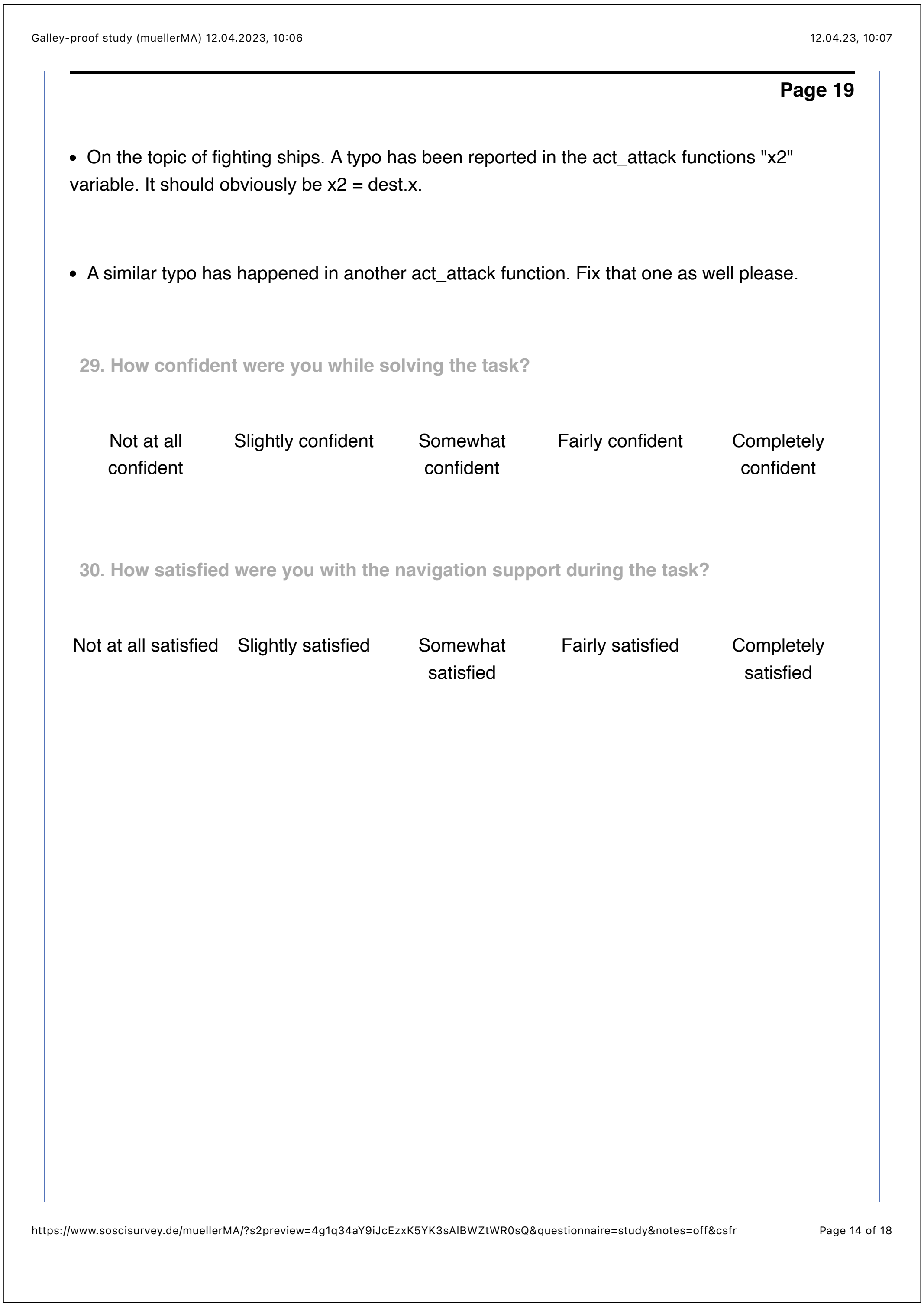
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

28. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 13 of 18

Figure C.14



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 19

- On the topic of fighting ships. A typo has been reported in the act_attack functions "x2" variable. It should obviously be x2 = dest.x.

- A similar typo has happened in another act_attack function. Fix that one as well please.

29. How confident were you while solving the task?

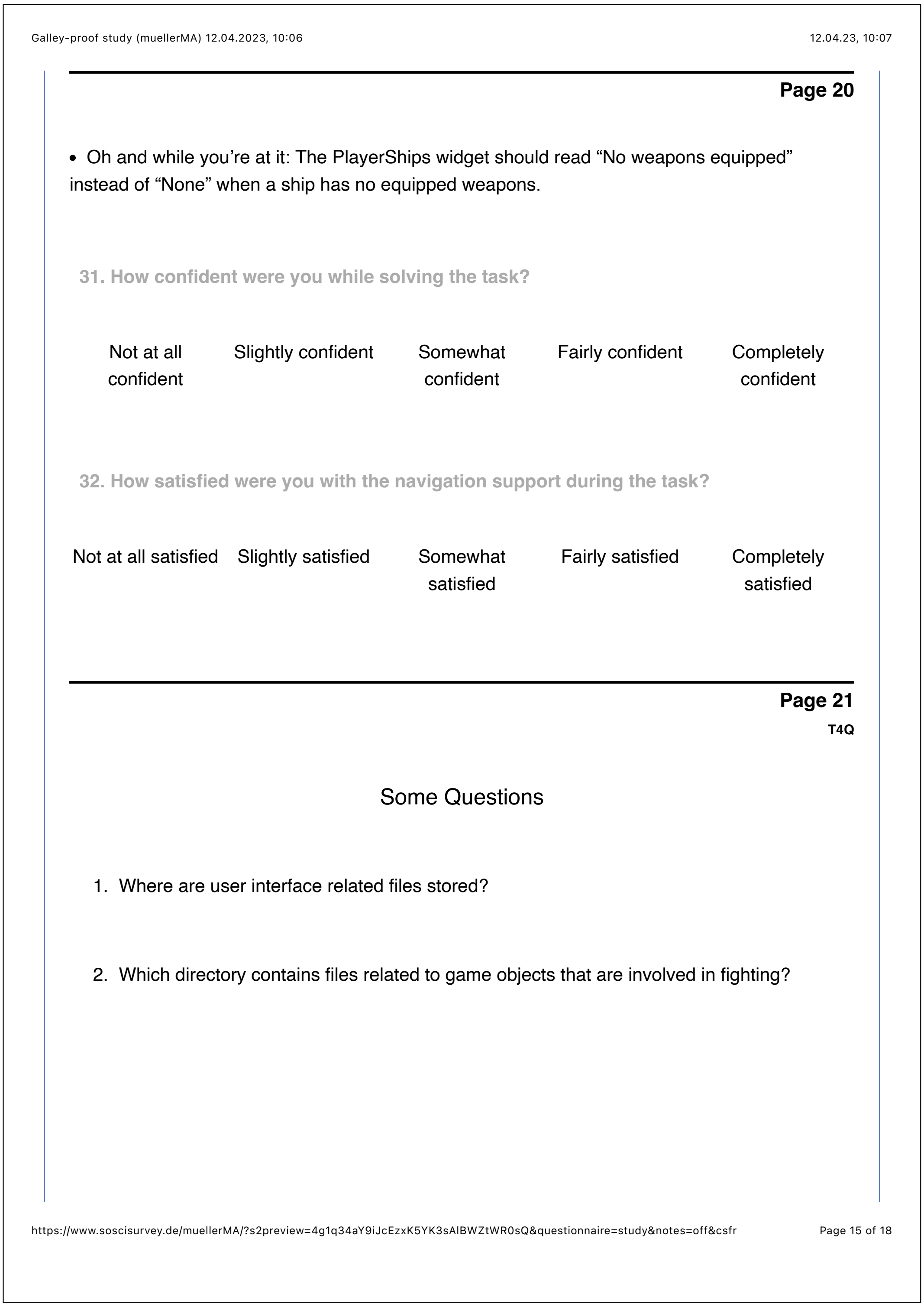
Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

30. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 14 of 18

Figure C.15



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 20

- Oh and while you're at it: The PlayerShips widget should read "No weapons equipped" instead of "None" when a ship has no equipped weapons.

31. How confident were you while solving the task?

Not at all confident	Slightly confident	Somewhat confident	Fairly confident	Completely confident

32. How satisfied were you with the navigation support during the task?

Not at all satisfied	Slightly satisfied	Somewhat satisfied	Fairly satisfied	Completely satisfied

Page 21

T4Q

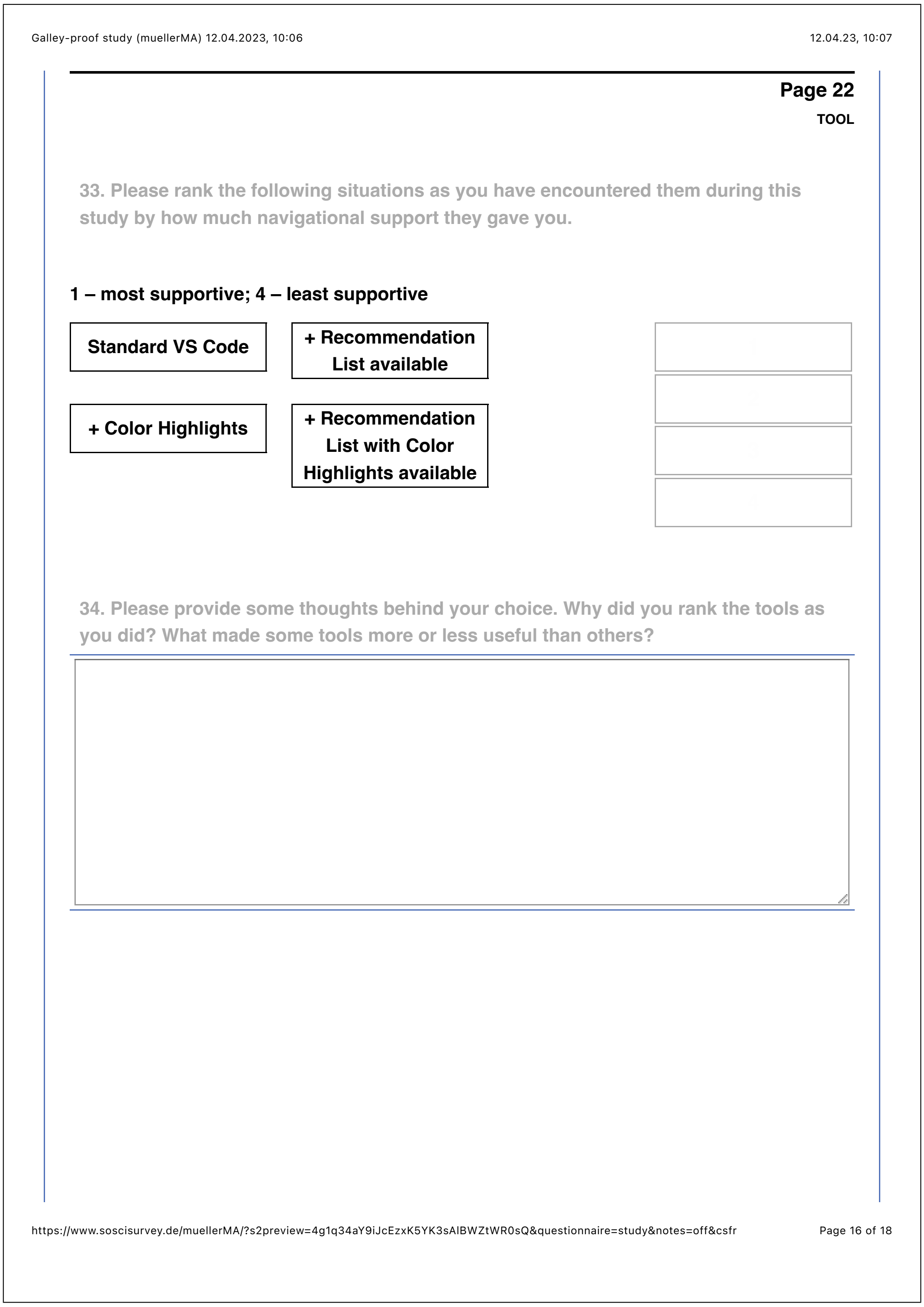
Some Questions

1. Where are user interface related files stored?

2. Which directory contains files related to game objects that are involved in fighting?

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 15 of 18

Figure C.16



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 22

TOOL

33. Please rank the following situations as you have encountered them during this study by how much navigational support they gave you.

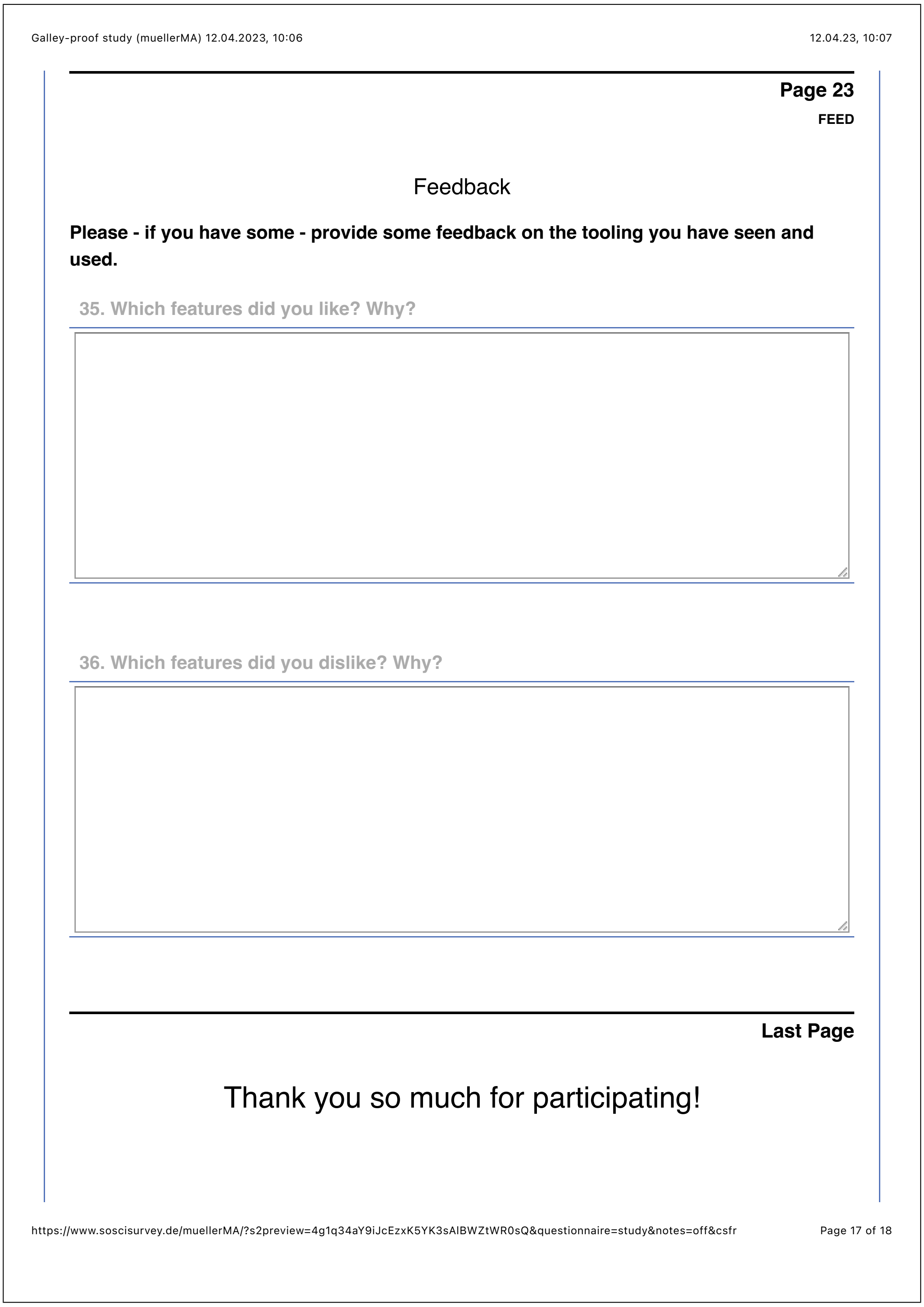
1 – most supportive; 4 – least supportive

- Standard VS Code
- + Recommendation List available
- + Color Highlights
- + Recommendation List with Color Highlights available

34. Please provide some thoughts behind your choice. Why did you rank the tools as you did? What made some tools more or less useful than others?

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 16 of 18

Figure C.17



Galley-proof study (muellerMA) 12.04.2023, 10:06 12.04.23, 10:07

Page 23

FEED

Feedback

Please - if you have some - provide some feedback on the tooling you have seen and used.

35. Which features did you like? Why?

36. Which features did you dislike? Why?

Last Page

Thank you so much for participating!

https://www.soscisurvey.de/muellerMA/?s2preview=4g1q34aY9iJcEzxK5YK3sAlBWZtWR0sQ&questionnaire=study&notes=off&csfr Page 17 of 18

Figure C.18

Bibliography

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 207–216, New York, NY, USA, June 1993. Association for Computing Machinery. ISBN 978-0-89791-592-2. doi: 10.1145/170035.170072.

Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 124–134, Suita, March 2016. IEEE. ISBN 978-1-5090-1855-0. doi: 10.1109/SANER.2016.39.

Fasil Bantelay, Motahareh Bahrami Zanjani, and Huzefa Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 311–320, October 2013. doi: 10.1109/WCRE.2013.6671306. ISSN: 2375-5369.

Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, page 455, Cape Town, South Africa, 2010. ACM Press. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806866.

L.C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems.

In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 475–482, August 1999. doi: 10.1109/ICSM.1999.792645. ISSN: 1063-6773.

G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 9 pp.–29, September 2005. doi: 10.1109/METRICS.2005.28. ISSN: 1530-1435.

Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, September 2010. doi: 10.1109/ICSM.2010.5609732. ISSN: 1063-6773.

D. Cubranic and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 408–418, May 2003. doi: 10.1109/ICSE.2003.1201219. ISSN: 0270-5257.

D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005. ISSN 1939-3520. doi: 10.1109/TSE.2005.71.

Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE Transactions on Software Engineering*, 43(4):359–371, April 2017. ISSN 1939-3520. doi: 10.1109/TSE.2016.2592905.

Kostadin Damevski, Hui Chen, David C. Shepherd, Nicholas A. Kraft, and Lori Pollock. Predicting Future Developer Behavior in the IDE Using Topic Models. *IEEE Transactions on Software Engineering*, 44(11): 1100–1111, November 2018. ISSN 1939-3520. doi: 10.1109/TSE.2017.2748134.

R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In

2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pages 241–248, September 2005a. doi: 10.1109/VLHCC.2005.32. ISSN: 1943-6106.

Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 2, page 207, Cape Town, South Africa, 2010. ACM Press. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810331.

Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 183–192, New York, NY, USA, May 2005b. Association for Computing Machinery. ISBN 978-1-59593-073-6. doi: 10.1145/1056018.1056044.

Francoise Detienne. *Software Design – Cognitive Aspect*. Springer Science & Business Media, October 2001. ISBN 978-1-85233-253-2.

H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 190–198, November 1998. doi: 10.1109/ICSM.1998.738508. ISSN: 1063-6773.

H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23, September 2003. doi: 10.1109/IWPSE.2003.1231205.

Marko Gašparič and Andrea Janes. What Recommendation Systems for Software Engineering Recommend: A Systematic Literature Review. *Journal of Systems and Software*, 113, November 2015. doi: 10.1016/j.jss.2015.11.036.

Zhongxian Gu, Drew Schleck, Earl T. Barr, and Zhendong Su. Capturing and Exploiting IDE Interactions. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 83–94, Portland Oregon USA, October

2014. ACM. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661144.

A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293, September 2004. doi: 10.1109/ICSM.2004.1357812. ISSN: 1063-6773.

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, pages 2511–2520, New York, NY, USA, April 2014. Association for Computing Machinery. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557073.

Austin Z. Henley, Alka Singh, Scott D. Fleming, and Maria V. Luong. Helping programmers navigate code faster with Patchworks: A simulation study. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 77–80, July 2014. doi: 10.1109/VLHCC.2014.6883026. ISSN: 1943-6106.

Austin Z. Henley, Scott D. Fleming, and Maria V. Luong. Toward Principles for the Design of Navigation Affordances in Code Editors: An Empirical Investigation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 5690–5702, Denver Colorado USA, May 2017. ACM. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025645.

Riitta Jääskeläinen. Think-aloud protocol. *Handbook of translation studies*, 1:371–374, 2010. Publisher: John Benjamins publishing company Amsterdam.

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, pages 217–224, New York, NY, USA, October 2011. Association for Computing Machinery. ISBN 978-1-4503-0716-1. doi: 10.1145/2047196.2047225.

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, pages 159–168, New York, NY, USA, March 2005. Association for Computing Machinery. ISBN 978-1-59593-042-2. doi: 10.1145/1052898.1052912.

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, Portland, Oregon, USA, 2006. ACM Press. ISBN 978-1-59593-468-0. doi: 10.1145/1181775.1181777.

Amy J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, April 2004. Association for Computing Machinery. ISBN 978-1-58113-702-6. doi: 10.1145/985692.985712.

Amy J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, May 2008. Association for Computing Machinery. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368130.

Amy J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, April 2009. Association for Computing Machinery. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518942.

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, December 2006. ISSN 1939-3520. doi: 10.1109/TSE.2006.116.

Takashi Kobayashi, Nozomu Kato, and Kiyoshi Agusa. Interaction histories mining for software change guide. In

2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE), pages 73–77, June 2012. doi: 10.1109/RSSE.2012.6233415. ISSN: 2327-0942.

Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. Blaze: supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2195–2200, New York, NY, USA, May 2012. Association for Computing Machinery. ISBN 978-1-4503-1016-1. doi: 10.1145/2212776.2223775.

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, pages 3073–3082, New York, NY, USA, April 2013. Association for Computing Machinery. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466419.

Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, page 185, Cape Town, South Africa, 2010. ACM Press. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806829.

Thomas D. LaToza and Brad A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124, September 2011. doi: 10.1109/VLHCC.2011.6070388. ISSN: 1943-6106.

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*, pages 492–501, Shanghai China, May 2006. ACM. ISBN 978-1-59593-375-1. doi: 10.1145/1134285.1134355.

Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. Scents in Programs:Does Information Foraging Theory Apply to Program Maintenance? In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 15–22, September 2007. doi: 10.1109/VLHCC.2007.25. ISSN: 1943-6106.

Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1323–1332, New York, NY, USA, April 2008. Association for Computing Machinery. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357261.

Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, February 2013. ISSN 1939-3520. doi: 10.1109/TSE.2010.111.

Seonah Lee and Sungwon Kang. Clustering navigation sequences to create contexts for guiding code navigation. *Journal of Systems and Software*, 86(8):2154–2165, August 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2013.03.103.

Seonah Lee and Sungwon Kang. What situational information would help developers when using a graphical code recommender? *Journal of Systems and Software*, 117:199–217, July 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.02.050.

Seonah Lee, Sungwon Kang, and Matt Staats. NavClus: A graphical recommender for assisting code exploration. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1315–1318, May 2013. doi: 10.1109/ICSE.2013.6606706. ISSN: 1558-1225.

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, March 2015. ISSN 1939-3520. doi: 10.1109/TSE.2014.2362138.

Seonah Lee, Jaejun Lee, Sungwon Kang, Jongsun Ahn, and Heetae Cho. Code Edit Recommendation Using a Recurrent Neural Network. *Applied Sciences*, 11(19):9286, January 2021. ISSN 2076-3417. doi: 10.3390/app11199286.

Walid Maalej, Thomas Fritz, and Romain Robbes. Collecting and Processing Interaction Data for Recommendation Systems. In Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 173–197. Springer, Berlin, Heidelberg, 2014. ISBN 978-3-642-45135-5. doi: 10.1007/978-3-642-45135-5_7.

Sana Maki, Sègla Kpodjedo, and Ghizlane El Boussaidi. Context extraction in recommendation systems in software engineering: a preliminary survey. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 151–160, USA, November 2015. IBM Corp.

Yuri Malheiros, Alan Moraes, Cleyton Trindade, and Silvio Meira. A Source Code Recommender System to Support Newcomers. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 19–24, July 2012. doi: 10.1109/COMPSAC.2012.11. ISSN: 0730-3157.

Roberto Minelli, Andrea Mocci, Michele Lanza, and Lorenzo Baracchi. Visualizing Developer Interactions. In *2014 Second IEEE Working Conference on Software Visualization*, pages 147–156, September 2014. doi: 10.1109/VISSOFT.2014.31.

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, May 2015. doi: 10.1109/ICPC.2015.12. ISSN: 1092-8138.

G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006. ISSN 1937-4194. doi: 10.1109/MS.2006.105.

C. Parnin and C. Gorg. Building Usage Contexts During Program Comprehension. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 13–22, June 2006. doi: 10.1109/ICPC.2006.14. ISSN: 1092-8138.

Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, July 1987. ISSN 0010-0285. doi: 10.1016/0010-0285(87)90007-7.

David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret Burnett, and Rachel Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 109–116, September 2011. doi: 10.1109/VLHCC.2011.6070387. ISSN: 1943-6106.

David Piorkowski, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. Foraging and navigations, fundamentally: developers' predictions of value and cost. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 97–108, Seattle WA USA, November 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950302.

David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3063–3072, New York, NY, USA, April 2013. Association for Computing Machinery. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466418.

Peter Pirolli and Stuart Card. Information Foraging. *Psychological Review*, 106(4):643–675, 1999. doi: 10.1037/0033-295x.106.4.643.

Roger S. Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

Sebastian Proksch, Veronika Bauer, and Gail C. Murphy. How to Build a Recommendation System for Software Engineering. In Bertrand Meyer and Martin Nordio, editors, *Software Engineering: International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 1–42. Springer International Publishing, Cham, 2015. ISBN 978-3-319-28406-4. doi: 10.1007/978-3-319-28406-4_1.

Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, July 2010. ISSN 1937-4194. doi: 10.1109/MS.2009.161.

Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 11–20, New York, NY, USA, September 2005. Association for Computing Machinery. ISBN 978-1-59593-014-9. doi: 10.1145/1081706.1081711.

David Roethlisberger, Oscar Nierstrasz, and Stephane Ducasse. Autumn Leaves: Curing the Window Plague in IDEs. In *2009 16th Working Conference on Reverse Engineering*, pages 237–246, October 2009. doi: 10.1109/WCRE.2009.18. ISSN: 2375-5369.

Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave W. Binkley. Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212, March 2016. doi: 10.1109/SANER.2016.101.

Thomas Rolfsnes, Leon Moonen, Stefano Di Alesio, Razieh Behjati, and Dave Binkley. Aggregating Association Rules to Improve Change Recommendation. *Empirical Software Engineering*, 23(2):987–1035, April 2018. ISSN 1573-7616. doi: 10.1007/s10664-017-9560-y.

David Rothlisberger, Oscar Nierstrasz, Stephane Ducasse, Damien Pollet, and Romain Robbes. Supporting task-oriented navigation in IDEs with configurable HeatMaps. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 253–257, May 2009. doi: 10.1109/ICPC.2009.5090052. ISSN: 1092-8138.

Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer. Eyes on Code: A Study on Developers' Code Navigation Strategies. *IEEE Transactions on Software Engineering*, 48(5):1692–1704, May 2022. ISSN 1939-3520. doi: 10.1109/TSE.2020.3032064.

Alberto Sillitti, Giancarlo Succi, and Jelena Vlasenko. Understanding the impact of Pair Programming on developers attention: A case study on a large industrial experimentation. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1094–1101, June 2012. doi: 10.1109/ICSE.2012.6227110. ISSN: 1558-1225.

Janice Singer, Robert Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 325–334. IEEE, 2005. doi: 10.1109/ICSM.2005.66.

Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers on - CASCON '10*, pages 174–188, Toronto, Ontario, Canada, 2010. ACM Press. doi: 10.1145/1925805.1925815.

Justin Smith, Chris Brown, and Emerson Murphy-Hill. Flower: Navigating program flow in the IDE. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 19–23. IEEE, 2017. doi: 10.1109/VLHCC.2017.8103445.

Zephyrin Soh, Thomas Drioul, Pierre-Antoine Rappe, Foutse Khomh, Yann-Gael Gueheneuc, and Naji Habra. Noises in Interaction Traces Data and Their Impact on Previous Research Studies. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, October 2015. doi: 10.1109/ESEM.2015.7321209. ISSN: 1949-3789.

Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Noise in Mylyn interaction traces and its impact on developers and recommendation systems. *Empirical Software Engineering*, 23(2):645–692, April 2018. ISSN 1573-7616. doi: 10.1007/s10664-017-9529-x.

M.-A. Storey, L.-T. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall. How Programmers can Turn Comments into Waypoints for Code Navigation. In *2007 IEEE International Conference on Software Maintenance*, pages 265–274, October 2007. doi: 10.1109/ICSM.2007.4362639. ISSN: 1063-6773.

Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Waypointing and social tagging to support program navigation. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pages 1367–1372, New York, NY, USA, April 2006. Association for Computing Machinery. ISBN 978-1-59593-298-3. doi: 10.1145/1125451.1125704.

Akihiro Yamamori, Anders Mikael Hagward, and Takashi Kobayashi. Can Developers' Interaction Data Improve Change Recommendation? In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 128–137, July 2017. doi: 10.1109/COMPSAC.2017.79. ISSN: 0730-3157.

A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004. ISSN 1939-3520. doi: 10.1109/TSE.2004.52.

Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, USA, May 2004. IEEE Computer Society. ISBN 978-0-7695-2163-3. doi: 10.1109/ICSE.2004.1317478. ISSN: 0270-5257.

Index

Typeset May 23, 2023