Bachelor's Thesis

Understanding How Different Artifacts Contribute to Developer Mental Models in CML-assisted Software Maintenance

Lily Mae Busche

September 11, 2026

Advisors: Alisa Welter
Anna-Maria Maurer

Reviewers: Prof. Dr. Sven Apel
Prof. Dr. Anna Maria Feit





Chair of Software Engineering ◇ Saarland Informatics Campus ◇ Saarland University

Lily Mae Busche: *Understanding How Different Artifacts Contribute to Developer Mental Models in CML-assisted Software Maintenance,* © September 2026

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und zur Code-Vervollständigung und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision and code completion, and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional) Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Abstract

Software maintenance, the adaptation of an existing software system to new requirements, represents a majority of the everyday work of programmers. Much of this work is focused on maintaining multiple artifacts, like code and class diagrams, at the same time. Within this context, the programmer's mental model takes center stage for informing their understanding and workflow decisions. Many previous studies have thoroughly investigated how program comprehension functions for single specific artifacts, and there is substantial research on the mental models held by programmers. Yet, little to no prior work has focused on understanding how these factors interact and function together in context. Moreover, the recent rapid rise in adoption of Large Language Model-based Coding Assistance Tools has without a doubt fundamentally changed these workflows: Programmers no longer write most of the code, don't create implementation plans on their own and use these tools for guidance when comprehending new codebases. All of this is also very likely to influence the programmers' mental models.

Consequently, this thesis performs an exploratory study in multi-artifact software maintenance. We conduct an experiment in which participants must work on a software maintenance task involving multiple artifacts, some with GitHub Copilot and some without. We analyze which artifacts they prioritize during the initial stages of implementation and how they distribute their time across the artifacts overall. Using this data, we propose explanations for how this relates to their mental model and then further analyze how using GitHub Copilot affects the accuracy of their mental model after completing the task.

With this, we lay the foundations for future research in the area of multi-artifact software maintenance. We propose a number of possible alterations to and continuations of our study, which should help future work broaden this foundation and begin developing proper theoretical frameworks.

Acknowledgments

My most sincere appreciation extends to all participants of the study performed for this thesis, who so graciously donated a significant amount of their time and without whom this work would not have been possible.

My most personal gratitude extends to my friend Hana, who endured so many of my grievances and endless ramblings during the many months that this work took to create. Her constant emotional support, especially in the final weeks of writing, has been absolutely invaluable.

I also want to thank my friends in Wisp Forest and The Library, with whom I shared many interesting and often entertaining conversations about my research, for helping foster and nurture my interest in programming and especially its requisite human element.

Beyond this, I am grateful to my remaining friends and my family for their love, support and constant encouragement.

Finally, I am indebted to my advisors, Alisa Welter and Anna-Maria Maurer, who always provided helpful feedback and helped introduce me to the world of scientific research and especially research involving human participants.

Contents

1 Introduction 1
2 Background 3
2.1 Artifacts . . . 3
2.2 Software Models . . . 3
2.3 Program Comprehension . . . 4
2.4 Mental Models . . . 4
2.4.1 Understanding Mental Models: An example . . . 5
2.5 Coding Assistance Tools . . . 5
3 Related Work 7
3.1 Software Models in Practice . . . 7
3.2 Program Comprehension and Software Models . . . 7
3.3 Mental Models and Software Models . . . 8
3.4 Mental Models of Code . . . 8
3.5 Mental Models of Technical Systems . . . 9
3.6 Influence of Coding Assistance Tools on Mental Models . . . 10
4 Methodology 11
4.1 Research Questions . . . 11
4.2 Operationalization . . . 12
4.2.1 Participants . . . 12
4.2.2 Pre-Questionnaire . . . 12
4.2.3 Participant Workflow . . . 13
4.2.4 Template System . . . 13
4.2.5 Participant Task . . . 14
4.2.6 Technologies . . . 15
4.2.7 LLM Selection . . . 15
4.2.8 Post-Questionnaire . . . 16
4.2.9 Data Collection . . . 16
4.2.10 Processing the Screen Recordings . . . 17
5 Evaluation 19
5.1 Results . . . 19
5.1.1 Demographic . . . 19
5.1.2 Research Question 1 . . . 20
5.1.3 Research Question 2 . . . 20
5.1.4 Research Question 3 . . . 22
5.2 Discussion . . . 23
5.2.1 Research Question 1 . . . 23
5.2.2 Research Question 2 . . . 25

5.2.3 Research Question 3 . . . 27
5.2.4 Summary . . . 28
5.3 Threats to Validity . . . 29
5.3.1 Internal Validity . . . 29
5.3.2 External Validity . . . 29
5.3.3 Construct Validity . . . 30
6 Concluding Remarks 31
6.1 Conclusion . . . 31
6.2 Future Work . . . 32
A Appendix 33
A.1 Pre-Questionnaire . . . 33
A.2 Post-Questionnaire . . . 38
A.3 Task Specification . . . 45
A.4 Consent Form . . . 48

Statement on the Usage of Generative Digital Assistants 51

Bibliography 53

List of Figures

Figure 5.1 Box Plot describing the time (in minutes) spent on each artifact, split by experimental group 21
Figure 5.2 Box Plot describing the post-questionnaire performance in each question category, split by experimental group 23
Figure A.1 Pre-Questionnaire for the Non-Copilot group (Page 1) 33
Figure A.2 Pre-Questionnaire for the Non-Copilot group (Page 2) 34
Figure A.3 Pre-Questionnaire for the Copilot group (Page 1) 35
Figure A.4 Pre-Questionnaire for the Copilot group (Page 2) 36
Figure A.5 Pre-Questionnaire for the Copilot group (Page 3) 37
Figure A.6 Post-Questionnaire for the Non-Copilot group (Page 1) 38
Figure A.7 Post-Questionnaire for the Non-Copilot group (Page 2) 39
Figure A.8 Post-Questionnaire for the Non-Copilot group (Page 3) 40
Figure A.9 Post-Questionnaire for the Copilot group (Page 1) 41
Figure A.10 Post-Questionnaire for the Copilot group (Page 2) 42
Figure A.11 Post-Questionnaire for the Copilot group (Page 3) 43
Figure A.12 Post-Questionnaire for the Copilot group (Page 4) 44
Figure A.13 Task specification for participants in the Non-Copilot Group (Page 1) 45
Figure A.14 Task specification for participants in the Copilot Group (Page 1) . . . 46
Figure A.15 Task specification for participants (Page 2, both groups) 47
Figure A.16 Consent form for participating in the experiment (Page 1) 48
Figure A.17 Consent form for participating in the experiment (Page 2) 49

List of Tables

Table 5.1 Descriptive Statistics for Participant Experience ($N = 11$) 19
Table 5.2 Multitasking metrics by experimental group, mean + SD 21
Table 5.3 Post-Questionnaire performance by experimental group, mean + SD 22

1

Introduction

In the practice of software engineering, it is common for existing functionality of a software system to change or for new functionality to be added in response to changing requirements. To do this, programmers implement the necessary changes and additions to multiple artifacts of a system like code and code comments, textual documentation, software models, unit tests and others.

This process, referred to as *software maintenance*, and many of its individual facets are well-studied in a large body of literature [3, 19, 27]. As one of the facets, it is understood that software models following the Unified Modeling Language (UML) help reduce the mental load experienced by developers writing and maintaining code—consequently, they are frequently used in industry[2, 24]. It is also clear that significant knowledge pertaining to a software system's design and implementation is never properly written down and is instead held in the mental models of programmers familiar with the system [18]. It is thus evident that both UML diagrams and the mental models which programmers hold about the software system are highly relevant aspects of the everyday software maintenance workflow. Together with the code itself, they form the backbone of knowledge by which a software system is maintained.

In recent years, coding assistance tools (CATs) such as *GitHub Copilot*, *Claude Code* and *OpenAI Codex* based on Large Language Models (LLMs) have significantly changed how many programmers write code—helping with architectural decisions, code comprehension and generating entire programs. Without a doubt, this alters how programmers comprehend a given subject software system and how their mental model of it forms and develops.

Some important aspects of the software maintenance workflow are, currently, not well understood: How developers form their mental model when multiple artifacts are involved, how they update it to reflect required changes and how they implement those changes on multiple artifacts. Moreover, there is emerging but limited knowledge of how all this proceeds differently when a coding assistance tool is involved.

Thus, in this thesis we perform an initial exploratory study with the goal of understanding how the participants proceed as they modify a subset of an existing software system. The system is expressed as a UML class diagram, code and system (end-to-end) tests—all of which need to be modified given a change to the system's specification. We analyze how the participants comprehend the given artifacts, how they use them to inform their mental model, and how they proceed when implementing the necessary changes. We split the participants into two groups: one is allowed to use a coding assistance tool during the study, the other must solve the task without.

2

Background

In this chapter, we establish the necessary concepts required to understand the context and approach of our study.

2.1 Artifacts

During the development and maintenance of a software system, many different artifacts (i.e., specific tangible results) are produced. In our study, we concern ourselves with three of these: Code, Class Diagrams, and System Tests.

Class diagrams, part of the family of Software Models, are more closely discussed in the next section. Systems tests, also known as end-to-end tests, are used to verify the entire software system as one unit. Consequently, we employ them in our study since they cannot easily be changed without an adequately formed mental model of the entire system—a concept which is more closely discussed further below.

2.2 Software Models

Software models are abstracted representations of programs which in some way describe their structure, functionality, or both. In practice, software models are generally expressed using the Unified Modeling Language (UML), a standard adopted in 1997 [28] which defines a wide range of diagrams. Of these, class diagrams, use case diagrams, sequence diagrams and activity diagrams in particular are most commonly used in industry and research [9, 17]. While other modeling standards exist, UML dominates the vast majority of uses. Still, it is often not followed precisely, with practitioners instead relying on informal sketches and only adhering to some constructs of the UML standard to aid comprehension [2].

The UML diagram type relevant for our research, the class diagram, describes the structure of an object-oriented software system. It shows individual classes of the system, their attributes and how they connect to other classes. The level of detail at which the system class structure is described is of course variable by the designer. A high level of detail improves diagram comprehensibility [21] but also seemingly risks worsening programmer performance on maintenance tasks [11], as found by Fernández-Sáez et al in 2016. Accordingly, in our study we prepare a diagram with a high level of detail but omit high-density minutiae which are irrelevant to the participants' task.

2.3 Program Comprehension

Program comprehension is the first necessary step of software maintenance, and the process which programmers use to form their understanding of a software system's structure and functionality. Although we do not aim to directly measure comprehension in our study, it is plainly relevant for understanding how participants behave and how they use the artifacts. While numerous comprehension strategies can be classified, recent research generally accepts that the two main modes are *top-down* and *bottom-up* program comprehension [22].

In bottom-up comprehension, the programmer starts by examining source code and other artifacts at the lowest level (i.e., reading line by line), slowly building connections between units of logic and repeating this process to build a general understanding of the entire program.

On the flip side, an experienced programmer begins top-down comprehension by first identifying special places of interest in the program and building hypotheses about related functionality. Given sufficient domain knowledge, the programmer will then confirm these hypotheses by using further mental heuristics to find the implementations they expect. If the actual implementation deviates from this expectation, the programmer falls back to bottom-up comprehension to refine their hypotheses. Since significantly less code must be read this way, top-down comprehension proceeds significantly faster process than bottom-up comprehension [22].

The primary effect of the program comprehension process is that the programmer forms and develops their mental model of the software system, something which is further discussed in the next section.

2.4 Mental Models

Mental models are the cognitive representations or frameworks which individuals form about particular objects, systems, activities, or similar. They are the tool used to predict the outcome of an activity or the behavior of a system and are thus vital to the work of a programmer. In fact, mental models play such a central role that the majority of knowledge about a given software system is held in the mental models of its maintainers—without ever being formally documented [18].

According to an overview and analysis by Westbrook [30], the mental models which users develop of systems or processes consist of three major elements: key components, relationships between these components and techniques for interacting with these components. The third element is the key factor influencing how the mental model is operationalized when interacting with the system or process.

Westbrook also notes that due to the inherent complexity and drastic variability of mental models different users form of the same subject system or process, it is not practical to tailor systems and processes to individuals. Patterns which emerge upon analyzing different users' models however, can and should be used by designers to better align the actual experience with that which the most common mental models describe.

2.4.1 Understanding Mental Models: An example

Also included in the work by Westbrook is an exploratory study on the mental models held by students about *information-seeking*, i.e., the process of using information resources like the internet to research a specific topic. We will describe the process and findings of this study here, since understanding how a mental model can be elicited from its holder, and which patterns may emerge between those of multiple individuals, helps in picturing how a mental model might be understood.

Since documenting the mental model of a complex process such as *information-seeking* is likely to be an unfamiliar task to most people, participants were first tasked with documenting their process for grocery shopping. Specifically, they were asked to describe all steps and decision frameworks they use in such detail that a hypothetical immigrant unfamiliar with American grocery stores would not be lost at any point in the process.

Equipped with this requisite experience in expressing the mental model held of a reasonably complex process, participants then continued to apply the same technique to documenting their mental model of information-seeking. In analyzing the resulting narratives and diagrams, Westbrook focuses on identifying the major components of an information system described therein alongside the strategies described for exploring these components.

Within the components described, a common pattern emerges: the internet, the library and other people formed the major components of most mental models. For interacting with these information resources, Westbrook identifies three major strategy patterns: The *decision tree*, where participants model information seeking as a linear process with beginning, middle and end and straight-forward choices at decision points based on criteria like resource relevance. The *network*, which describes a much more haphazard layout of the information system and sees the participant in a less active role, being bounced around between network components since information queries are seen as turning up results with high variance making it difficult to plan ahead. Lastly, the *storm* model sees participants envisioning information resources in a similarly uncontrollable manner as the network model, but with a more positive outlook and active role in the system. Participants take in the stream of highly variant resources and pluck out those which are relevant to them, using the randomness to their advantage and to inform further research focus.

2.5 Coding Assistance Tools

In our study, some participants are allowed to use **GitHub Copilot**[1]—a Coding Assistance Tool (CAT) based on a a Large Language Model (LLM). These integrate directly with the programmer's development environment and provide a chat interface for generating code, explaining existing code and performing a wide range of refactoring tasks. In recent years, their use (through products such as **GitHub Copilot**[1] and **Claude Code**[2]) has become widespread in industry [16] and, to a lesser extent, in open-source communities which are frequently driven by hobbyists [6].

1 https://github.com/copilot
2 https://code.claude.com/docs/en/overview

In their most advanced, so-called *agentic*, mode, these tools are capable of autonomously performing all operations the programmer would normally do: write and refactor code, create and restructure files and run tests and shell commands. They often support further extension of their capabilities by way of so-called *tools*—external programs which interface with the CAT to perform a task it would not be capable of by default.

When an CAT is operating in this autonomous way, the human programmer retains control to interrupt and re-instruct the tool when they see fit. There is evidence that knowing how and when to change instructions and prompt the CAT to rework some generated code becomes a new vital skill which experts rely on to retain control over their codebases [15].

3

Related Work

With the necessary background established, we now move on to presenting related work in an effort to contextualize our research and contributions.

3.1 Software Models in Practice

The value of software models in the software engineering workflow is undisputed, by both researchers and practitioners: In an industry study covering employees from 3 companies and 394 participants from an online survey, Baltes and Diehl [2] find that the majority (77%) of their participants used or created a software model sketch in the last week before questioning. They further find that most of these sketches were created in less than an hour of time (68%), were informal (i.e. did not adhere to a modeling standard like the Unified Modeling Language or UML) and about half of them (58%) were archived.

75% of the sketches were created for purposes related to designing, 60% were related to explaining code or architecture to somebody else and 56% were made in an effort to understand something about the problem space or implementation. With this in mind, we present the results from Fernández-Sáez et al. [10], who conducted a study on 169 master's students in an effort to understand whether there is a difference between forward-designed (FD) and reverse-engineered (RE) diagrams with respect to code maintenance tasks. They find that participants which were given FD diagrams showed statistically significant improvements in test scores compared to those using RE diagrams. Further, members of the FD cohort reported a significantly better subjective experience using the diagram.

3.2 Program Comprehension and Software Models

As we will be presenting participants with both code and a UML class diagram, we are not merely interested in program comprehension based solely on source code. Instead, we present some findings in the area of multi-artifact program comprehension.

A study by Hendrix et al. [14] finds that programmers develop a more correct understanding of the control flow in a program and do so more quickly when the code is presented inside a Control Structure Diagram (CSD). A CSD clearly visualizes branching or looping points in the code and delineates scope and thus control flow boundaries. To find this result, the study tasked university students from two courses which required extensive programming in Java to read and comprehend snippets of source code—annotated with a CSD in the experimental group and without for the control group. Afterwards, participants had to answer a fixed set of 12 questions about the code, where the study finds that the

CSD group consistently responds more quickly and is on average significantly more likely to answer correctly.

Similarly, Yang et al. [31] experimented with displaying UML object, class and sequence diagrams next to source code in such a fashion that the parts of each diagram which are relevant to the source code fragment under inspection are highlighted. This system was experimentally tested in a classroom study, with a subsequent questionnaire asking the students questions which required program understanding and about their perception of the visualization tool. They find that the system is helpful for beginners in understanding the influence of Object-Oriented (OO) concepts like inheritance and polymorphism on the execution of a program and thus aided them in comprehending it.

Without synchronization however, Gravino et al. [13] find that only experienced practitioners benefit from using forward-designed UML models alongside OO code during program comprehension. With beginners, they instead see a significant increase in comprehension times and no improvement in the level of comprehension achieved. They suspect that this is likely due the extra effort UML novices need to invest into finding and understanding the relevant parts of each diagram for whatever they are currently in the process of comprehending.

This is a potential confounding factor for our study, since our participants will not be experienced professional programmers and thus are likely to spend extra effort comprehending both the class diagram and source code. However, since comprehension does not in itself suffer and we are not interested in comprehension speed, we do not anticipate this to be a major issue.

3.3 Mental Models and Software Models

When a programmer works to evolve or maintain an existing software system, they will frequently have two types of system models at their disposal: their own mental model and external (likely UML-formatted) software models which act as a type of documentation. The interplay between these resources, both of which will invariably be accessed by the programmer during the job, is not a studied phenomenon. One would expect that the existing mental model influences the programmer's understanding and usage of the external software models and that, vice-versa, the software model works to inform the mental model. Yet, while both effects are studied in isolation [13, 20, 31] and it has been shown that properly designed UML models benefit developer performance [7, 10], the mechanisms and manner by which one influences usage and comprehension of the other has not been investigated.

3.4 Mental Models of Code

One of the earliest studies of the mental models programmers use when making changes to a software system was conducted in 1987 by Littman et al. [20]. In their research, they tasked 10 participants (experienced developers from the Jet Propulsion Laboratory) with adding "restore" functionality to an existing database management program which only supported deleting entries prior to modification.

Littman et al. identified two primary categories of requisite knowledge their participants can acquire about the program. The first kind, *static knowledge*, represents the elements of the program like functions, objects and the actions (conceptually) taken by those objects. The second kind, *causal knowledge*, encompasses the specific interactions between objects and functions and the causal consequences for program execution thereof. Both categories of knowledge, they argue, must be acquired by a participant in order to successfully carry out the modification they were tasked with.

To gather data on how participants approached the task, Littman et al. conducted one-on-one interviews with each participant as they studied the program and implemented their changes (in effect, they motivated the participants to think aloud). Each interview was recorded on video. It is not clear whether a specific strategy or set of questions was used during the interviews, only a general goal of receiving a "steady report of his or her thoughts" is mentioned. The mental model each participant formed of the program is seemingly assessed with specific questions about how different parts of the program interact and how data flows. For this too, no specific method is described.

In evaluating the approaches taken by their participants, Littman et al. found that there were two distinct strategies in how programmers approach the task: The *systematic strategy* sees the programmer first explore large parts of the program in an effort to understand the big picture of data and causality flow throughout it before starting any work on implementing the modification. In contrast, the *as-needed strategy* sees the programmer minimizing how much they study the program before getting started on the implementation. Instead, they immediately seek to identify the most appropriate parts of the program to modify and begin implementing changes, studying related parts of the program only as needed to guide their efforts.

Comparing these strategies with how successfully each participant implemented the requested changes, Littman et al. find that every participant using the *systematic strategy* was successful whereas every participant using the *as-needed strategy* failed to correctly implement all of the changes (i.e., make all necessary modifications to the program). They suggest that this is because using the *as-needed strategy*, the programmer never fully understands critical sections of the program's data flow (and hence misses crucial modifications) since which parts of the program they learn about depends heavily on how they choose to implement the changes.

3.5 Mental Models of Technical Systems

In 2019, Vogel-Heuser et al. [29] studied the mental models students build of technical systems (like a bicycle's gear shift mechanism). Similarly to Westbrook and Littman et al., they too found patterns in the models which students built—further reinforcing the observation that while mental models are highly variable in the details, they tend to fall into a limited number of big-picture categories.

In their study, data was taken from 48 exam solutions in which students were tasked with modeling a manufacturing process involving multiple substations and other machinery. During evaluation, they find that the models produced by their students fall into three groups: The first group chose to abstract individual parts of the manufacturing process by

their physical representation (e.g. substation, conveyor) whereas the second group instead chose to abstract parts of the process by responsibility (e.g. "Recognize Workpieces"). The third group combined both approaches.

3.6 Influence of Coding Assistance Tools on Mental Models

With the proliferation of coding assistance tools, the mental model which a programmer forms of the system under development takes center stage—after all, communicating and synchronizing this mental model with the assistance tool is the core task in prompting. How this change to the programming workflow affects such mental models is not currently a studied phenomenon. Much research, like that of Desolda et al. [5] and Brachman et al. [4], has gone and continues to go into understanding the mental model the user forms of the *coding assistance tool*. This, however, is a largely unrelated kind of mental model—adapting coding assistance tools such that they more closely align with user expectations (i.e., the mental model the user has about the coding assistance tool) is helpful for making better tools but it does not at all correlate with the mental model the user forms about the subject software system. Starting to fill this gap, that is, examining how users form their mental model of a software system differently when using a coding assistance tool, is a core goal of this thesis.

4

Methodology

In this chapter, we first introduce the research questions we posed to begin understanding how programmers approach multi-artifact software maintenance workflows. We then move on to detailing the design of the user study which we conducted to answer them: how we selected participants, the exact experimental task that was posed, which data we collected and how we analyzed it.

4.1 Research Questions

Since we are performing an exploratory study and there is little to no existing theory of how programmers approach maintenance (like fixing bugs, implementing new features or extending existing features) on multiple artifacts, we must start by establishing basics.

When tasked with implementing changes to an existing software system consisting of multiple artifacts (which is certainly the generally applicable case in practice), one of the first decisions the programmer must make is deciding in which artifact to implement changes first. For instance, in a model-driven development workflow one might begin by updating a class diagram and then synchronizing those changes to the actual codebase. Similarly, someone using a test-driven workflow might first write a set of new unit tests and then proceed to write code until the implementation passes the tests.

Since we do not prescribe a workflow in our study and this decision is certainly of importance for understanding how the programmer builds their mental model, we pose the first research question:

> **RQ1:** When implementing changes, which artifacts do participants begin with?

Both during implementation and the preceding comprehension process, the programmer must interact with one or multiple of the artifacts they're working with. It is to be expected that differing amounts of time are spent on each artifact. Furthermore, the usage of a Coding Assistance Tool (CAT) might influence this distribution: one could imagine that it biases towards testing and the diagram rather than code. Precluding subjective difficulties with one or more artifacts, the distribution is also likely to indicate which artifact the programmer's mental model is primarily based on. Thus, the second research question:

> **RQ2:** How much time do participants spend on each artifact?

We are generally interested in how the usage of a CAT influences the answers to both **RQ1** and **RQ2**. More than that, however, we are also looking to ascertain more broadly whether and how this affects the mental models that the participants form about the subject system. To this end, we pose the final research question:

> **RQ3:** Can we find a difference in mental model accuracy between participants who used a Coding Assistance Tool and those who did not?

4.2 Operationalization

In order to answer our research questions, we conducted an in-person user study. We provided participants with an existing software system in the form of a class diagram, the implementation and system tests. Their task was to implement a change to this system, and reflect it properly in all three artifacts. In other words, the code must support all new functionality, the class diagram must match the new code and the system tests must test the new functionality and pass on the modified implementation.

4.2.1 Participants

For our participants, we recruited students of Computer Science at the bachelor's, master's and PhD level. We purposefully did not attempt to recruit only at a specific skill level, and instead only set the minimum requirement that all participants must have taken and passed an introductory programming class at some point in their respective programs. This way, we were able to gather data from participants with a wide gamut of programming experience. No compensation was offered for participating in the study.

4.2.2 Pre-Questionnaire

Before beginning the main part of the experiment, all participants were asked to complete the pre-questionnaire. It consists primarily of questions asking the participants to estimate their own experience with programming in general and the specific technologies our task uses (i.e., the Kotlin programming language, PlantUML and GitHub Copilot where applicable). We chose these questions to collect data that would allow us to investigate if prior experience correlates to artifact use and mental model accuracy.

All questions which ask for experience estimation also ask for a relative estimation compared to peers. This strategy is taken from the work of Feigenspan et al. [8], who found that these relative estimation measures correlated more highly with students' performance in program comprehension tasks.

Importantly, we also asked whether the participant had taken the 2025 iteration of the SELab, from which our task was taken. This would otherwise be a major confounding factor, however, equipped with this knowledge we planned to additionally investigate how prior familiarity with the system affects the participant's workflow.

4.2.3 Participant Workflow

During the main part of the experiment, while working on the task, we let participants freely organize their workflow within the 50 minutes of allotted working time. This is important—in order to properly gather data for **RQ1** and **RQ2**, it must be possible for participants to work at their own pace and in their own style. Moreover, forcing someone to work according to a schema which they would not usually follow is likely to majorly impact their productivity and effectiveness.

Still, it is also important that all participants spend at least some time with each artifact. To ensure this without disrupting the data for **RQ1** and **RQ2**, we closely observed participants during the experiment and, if we found that someone was neglecting a certain artifact, gave a gentle reminder to work on it.

4.2.4 Template System

For the existing software system that our participants were be provided with, we chose the reference implementation of the task from the 2025 iteration of the Software Engineering Lab (SELab) held by our Chair. This system in particular is a good candidate for a few reasons:

- By design, the tasks provided to students in the SELab mimic the requirements and complexity of real-world software systems. Since we are interested in programmer mental models under real-world conditions, this is ideal.
- The implementation itself is readily available, known to be correct and of consistent quality. This is important since we are *not* interested in studying the effects of inconsistent or subpar code quality.
- Besides the main implementation, this system also provides a full suite of system (end-to-end) tests which verify a majority of the specification and its edge cases. This allowed us to easily verify that the participant's changes were implemented correctly and meant that we could easily provide two of these tests for the participants to work on.

This system implements a farming simulation, the initial conditions of which are given by the user in a set of configuration files for each run. Two relevant properties of this simulation are that the world it simulates is divided into tiles of different types and that the user can configure certain *incidents* to occur at specific points in time during the simulation.

Alongside the code and two system tests, which are discussed in more detail in the next section, we also provided the class diagram as a PlantUML file. The diagram covers the entire system, documenting every class along with their properties, methods and relationships at a high level of detail.

The system is implemented in approximately 3600 lines of Kotlin[1] code and comments, across 44 files. Additionally, the system ships 166 systems tests in individual files which make up about 10,000 lines of Kotlin code along with 20 system test utility files consisting

1 https://kotlinlang.org

of roughly 1300 lines of code. The class diagram denotes all 50 classes, enums and objects of the system in approximately 650 lines of PlantUML code.

4.2.5 Participant Task

For our scenario we required a task which it was not possible to implement without a basic understanding (and thus, mental model) of the existing system. Consequently, we needed to design a task which involved not only the immediate elements of the codebase that were required to implement it—and ideally also immediately affects more elements than a single registration function—but also additional elements of the system that it did not directly interact with. The ideal form of this required the participants to make changes to parts of the code which were only necessary due to conceptual changes in the system's functionality but not to make the system compile and run: someone with an insufficient mental model of cross-system interactions cannot discover that changes like these are necessary.

We thus applied the following two modifications to our system to prepare it for the experiment: We removed the *Forest* tile type and the *Animal Attack* incident. Consequently, the participant's task in the experiment was to reimplement their functionality inside the simulation, alongside parsing and validation for their configuration. To do this, participants were provided with an excerpt from the full system specification. Said excerpt partially introduced the system itself (since we still expect participants to build their understanding primarily using the artifacts) and then went on to explain functionality which is required for understanding the removed features. These removed features, the ones participants were tasked with reimplementing, were framed as new changes in the participant document.

This specific task required approximately 100 lines of code to be changed or added and fully satisfied the requirements as stated above. For instance, in order to implement the animal attack incident, other elements of the simulation (like basic plant behaviors) needed to be understood. Properly discovering necessary validation changes for the forest tile required comprehending the tile validation rules which were already implemented for other tiles. Moreover, one specific validation change also required understanding how another existing incident type behaved over the lifetime of the simulation. Making only local changes without understanding the system context thus could not be sufficient to solve the task properly.

The same goes for the system tests: modifying them to test the new features correctly required understanding how they interact with existing features. This is because system tests verify entire multi-tick runs of the simulation, meaning that participants needed to have formed a basic grasp of not only how the incident and tile they added worked, but also the general ticking behavior of the simulation, incident ordering rules and other details which were not explicitly stated in the participant document.

Since the simulation always requires a complete configuration to run, the system test framework expects each set of system test assertions to be bundled with its respective configuration files. This means that, given system tests which pass on the template system, re-adding the features we removed would not stop them from passing. To solve this, we additionally provided the participants with new configuration files for both system tests. These configurations were designed to set up a simulation only slightly altered from the one initially given to the participants to include the two new features. For the participants

then, the only remaining job was to modify the assertions in both tests to, one, pass on the modified system and, two, properly test new functionality.

4.2.6 Technologies

Participants in the experiment were provided with the existing implementation's artifacts in a Git repository. They had to use Visual Studio Code[2], set up with the official JetBrains-provided Kotlin language server plugin[3] and the PlantUML plugin[4] to work on the task. Additionally, if the participant was allowed to use GitHub Copilot, they were provided with the built-in Copilot support of Visual Studio Code and needed to use the Gemini 2.5 Pro or Gemini 3.5 Flash model.

The experimental setup provided to participants consisted of a 27″ 1440p 16:9 monitor, standard office mouse and a choice between either a standard German or standard North American keyboard. We were unable to accommodate participants which would normally work on an Apple-layout keyboard.

The GitHub Copilot panel was set up to use its standard location in the right-hand side panel of the Visual Studio Code default interface layout, covering approximately a sixth of the screen's width if not modified by the participant.

4.2.7 LLM Selection

To ensure that our findings for those participants who used Copilot are accurate to real world scenarios, we verified (to the best of our ability) that the model we allowed did not have the base SELab project in its training data. Given the large amount of students likely using Copilot while working on the project during the SELab, along with GitHub and OpenAI's data collection practices, this is a real concern.

In fact, we initially planned to use a different template project: the reference implementation of the 2024 iteration. During a pilot experiment where Copilot was prompted to implement the changes which we planned for that template project, however, it quickly became clear that it overperformed using all recent models. On one occasion, the internal thinking logs of the LLM in use at the time (GPT-5.3-Codex) revealed that it was looking for parts of the full implementation which it should have had no way of knowing about. This was a clear indicator that the 2024 project had made it into that model's training data—something which the official knowledge cutoff at 31.08.25 corroborates.

To mitigate this, we switched to the 2025 project with the task as described above and re-ran the experiment. This time, Copilot performed noticeably worse with all models—that is to say, it performed most steps of the task correctly but missed multiple important details. These were precisely of the kind which one would expect an LLM to miss: they required a non-trivial knowledge transfer since they were not explicitly specified in the task description. Still, since the previously mentioned knowledge cutoff for GPT-5.3-Codex lies right in the middle of 2025's SELab iteration, we only allowed Gemini 2.5 Pro and Gemini 2.5 Flash in our experiment (subject to availability on GitHub Copilot). Both are frontier models

2 https://code.visualstudio.com
3 https://github.com/Kotlin/kotlin-lsp
4 https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml

and—according to OpenRouter documentation[5][6]—have a knowledge cutoff in January of 2025.

4.2.8 Post-Questionnaire

After either completing the task or running out of time trying to do so, all participants were asked to complete the post-questionnaire. It was split into two sections, where one is about comprehension and the other about subjective experience.

The goal of the comprehension section was to obtain a proxy metric for how well-formed the participant's mental model was, a strategy which is supported by the work of Seupke et al. [25]. To this end, it consisted of questions which quizzed knowledge that was (or would have been) required to solve the task. Some of these questions required understanding the purpose of code in the template system (meaning they could be answered with just that system's specification) and some asked about specific implementation details that one would have come across while solving the task. Further, some questions required answers based on how the participant implemented the task. For an example, one question is as follows:

> *Where do animal attack events originate?*

This is the kind of question which can be answered easily without having studied the codebase, but simply using the participant document. It requires system-level understanding in the mental model but no specific details. Another question is as follows:

> *Which class is responsible for validating incident constraints?*

This question then instead cannot be answered using the system's specification, since it is immaterial to the big picture functionality. It requires knowing an implementation detail which someone who implemented constraint validation for an incident would have invariably come across, but which is otherwise unknowable. Overall, all questions are designed in this specific manner which makes them trivial to answer for someone who has a well-formed mental model and extremely difficult or impossible otherwise.

The subjective experience questions then asked participants to estimate their own understanding of the codebase and artifacts, as well as prompting them to report any unexpected difficulties and any steps they found particularly challenging. If applicable, they were also asked to rate how useful they found Copilot to be in general and on some particulars of the task.

4.2.9 Data Collection

In addition to the pre- and post-questionnaires, we collected data in the following ways while participants were working on the task: Primarily, we had a continuous screen recording running, along with audio such that we could encourage participants to think aloud. For participants which were allowed to use Copilot, we also recorded every Copilot interaction

5 https://openrouter.ai/google/gemini-3.5-flash
6 https://openrouter.ai/google/gemini-2.5-pro

using the Flight Recorder[7] extension for Visual Studio Code. The availability of this extension informed our unusual decision to use Visual Studio Code for working on a Kotlin project.

4.2.10 Processing the Screen Recordings

In order to obtain the necessary data for answering research questions 1 and 2 from the screen recordings, they were annotated using the open-source BORIS [12] software. Each recording was reviewed from beginning to end, with the reviewer annotating which behavior (representing artifacts in our case) the participant was performing at any given time. This was decided based upon the tabs open in the participant's editor at that time and artifacts therein. Once completed, this process resulted in a list of time windows annotated with the artifacts in use during that time window—this data was used for further analysis.

7 https://marketplace.visualstudio.com/items?itemName=SE-SIC.flight-recorder

5 Evaluation

5.1 Results

This section lays out the results of the study and the analysis performed thereupon, organized by research question.

5.1.1 Demographic

When the experimental sessions concluded, we had collected data from 11 participants in total, 6 of which were in the non-Copilot group. Participants were between 21 and 31 years old, with a median age of 24 years. Except for a single participant, nobody had previously participated in the Software Engineering Lab iteration from which the experimental task was taken. Table 5.1 presents descriptive statistics for the programming experience metrics collected in the pre-questionnaire across all participants.

Metric	Absolute	Range	Relative	Range
Programming Experience	5.73 ± 1.74	1 to 10	3.36 ± 0.81	1 to 5
Kotlin Experience	1.82 ± 0.87	1 to 5	—	—
Modeling Experience	4.27 ± 1.56	1 to 10	2.91 ± 0.94	1 to 5
Testing Experience	4.73 ± 1.62	1 to 10	2.82 ± 0.75	1 to 5

Table 5.1: Descriptive Statistics for Participant Experience ($N = 11$)

On a 10-point scale, participants reported strongly average general programming experience and slightly below average software modeling and testing experience. Experience with the Kotlin programming language in particular was reported as a very low 1.82 on a 5 point scale. The one participant who had previously taken the Software Engineering Lab also reported a lack of experience with the Kotlin language. Since participants also reported being familiar with a median 3 other languages however, we did not consider this to be an issue for the study.

As the questionnaire additionally asked participants to rate their programming experience compared to their classmates, Table 5.1 also presents descriptive statistics for those metrics. Across all metrics, participants rated themselves to be essentially among the average of their classmates, with only small deviations from the mean.

We did not observe a statistically significant correlation between the absolute and relative metrics in any category. This potentially explained by multiple of our participants not having

sufficient recent experience with peers to be able to judge what the average among their peers would be (which is the mechanism behind relative estimation proposed by Feigenspan et al. [8]). Importantly, the Mann-Whitney-U test did not discover any statistically significant difference in all four experience metrics between the Non-Copilot and Copilot groups.

5.1.2 Research Question 1

To evaluate data for **RQ1**, the five minutes of behavior data immediately following the first intentional edit were analyzed for each participant. The first intentional edit for this purpose was defined as the first edit made to any artifact which did not result from a misinput or similar user error. The five minute time window was chosen based on the observation that participants would, after the first edit, often enter a short period of establishing basic changes to the codebase before pausing to plan further. We thus wanted to capture this initial behavior.

After performing this analysis across all participants, a clear pattern emerged: participants are split into 2 distinct groups. Of these, the first group ($N = 3$) spent their five minutes on only the class diagram. The second group ($N = 8$), spent their five minutes on both the class diagram and code. Checking for the dominant artifact, we found that both groups had the class diagram on screen for at least 4 minutes and 45 seconds on average, indicating that implementation and the associated exploration begins in the class diagram.

Checking the artifact in which the first intentional edit occurs, it was of note that for 9 out of 11 participants it occurred in the class diagram. Moreover, in 8 of these 9 cases the specific edit was to add the forest tile type into the respective enum type in the class diagram. We believe this is potentially a consequence of the ordering of the task specification document and did not observe any correlation between mental model accuracy and both the first intentional edit artifact or the groups as discussed above.

5.1.3 Research Question 2

Artifact Prioritization As shown in Figure 5.1, participants in the non-Copilot group spent, on average, a roughly equal amount of time on code and the class diagram, with tests coming firmly in last place. It is notable that the amount of time spent on the class diagram varies much more strongly than the other two artifacts. Manual inspection of the screen recordings shows that this is because participants in the non-Copilot group formed two distinct modes: some would keep the class diagram open in a second tab for almost the entire duration of the experiment, others would close it at some point early on after they were done editing it and not touch it again. Which mode a given participant belonged to did not appear to correlate with their mental model accuracy.

The Copilot group instead favored code across the board, achieving a substantially higher median time spent on code than on the class diagram and consequently shows a clear prioritization of code then class diagram then tests.

Multitasking To evaluate the participants' multitasking behavior, we computed two metrics on the artifact duration data: the multitasking index and the prevalence of the primary artifact.

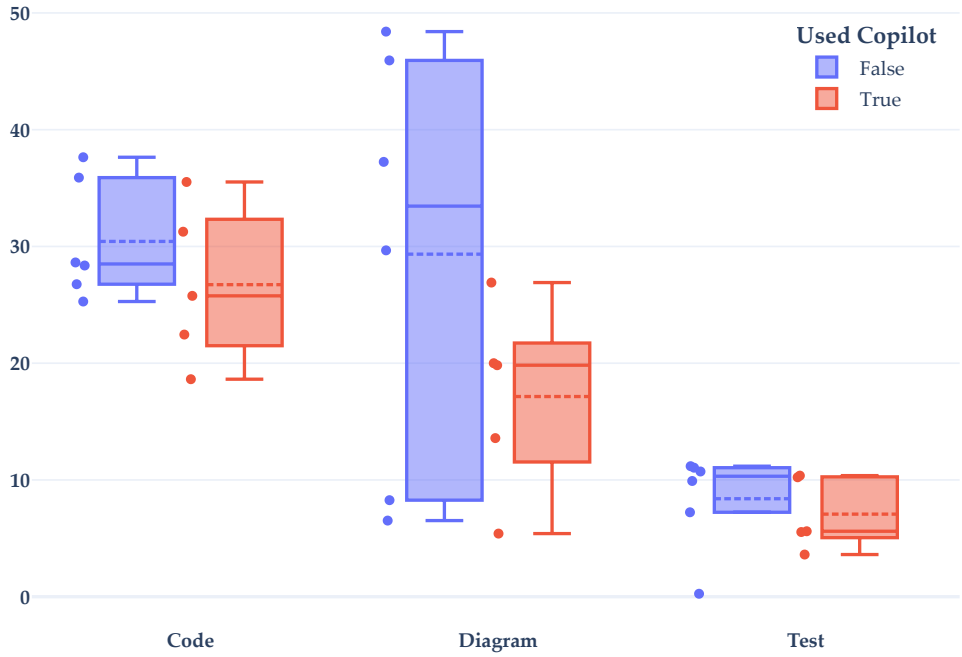


Figure 5.1: Box Plot describing the time (in minutes) spent on each artifact, split by experimental group

Metric	All Participants	Non-Copilot	Copilot
Multitasking Index	1.25 ± 0.29	1.39 ± 0.24	1.08 ± 0.27
Prevalence of Primary Activity	0.69 ± 0.17	0.8 ± 0.11	0.57 ± 0.14

Table 5.2: Multitasking metrics by experimental group, mean + SD

The multitasking index is computed from the sum of time spent on each individual artifact divided by the total session duration. Accordingly, a multitasking index close to 1 indicates that the participant only had a single artifact on screen for the majority of their session, significantly above 1 indicates that they spent significant time with at least two artifacts on screen. It is possible for the multitasking index to be below 1, which indicates that the participant spent significant time during their session reading the specification document without referencing any artifacts on screen.

Prevalence of the primary artifact (PPA) is simply the ratio between the time spent on the participant's primary artifact (i.e. the one which they spent the most time on) and the total session duration. Accordingly, a PPA close to 1 indicates that a single artifact was active during the majority of the session, a lower PPA indicates that multiple artifacts got significant attention. Combined with a lower multitasking index this means that the participant cleanly split their time between multiple artifacts.

As laid out in Table 5.2, the non-Copilot group exhibited a reasonably high PPA of 0.8, combined with a multitasking index significantly over 1 at 1.39. This indicates that there was a single artifact which received dominant focus, while other artifacts were worked on in parallel. This dominant artifact, based on the raw duration data above, is the class

diagram—this finding is consistent with the manual observation that the class diagram is often left on screen in a split tab.

Conversely however, the Copilot group exhibited both a lower PPA and lower multitasking index, indicating that they had a more isolated workflow which involved less multitasking while simultaneously distributing time more evenly across artifacts.

5.1.4 Research Question 3

The questions in the post-questionnaire's comprehension section were split into two categories: system questions and detail questions. System questions cover aspects of the simulation's general design, including ones which came from the task specification. Detail questions instead cover specific implementation details (either preexisting or created by the participant) as well as fine-grained aspects of the simulation's design and rules.

To process the comprehension section, participant answers were scored according to the questionnaire's solution key. Since some questions required multi-part answers, it was possible to obtain 0, 0.5 or 1 points for an answer. The aggregated score of a participant in a given category is then the mean of the points awarded for each question in this category.

Question Category	All Participants	Non-Copilot	Copilot
All Questions	0.46 ± 0.22	0.54 ± 0.18	0.36 ± 0.24
System Questions	0.58 ± 0.23	0.68 ± 0.19	0.45 ± 0.22
Detail Questions	0.31 ± 0.28	0.37 ± 0.29	0.24 ± 0.29

Table 5.3: Post-Questionnaire performance by experimental group, mean + SD

Table 5.3 presents the results of the questionnaire evaluation as described above. Across all participants, performance in the system questions (0.58) was nearly twice as good as in the detail questions (0.31). A similar observation applies for the results broken down by groups.

The non-Copilot group achieved an average performance of nearly 0.7 on system questions, indicating that big-picture system understanding appears to form in every participant of this group. While performance in details is significantly weaker at only 0.37 (Glass' $\Delta = -1.7$, $p = 0.0271$ as measured by a permutation test), it is clear that at least a baseline amount of detail was incorporated into the mental model of every participant in the group.

The Copilot group however only achieved an average performance across all questions of 0.36, significantly lower than the 0.54 of the non-Copilot group. While Copilot users still managed to answer about half (0.45) of system questions correctly, on average they were able to answer effectively only one detail question. The effect size and significance of this difference is much lower (Glass' $\Delta = -0.94$, $p = 0.131$) than in the non-Copilot group however, indicating that while the Copilot group performed worse overall they were more consistent between system and detail questions.

Between the two groups performance on all questions, we observed a large effect size (Glass' $\Delta = -0.91$) and reasonable significance given our small sample size ($p = 0.134$) as measured with a standard permutation test. All together this indicates both that the Copilot group formed a significantly weaker mental model overall and that it was specifically

lacking in detail. This result agrees with prior findings [1, 23], further confirming that reliance on LLM-based CATs decreases system understanding.

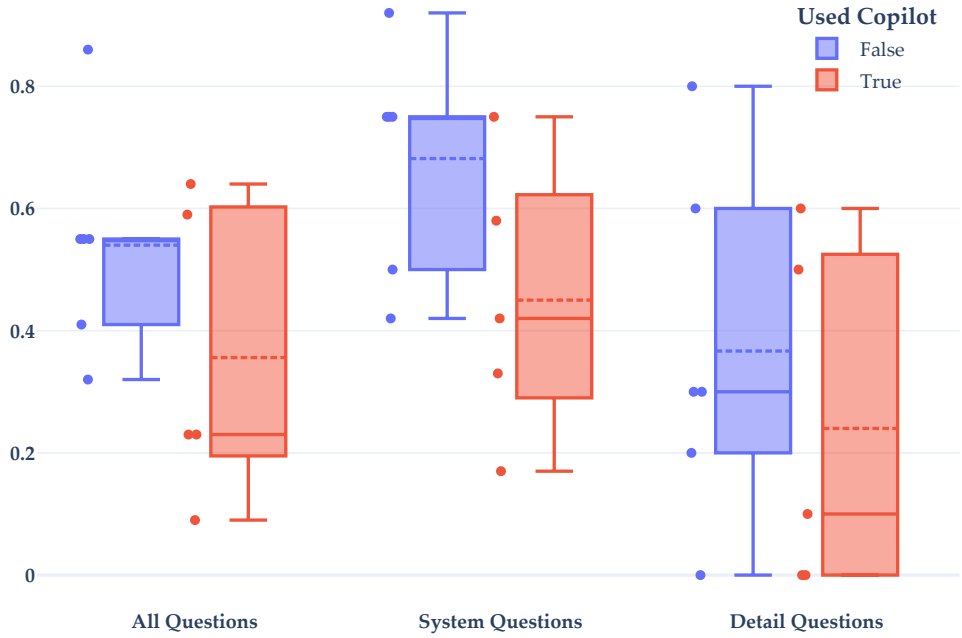


Figure 5.2: Box Plot describing the post-questionnaire performance in each question category, split by experimental group

5.2 Discussion

After presenting the results of our evaluation, we now move on to interpreting them in more detail in an effort to properly answer the three research questions we were investigating.

5.2.1 Research Question 1

RQ1 When implementing changes, which artifacts do participants begin with?

In order to find an answer to this question, we analyzed which artifacts the participants had on screen during the five minutes after their first intentional change to the codebase. This decision is based on the assumption that participants are not unlikely to start off not by making all required changes in a single artifact, but instead by making the changes required for a single functional unit across all (or at least multiple) artifacts. Consequently, capturing only the artifact where the first change in and of itself is made, does not capture this behavior appropriately.

We found that in both groups the class diagram was present for, on average, almost the entire duration of these 5 minutes and that it was always present, even in the singular trial where the first modification itself was made to the code artifact. The secondary observation,

that more than half of the participants spent time on both code and diagram, supports the above assumption upon which this analysis is based. Manual inspection of the respective screen recordings further confirms that when this was the case, making all the changes required for one feature (like for instance the Forest tile) in both code and diagram is the exact approach that was taken. This also indicates a desire to keep the artifacts in sync with each other and the participant's mental model. Further, in nearly all cases, the diagram came first.

This very strong tendency to implement changes in the class diagram first indicates that it, in some way, represents a natural starting point for developers working on an unfamiliar system. One possible explanation for this is that the class diagram represents the software system at a significantly higher level of abstraction than the code itself. This makes it easier to identify the exact changes which need to be made and where to make them, seeing as there will generally be fewer of them and they will be simpler in nature. To some extent this interpretation is supported by the observation that nearly all participants started with the (ostensibly) simpler change to make: implementation of the Forest tile. As already pointed out above however, it is very possible that this tendency is simply a result of the task document's ordering, which lists the Forest tile before the Animal Attack incident. It should also be noted that another possible explanation for this observation is simple bias introduced by the manner in which object-oriented programming is taught at most universities—class material will generally teach that modifying the class diagram first is the correct order.

Given these results, it is highly of note that not a single participant started by working on the system tests nor the few unit tests which were included with the system but not part of the task. In fact, the test artifact category was generally ignored by nearly all participants until they received a nudge from the experimenter. Together with the model-related behavior discussed above, this indicates that, on this task at least, model-driven development is in some sense a more natural approach to take than test-driven development. It must be noted that an important reason for this is likely that the system tests which participants were asked to modify can be interpreted to sit at an even lower level of abstraction than the code itself: since they test an actual run of the simulation, they require being able to simulate an execution of the system in the mind of the participant. This is a type of understanding which, given the complex nature of the system, simply cannot be formed in the short time before a participant generally makes their first change, strongly disadvantaging the test artifact in this context.

All together, these findings suggest the interpretation that the class diagram comes first specifically because its high level of abstraction means that it is the easiest starting point for forming a mental model of the software system. Not only does this make it quick to comprehend, at least at a surface level, it also makes it relatively easy to modify. Since the budding mental model a participant has at this stage does not yet contain enough detail to make specific changes to code or tests, the class diagram offers an easy way to begin applying the model under formation, which we know from prior research [30] helps with developing a more robust model more quickly.

5.2.2 Research Question 2

RQ2 How much time do participants spend on each artifact?

Analysis contributing towards an answer to this research question was based entirely on the artifact interval data extracted from the screen recordings. We split the analysis into two main sections: artifact prioritization itself (i.e., the distribution of time among the artifacts) and multitasking behavior (i.e., if and how participants spent time on multiple artifacts simultaneously). We will now discuss the findings from both analyses split by experimental group in an effort to gain better contextual insight.

Non-Copilot Group In the non-Copilot group, we found that time is, on average, distributed roughly equally between code and the class diagram, with tests firmly coming in last place. However, the class diagram artifact exhibits an extremely wide spread with clusters near the top and the bottom of the range. As mentioned above, this is the result of two distinct modes forming among the non-Copilot participants. The majority, sitting at or above the mean, referenced the class diagram for a significant duration during the study, often (as manual inspection reveals) for most of the time which they spent working on code. This agrees with prior findings [24] and indicates that programmers naturally gravitate towards using a class diagram when it is available.

The minority, sitting close to the minimum time spent on the class diagram, instead did all of their work on the class diagram early on, closed it as soon as they were done and never opened it again. Interestingly, this does not appear to correlate with either programming experience as reported in the pre-questionnaire or how useful the class diagram was reported to be in the post-questionnaire. This also does not correlate with how frequently participants reported using UML diagrams in their own development. However, there is a slight but potentially insignificant correlation ($r = 0.7$, $p = 0.123$) to reported software modeling experience: this offers the potential explanation that participants who are less experienced with software modeling overall are more likely to ignore the class diagram for most of their session.

In keeping with the above findings, participants in the non-Copilot group were found to have a highly prevalent primary artifact with a PPA of 0.8, meaning that that primary artifact was on screen for 80% of the duration of their session. This primary artifact, in general, appears to be the class diagram. Combined with their multitasking index of 1.39, meaning that 39% of their session was spent with two artifacts on screen, this perfectly captures the participants' tendency to work simultaneously on both code and the class diagram. Overall, these findings make it clear that code is the most consistently important artifact, and while the class diagram was additionally preferred by most participants, it did not have a measurable impact on mental model accuracy as measured in the post-questionnaire.

Copilot Group Conversely, the Copilot group showed a strong preference of code over the class diagram, so that across the entire group a clear code then class diagram then tests prioritization became apparent. In addition, the class diagram artifact, while still having the largest variance of all three, had much reduced variance as compared to the non-Copilot group. Manual verification of the screen recordings showed that this is because the mode where participants continuously reference the class diagram while implementing did not

exist in this group. One possible, if not likely, explanation for this is twofold: On one hand, Copilot itself represents a source of architecture information—instead of combing the class diagram for the specific information they need, participants instead asked Copilot to find it for them. On the other hand, Copilot (or rather, LLM-based CATs in general) is well-known to perform poorly with large class diagrams, which this system has one of. Combining the factors, participants were more likely to fall into the "modify class diagram once, then focus on code" mode.

A possible explanation for why code is the dominant artifact among Copilot participants can be found when examining their behavior in the screen recordings. Generally, participants would prompt Copilot to implement a specific change or guide them along doing it themselves. Since the changes to the class diagram were reasonably trivial in the experimental task, participants would usually conclude working on the class diagram quickly and then move on to code. Here, they would then spend a very significant amount of time simply reviewing individual changes which Copilot introduced. Whether this behavior is natural to participants or prompted by the phrasing in the task document, which forbids handing the task to Copilot entirely to keep data comparable between the groups, is unclear. Overall, this is in keeping with prior research [26] which found that programmers who use Copilot write less code themselves and instead perform a code review-esque activity. This then eliminates the need for a reference artifact on screen, since Copilot itself provides both the reference and implementation. As will be seen later, this is also likely a contributing factor to worse mental model retention since participants did not need to form a mental model at the level of detail which is required to write code themselves.

Further distinct from the non-Copilot group is that the Copilot group's participants were found to exhibit significantly less multitasking behavior. In fact, their mean multitasking index was close to 1 (at 1.08) and their primary artifact less prevalent (with a PPA of only 0.57). As already mentioned above, this indicates that participants in this group had a more isolated workflow—specifically, the lower multitasking index means that they rarely had more than one artifact on screen at a time. This might support the previous interpretation that Copilot itself can replace a second artifact: Since one would generally expect participants not to work on multiple artifacts simultaneously but instead use one as reference and one as the active work site, it is feasible that Copilot replaces the reference artifact in this context. The primary artifact not being as dominant as for the non-Copilot group is then also a logical consequence of this workflow, since significant time was spent on multiple artifacts still. This in turn, without multitasking, takes away time to spend on the primary one.

The *Test* Artifact Across both groups, the test artifact received very little attention during all experimental sessions and time spent on it showed an extremely tight spread in the 5 to 10 minute range. We assume that the direct cause of this is that, apart from a single participant who briefly examined a system test during their session, no participant voluntarily began working on the system tests. Instead, work on the system tests was generally prompted by a nudge from the experimenter, which was issued 15 minutes before the end of every experimental session.

An explanation for why participants did not themselves choose to work on the system tests is likely twofold. First, a majority of participants did not finish implementing changes to code and class diagram in the first 35 minutes of the 50 minute experimental session,

and likely would not have done so even if the nudge had been issued at a later time. Since the system tests, by nature, test a complete implementation of the changes, it is natural for a participant to not proceed to altering system tests before the system under test is fully implemented. Second, the system tests, by consequence of the experimental software system being a complex simulation, require a reasonably involved understanding of system functionality which was not explained in the task document nor implemented in code which participants would naturally come into contact with. Thus, the system tests do not provide a good reference artifact for building a mental model and simultaneously require a mental model so sophisticated that it likely cannot be attained during the experimental session's limited time frame. Future studies of this scale should likely instead rely on unit tests or simple integration tests which can be more easily understood.

5.2.3 Research Question 3

> **RQ3** Can we find a difference in mental model accuracy between participants who used a Coding Assistance Tool and those who did not?

Since measuring the accuracy of someone's mental model directly is not possible due to the large variability in how individuals construct and apply their mental models, a proxy metric had to be established instead. For this, we evaluated the participants' answers to the post-questionnaire's comprehension section which was specifically designed so that someone with a more complete mental model would be able to answer more questions correctly.

We found that mean performance over all questions was below 0.5, meaning that participants on average were able to answer less than 50% of questions accurately. This is generally in line with the observation that the majority of participants were not able to finish the task, leaving significant portions of functionality unimplemented. We would thus expect a result like this, since participants will not have interacted with enough parts of the codebase to have a sufficiently formed mental model of relevant parts to answer the questionnaire. Alternatively, it is also possible that the questions were simply badly phrased or otherwise difficult to understand. Based on feedback given by participants during the experimental sessions however, and the kinds of clarifying questions they asked, we believe this to be unlikely.

Notably, we found that participants in the Copilot group formed a less accurate mental model, only answering about 36% of questions accurately (compared to 54% in the non-Copilot group). This finding too is generally in line with expectations, since less work on the codebase is actually performed by the participants in this group—instead, Copilot takes the active role in investigating existing implementations and generating new ones. Having less contact with the codebase thus means there are much fewer opportunities for a detailed mental model to form.

This interpretation is supported by the further observation that Copilot users not only performed worse in the post-questionnaire overall, they very specifically performed much worse on detail questions. Being able to answer, on average, only 1 detail question accurately implies that the mental model is extremely rudimentary and consists primarily of knowledge about overall system structure, with this knowledge also being less accurate than in the

non-Copilot group. This is likely explained by the same phenomenon as above: less active code contact (i.e., not simply reviewing Copilot suggestions) means that less surrounding implementation context gets incorporated into the mental model.

The overall takeaway is that yes, the Copilot group's participants in our study formed a very noticeably inferior mental model to those in the non-Copilot group. This implies that they are, with this mental model, less equipped to tackle future issues in the codebase themselves, likely less capable of communicating ideas about the system and consequently also less able to effectively supervise output of a CAT. Since participants in the Copilot group did not overall make less progress on the task, it stands to assume that using Copilot in some way replaces the missing parts of the mental model (which would agree with findings by Qiao et al. [23]). This causes reliance on the CAT, likely long-term, and it is not clear if the requisite mental model for properly diagnosing tricky issues and appropriately directing future expansion would ever develop even after sustained work on the same codebase.

5.2.4 Summary

In summary, we found that code is the single most important artifact upon which the mental model of our participants was built. While the class diagram was additionally utilized by many, it did not appear to have an impact on the accuracy of their final mental model. While we cannot say this with high certainty, as it could also be a consequence of the varying experience levels on our sample, we believe this could explain why many experienced programmers do not routinely utilize or perform maintenance on large class diagrams. It is possible that they themselves discovered (as their top-down comprehension improved) that utilizing a class diagram does not bring sufficient benefit to justify the maintenance cost.

In regards to system tests, we believe that their low utilization overall, and the fact that nobody started by exploring them, indicates that they require a suitably strong mental model to be formed already. In practice, this has two potential implications for their use: On the one hand, the programmer should consider delaying system test creation until they have developed a sufficiently advanced mental model. On the other hand though, intentionally authoring system tests "too early" as it were, might serve to surface inconsistencies in the mental model or implementation that can then be remedied.

Of using GitHub Copilot, we can say that it massively influences the programmer's workflow. We found that it leads to less multitasking, likely because participants are, instead of switching between multiple artifacts, switching between Copilot and the artifact in which they're reviewing its output. Even though participants did consistently review Copilot output in our study, they still formed a weaker mental model than those who did not use Copilot—in line with prior findings. We thus speculate that real-world use cases in which Copilot is left to write code largely unsupervised lead to even worse mental models on the part of the programmer. Consequently, such situations should ideally be avoided if long-term loss of critical codebase understanding is a non-goal.

5.3 Threats to Validity

This sections discusses the limitations of and potential issues with our study and its findings.

5.3.1 Internal Validity

Based on the observation that nearly none of the participants ended up working on the system tests without a nudge from us, it is easy to hypothesize that the complexity of the system tests relative to the rest of the tasks made to them too daunting to prioritize. One might also imagine that the limited time available to participants further biased the results away from the system test artifact. How likely this, or alternatively that most programmers would naturally deprioritize end-to-end tests, is to be the case is impossible to quantify based on our data.

Due to the rapid evolution in the space of CATs, the selection of LLMs available to GitHub Copilot users changed during the course of the study, and thus we were not able to use the same backing LLM for all relevant experimental sessions. Whilst we are not investigating the effects of LLM performance, and confirmed for all LLMs that they had no prior knowledge of the task, this does mean that the exact experimental conditions differed somewhat for participants in the same group. Simultaneously however, LLM performance sometimes varies starkly between individual sessions of the same backing LLM, so that we do not suspect this to be a significant factor for the measures we were investigating.

5.3.2 External Validity

While the participants in our study exhibited a range of self-reported programming experience, and this is the generally accepted measure for this variable, it is not possible to properly verify how broad a spectrum of experience this represents in reality—especially taking into account that none of the participants were programmers by profession.

This issue is further compounded by the very low sample size, which makes it impossible to draw any sort of robust, generalizable statistical conclusions from our results. Instead, as was the initial goal, this study serves as an exploratory device to inform the directions of further research.

Another major factor limiting broad applicability of the results is that our study was conducted at a university lab, in a controlled research environment. This setting lacks a number of important features an experienced programmer would generally have available to them in the field, including their personalized programming environment, preferred software and most importantly peers with knowledge about the software system under maintenance. This last point is especially relevant for how mental models are formed and shared (see LaToza et al. [18]). Consequently, our findings are likely to be most applicable to solo developers maintaining an unfamiliar system, which is not the general case in industry.

While PlantUML is the industry standard for representing UML diagrams and Kotlin is a widely used language, especially in enterprise settings, both (and especially the combination of them) are not necessarily representative of multi-artifact maintenance workflows in general. One major point to consider in this context is that much software is written without

the object-oriented paradigm, meaning the entire concept of a class diagram is not applicable and a different kind of diagram is likely to be used in practice—if one is used at all.

In regards to Coding Assistance Tools (CATs), our study captures the situation at a point in time when the tools themselves, their features and capabilities and the ways in which programmers apply them are still subject to rapid development and change. Thus, our conclusions in regards to how CATs affect the mental model are likely to apply only for a very limited period into the future.

5.3.3 Construct Validity

Our approach of measuring the participants' mental models using a questionnaire is supported by existing literature [25] and so is our approach in how the comprehension questions were chosen. Nonetheless, we did not have the means to validate out questionnaire. Consequently, it is not possible say with certainty that the questionnaire actually succeeds at measuring the mental model's accuracy in the way it was designed to do, potentially undermining our findings.

Our other major metric, time spent on each artifact, being measured based on the screen recordings runs the risk of not accurately capturing when an artifact is genuinely relevant to the participant's thought process. Especially when multiple artifacts are on screen simultaneously, this does not necessarily imply that they are both in active use. While we still believe that generally an artifact being placed on the screen by the programmer means it is in active use, we cannot properly verify that this was indeed always the case.

6 Concluding Remarks

6.1 Conclusion

In this thesis, we conducted a user study to examine how programmers organize their workflow when performing a programming task on a software system consisting of multiple artifacts, with and without using GitHub Copilot. In this context, we were very specifically interested in how the programmers' mental models affect and are affected by this process. We also anchored this relatively novel approach in the existing landscape of software comprehension, mental model and Coding Assistance Tool research.

To answer this, we investigated which artifacts they prioritized at the beginning of the implementation process, how time was distributed across artifacts overall, and how GitHub Copilot affected these questions and the accuracy of the programmers' mental models after finishing the task. We collected data about the participants' prior experience, their time distribution across the different artifacts, and assessed the state of their mental model in two different categories using a specially designed comprehension questionnaire after the task.

In analyzing the initial modification data, we found that participants consistently chose to begin by working on the class diagram and we posed the hypothesis that this is because the class diagram represents a higher level of abstraction where it is easier to start developing a mental model. We further saw that most participants also made initial modifications to code, indicating a desire the keep the artifacts synchronized.

From the overall time distribution data, we saw that participants in the non-Copilot group formed two distinct modes: some would reference the class diagram for nearly the entire duration of their session, others would make all necessary modifications once and then stop using it. This behavior was slightly correlated to prior software modeling experience, did however not affect the final mental model. We also found that participants in this group showed a significant tendency to multitask (i.e., work on more than one artifact at once).

We then saw however that participants in the Copilot group exhibited a more homogeneous workflow. Their artifact time distribution data expressed clear prioritization of code then class diagram then system tests. Using Copilot also decreased multitasking behavior, which we believe is potentially due to Copilot replacing a second artifact like the class diagram for purposes of, for instance, system comprehension.

With the comprehension questionnaire, which had questions pertaining to implementation details and to big-picture system logic, we confirmed prior findings that using Copilot decreases the accuracy of the programmer's mental model after the fact. We also observed that performance in detail questions was worse than performance in system questions

overall, but that this difference was much more pronounced for participants who did not use Copilot.

6.2 Future Work

One of the major weaknesses of our data, by virtue of us only having artifact usage data from screen recordings, is that we do not know when exactly a participant is actively using an artifact and when they merely have it on screen. Future work should incorporate eye-tracking data to mitigate this issue and repeat similar time distribution analyses.

Another possible major limitation of our data is the potential bias in initial artifact ordering introduced by the ordering of the task document. Future work should repeat a similar study but randomize the ordering of steps in the task document for each participant individually to mitigate this issue.

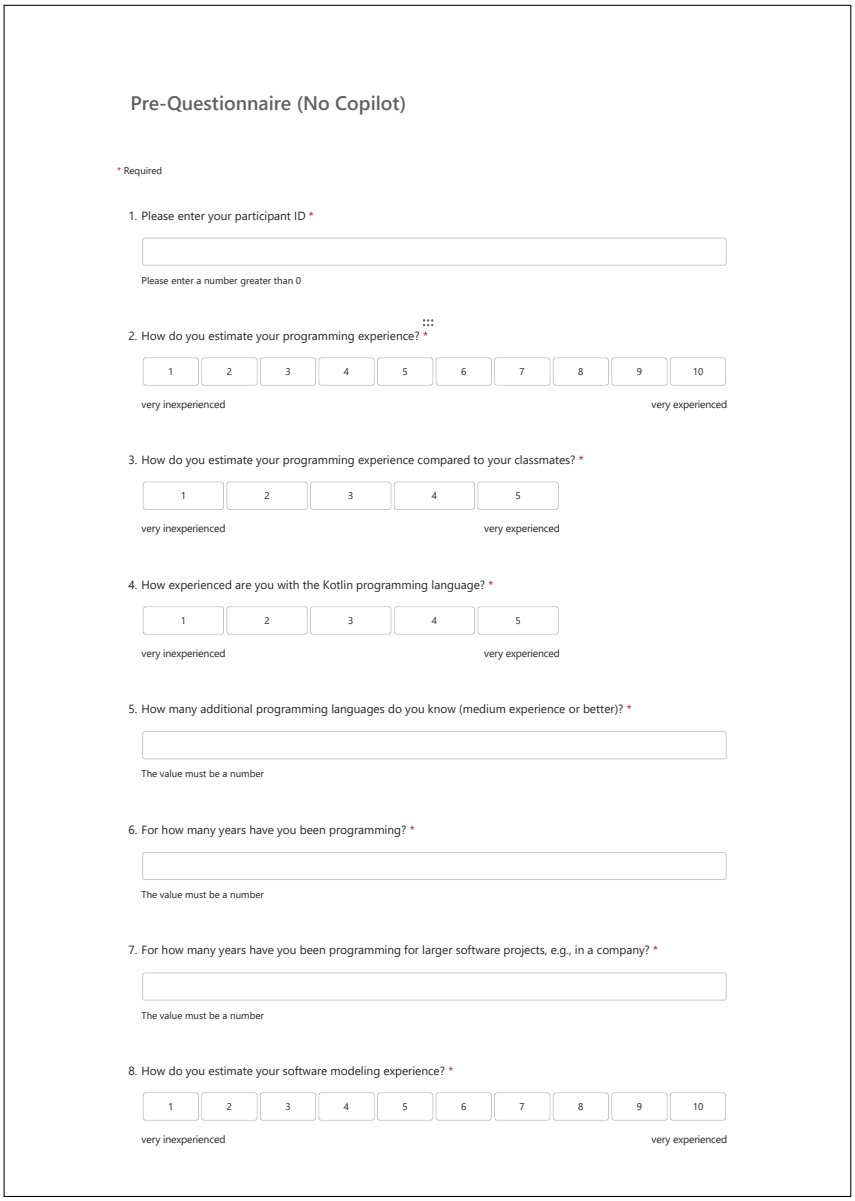
A last remaining weakness to address lies in the realm of artifact selection: the system test artifact is strongly biased against in our design, likely because it is too time-consuming and complicated to work on given the time available. Future work should thus consider replacing the system test artifact with unit or integration tests which require less intense system understanding to successfully modify. Alternatively, a study with similar artifact selection but much longer working time for participants might also alleviate the apparent bias towards code and class diagram.

Then, assuming the data itself is solid, it should receive better grounding in reality. To this end, further work should consider evaluating how successful a given artifact prioritization and workflow pattern is not only by mental model accuracy, but also by how correct and functional the actual implementation produced by participants is. This could be done in a reliable way by, for instance, running a test suite like was included with the template system in this study.

Overall, a larger-scale study with many more participants is needed to verify the findings of this exploratory experiment. To make the results of such a study more representative, it should also strive to include not just students but professional programmers with diverse backgrounds. Ideally, future work would also try to replicate the findings with other programming languages and perhaps other programming paradigms, helping further broaden applicability of the results.

Appendix

a.1 Pre-Questionnaire



Pre-Questionnaire (No Copilot)

* Required

1. Please enter your participant ID *

Please enter a number greater than 0

2. How do you estimate your programming experience? *

1 2 3 4 5 6 7 8 9 10

very inexperienced very experienced

3. How do you estimate your programming experience compared to your classmates? *

1 2 3 4 5

very inexperienced very experienced

4. How experienced are you with the Kotlin programming language? *

1 2 3 4 5

very inexperienced very experienced

5. How many additional programming languages do you know (medium experience or better)? *

The value must be a number

6. For how many years have you been programming? *

The value must be a number

7. For how many years have you been programming for larger software projects, e.g., in a company? *

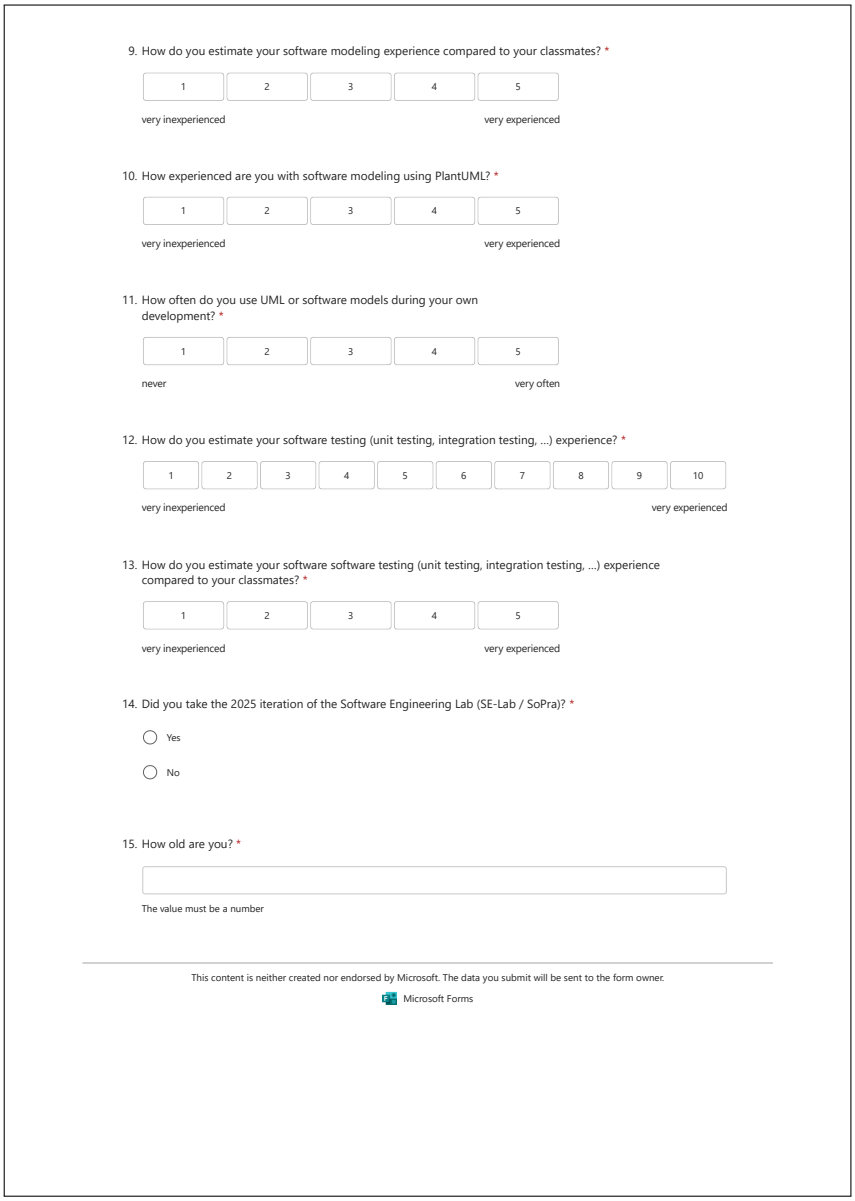
The value must be a number

8. How do you estimate your software modeling experience? *

1 2 3 4 5 6 7 8 9 10

very inexperienced very experienced

Figure A.1: Pre-Questionnaire for the Non-Copilot group (Page 1)



9. How do you estimate your software modeling experience compared to your classmates? *

1 2 3 4 5

very inexperienced very experienced

10. How experienced are you with software modeling using PlantUML? *

1 2 3 4 5

very inexperienced very experienced

11. How often do you use UML or software models during your own development? *

1 2 3 4 5

never very often

12. How do you estimate your software testing (unit testing, integration testing, ...) experience? *

1 2 3 4 5 6 7 8 9 10

very inexperienced very experienced

13. How do you estimate your software software testing (unit testing, integration testing, ...) experience compared to your classmates? *

1 2 3 4 5

very inexperienced very experienced

14. Did you take the 2025 iteration of the Software Engineering Lab (SE-Lab / SoPra)? *

Yes

No

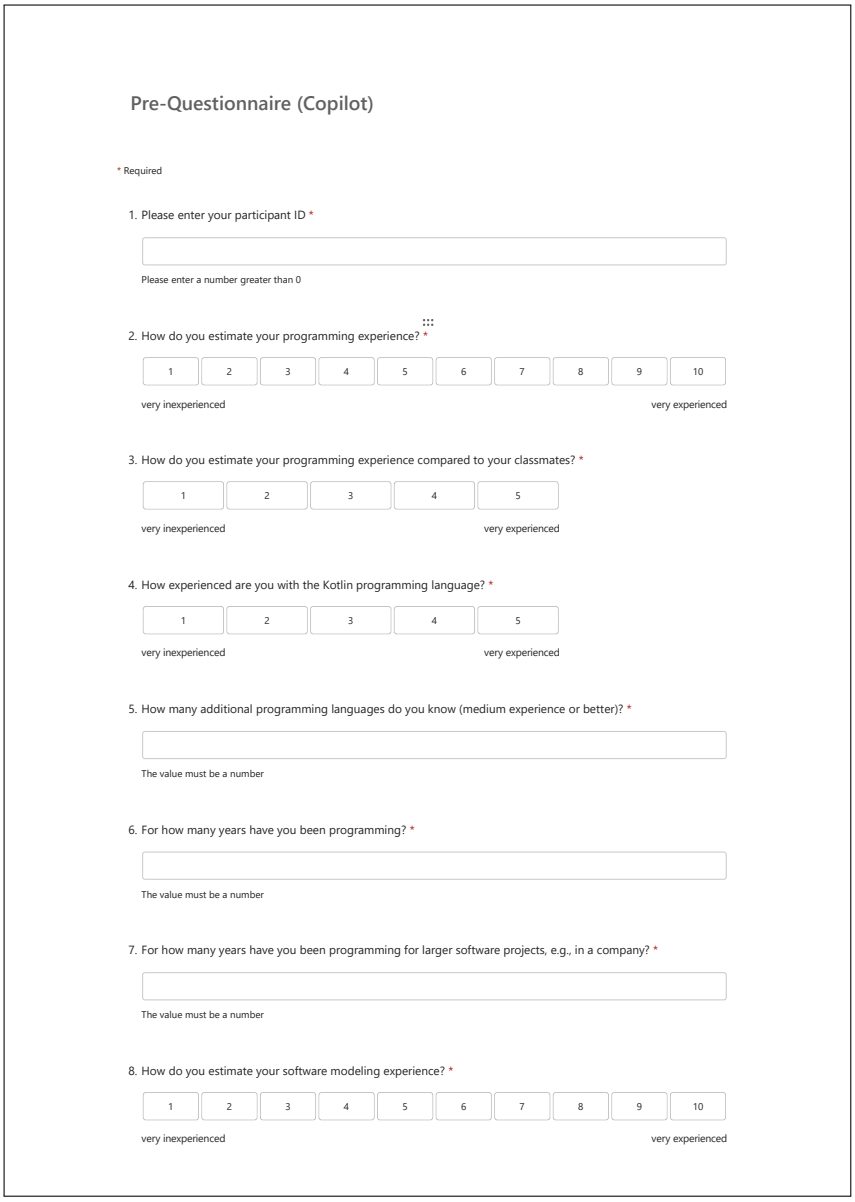
15. How old are you? *

The value must be a number

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms

Figure A.2: Pre-Questionnaire for the Non-Copilot group (Page 2)



Pre-Questionnaire (Copilot)

* Required

1. Please enter your participant ID *

Please enter a number greater than 0

2. How do you estimate your programming experience? *

1 2 3 4 5 6 7 8 9 10

very inexperienced very experienced

3. How do you estimate your programming experience compared to your classmates? *

1 2 3 4 5

very inexperienced very experienced

4. How experienced are you with the Kotlin programming language? *

1 2 3 4 5

very inexperienced very experienced

5. How many additional programming languages do you know (medium experience or better)? *

The value must be a number

6. For how many years have you been programming? *

The value must be a number

7. For how many years have you been programming for larger software projects, e.g., in a company? *

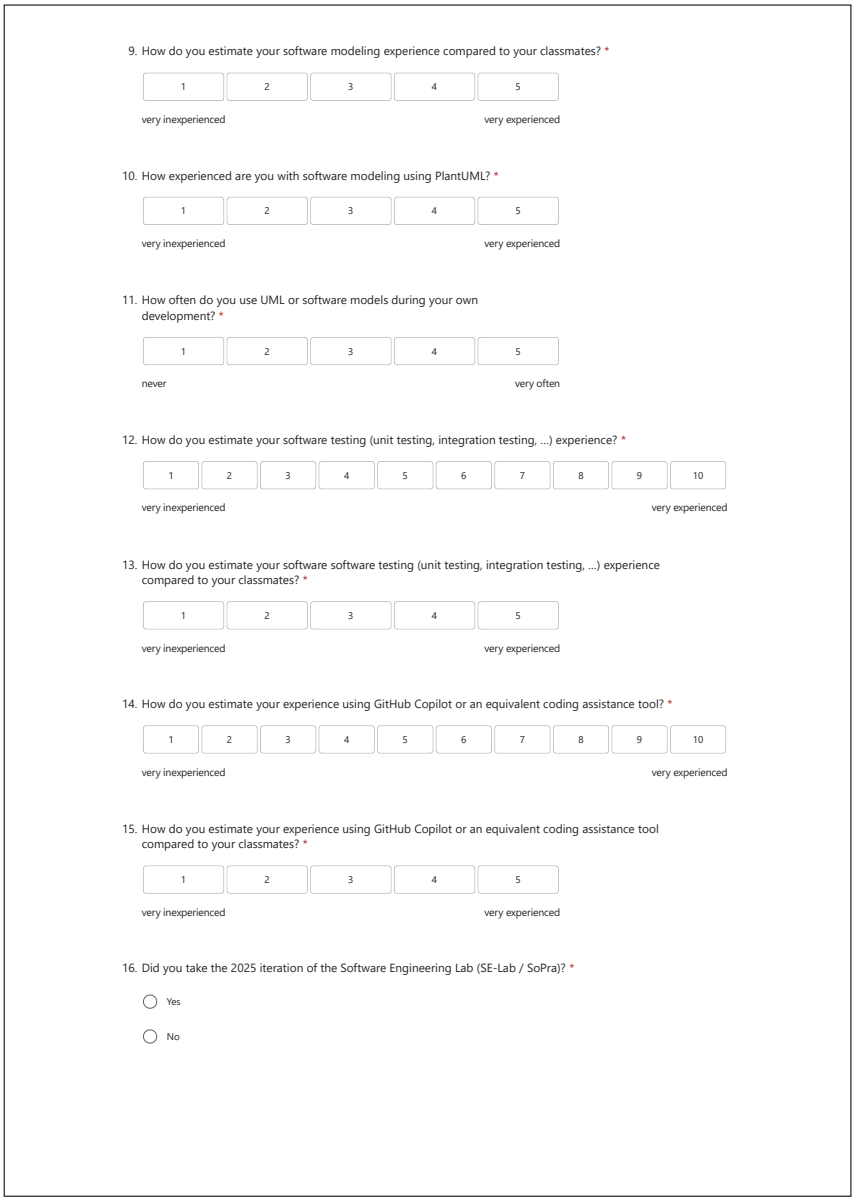
The value must be a number

8. How do you estimate your software modeling experience? *

1 2 3 4 5 6 7 8 9 10

very inexperienced very experienced

Figure A.3: Pre-Questionnaire for the Copilot group (Page 1)



9. How do you estimate your software modeling experience compared to your classmates? *

1	2	3	4	5

very inexperienced — very experienced

10. How experienced are you with software modeling using PlantUML? *

1	2	3	4	5

very inexperienced — very experienced

11. How often do you use UML or software models during your own development? *

1	2	3	4	5

never — very often

12. How do you estimate your software testing (unit testing, integration testing, ...) experience? *

1	2	3	4	5	6	7	8	9	10

very inexperienced — very experienced

13. How do you estimate your software software testing (unit testing, integration testing, ...) experience compared to your classmates? *

1	2	3	4	5

very inexperienced — very experienced

14. How do you estimate your experience using GitHub Copilot or an equivalent coding assistance tool? *

1	2	3	4	5	6	7	8	9	10

very inexperienced — very experienced

15. How do you estimate your experience using GitHub Copilot or an equivalent coding assistance tool compared to your classmates? *

1	2	3	4	5

very inexperienced — very experienced

16. Did you take the 2025 iteration of the Software Engineering Lab (SE-Lab / SoPra)? *

- ◯ Yes
- ◯ No

Figure A.4: Pre-Questionnaire for the Copilot group (Page 2)

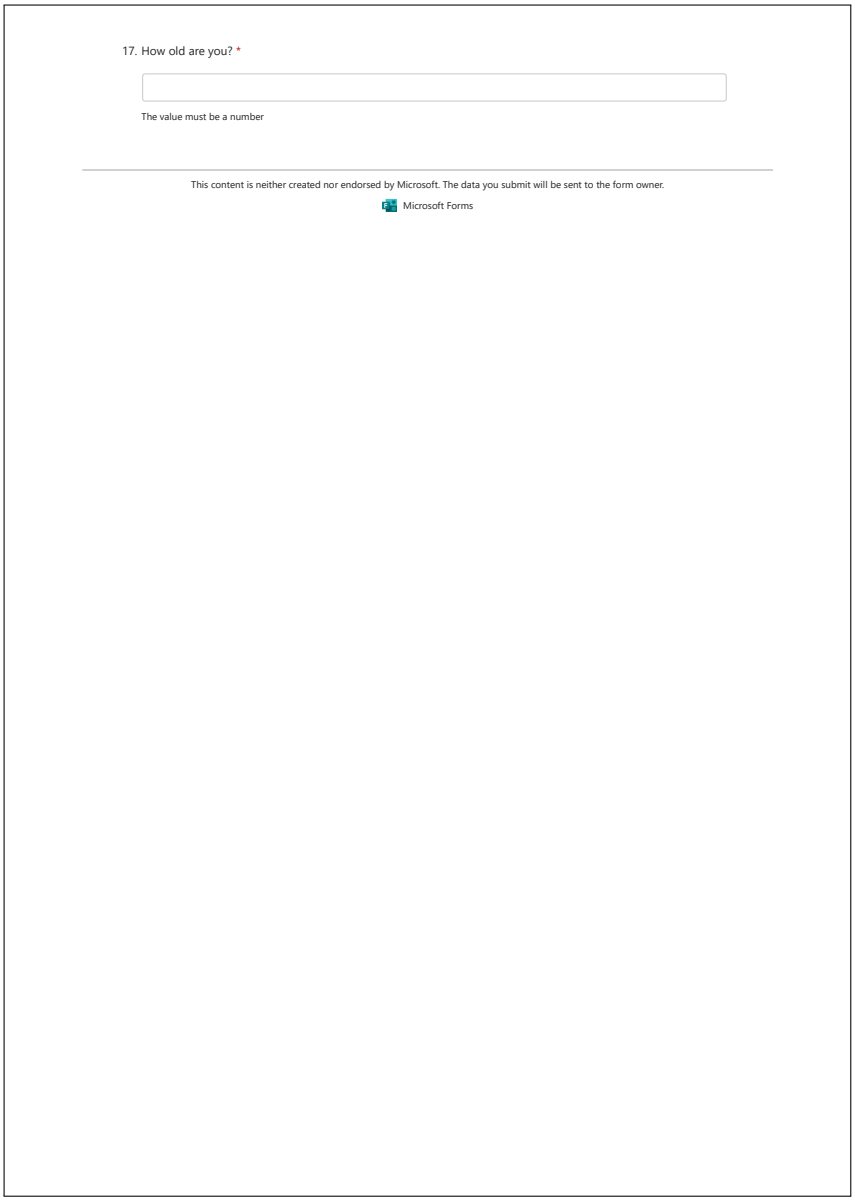
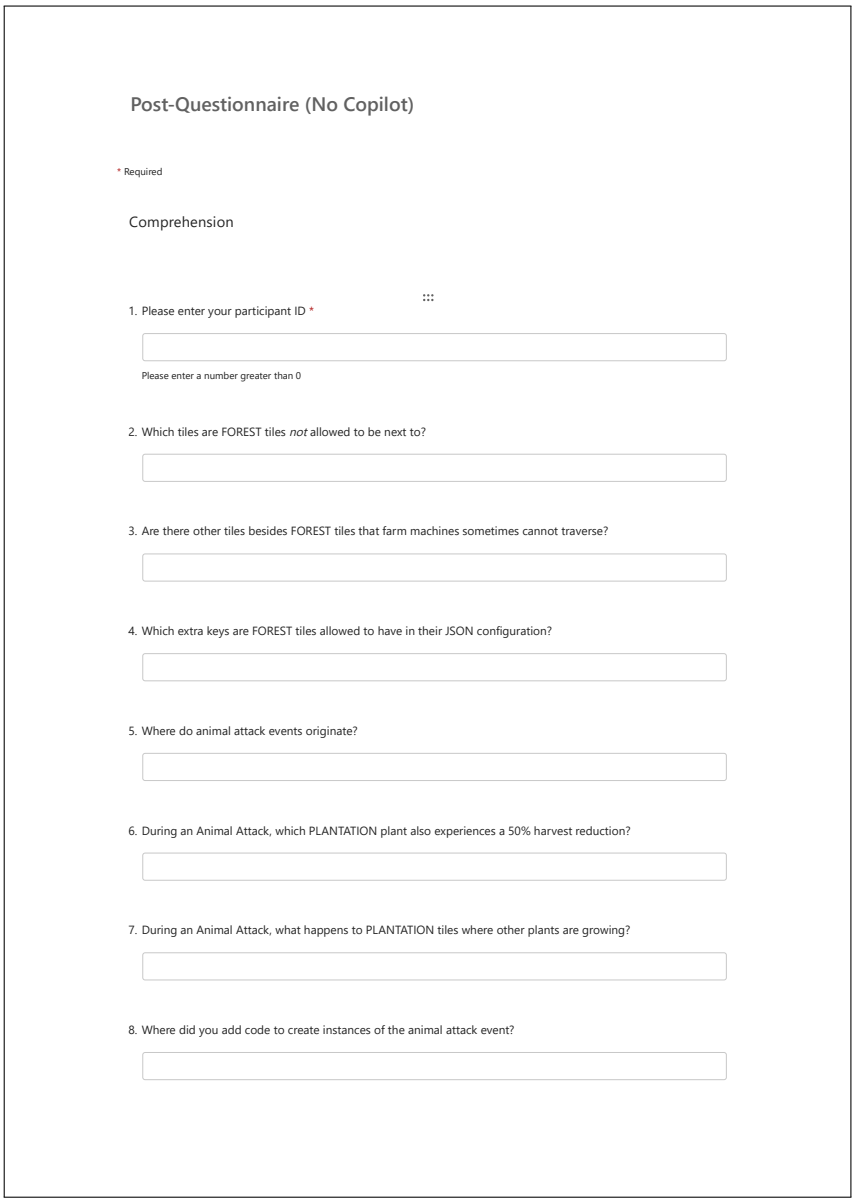


Figure A.5: Pre-Questionnaire for the Copilot group (Page 3)

a.2 Post-Questionnaire



Post-Questionnaire (No Copilot)

* Required

Comprehension

1. Please enter your participant ID *

Please enter a number greater than 0

2. Which tiles are FOREST tiles *not* allowed to be next to?

3. Are there other tiles besides FOREST tiles that farm machines sometimes cannot traverse?

4. Which extra keys are FOREST tiles allowed to have in their JSON configuration?

5. Where do animal attack events originate?

6. During an Animal Attack, which PLANTATION plant also experiences a 50% harvest reduction?

7. During an Animal Attack, what happens to PLANTATION tiles where other plants are growing?

8. Where did you add code to create instances of the animal attack event?

Figure A.6: Post-Questionnaire for the Non-Copilot group (Page 1)

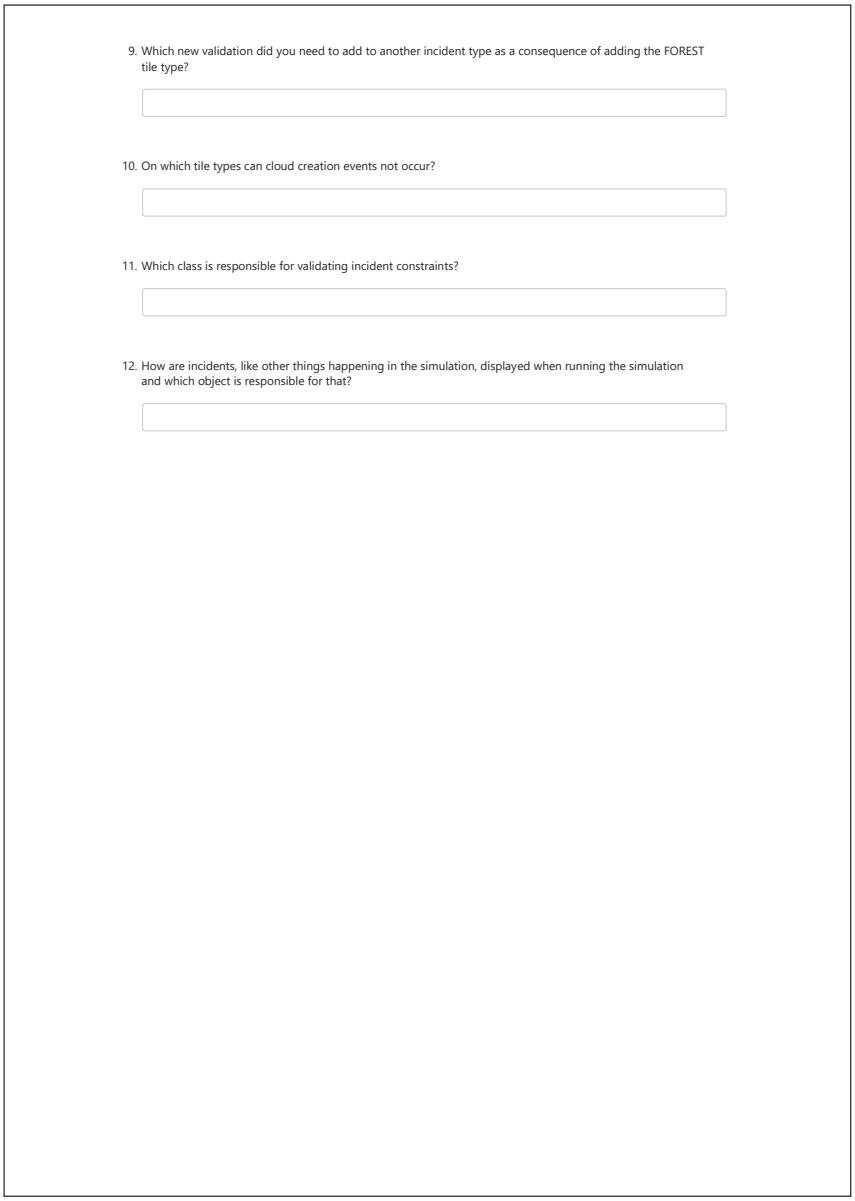
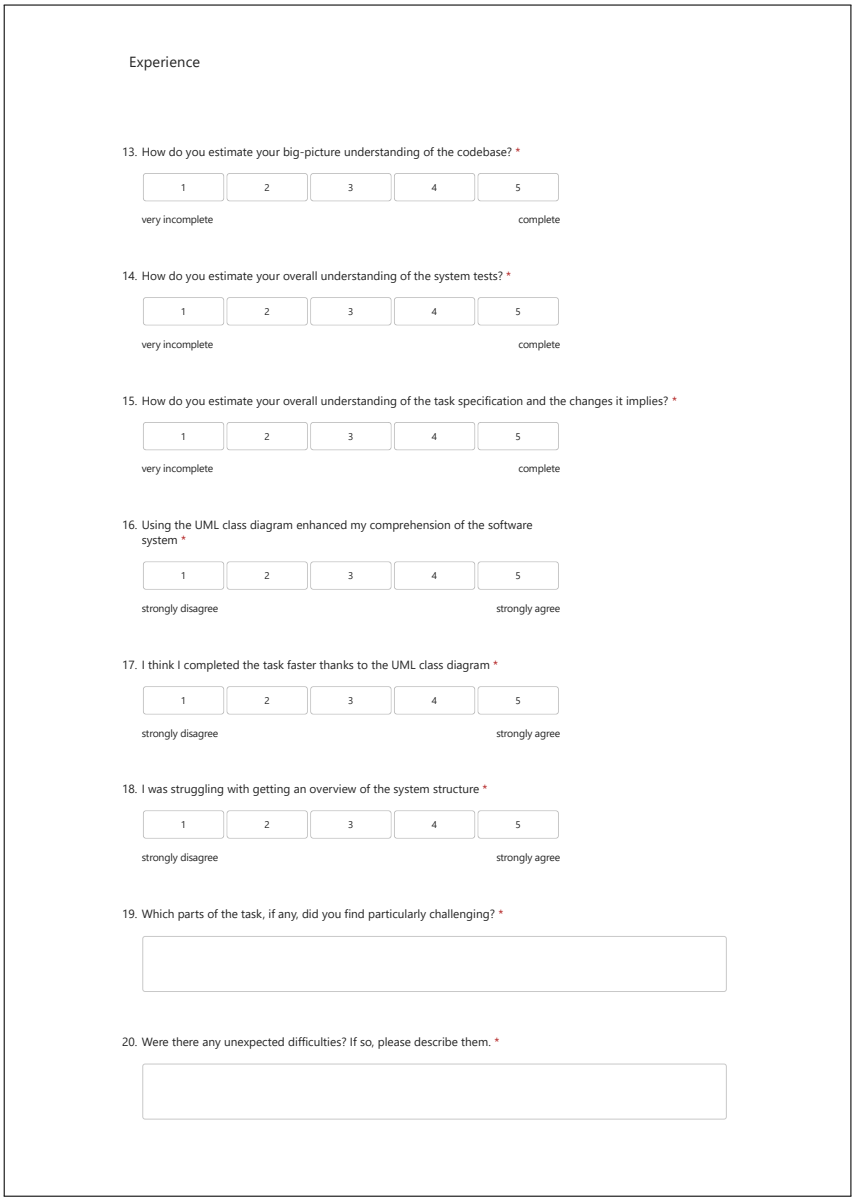


Figure A.7: Post-Questionnaire for the Non-Copilot group (Page 2)



Experience

13. How do you estimate your big-picture understanding of the codebase? *

1 2 3 4 5

very incomplete complete

14. How do you estimate your overall understanding of the system tests? *

1 2 3 4 5

very incomplete complete

15. How do you estimate your overall understanding of the task specification and the changes it implies? *

1 2 3 4 5

very incomplete complete

16. Using the UML class diagram enhanced my comprehension of the software system *

1 2 3 4 5

strongly disagree strongly agree

17. I think I completed the task faster thanks to the UML class diagram *

1 2 3 4 5

strongly disagree strongly agree

18. I was struggling with getting an overview of the system structure *

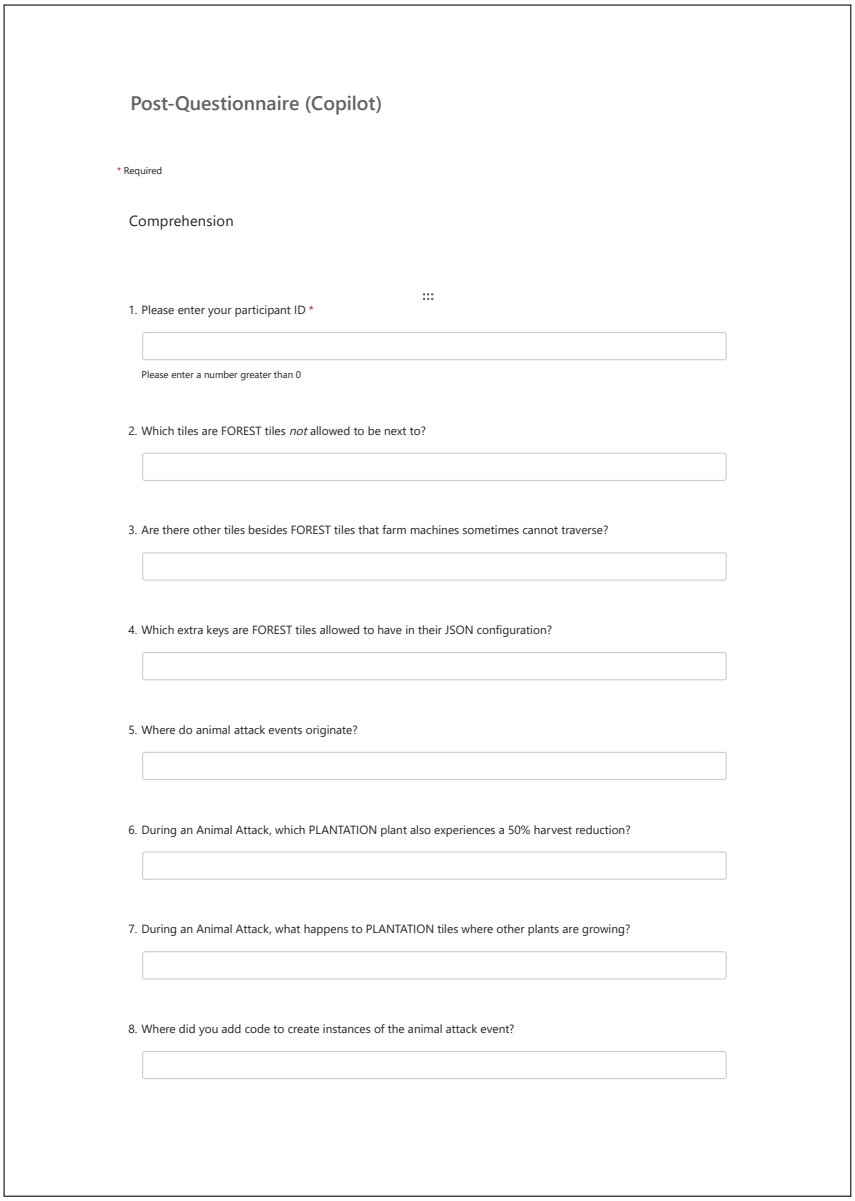
1 2 3 4 5

strongly disagree strongly agree

19. Which parts of the task, if any, did you find particularly challenging? *

20. Were there any unexpected difficulties? If so, please describe them. *

Figure A.8: Post-Questionnaire for the Non-Copilot group (Page 3)



Post-Questionnaire (Copilot)

* Required

Comprehension

1. Please enter your participant ID *

Please enter a number greater than 0

2. Which tiles are FOREST tiles *not* allowed to be next to?

3. Are there other tiles besides FOREST tiles that farm machines sometimes cannot traverse?

4. Which extra keys are FOREST tiles allowed to have in their JSON configuration?

5. Where do animal attack events originate?

6. During an Animal Attack, which PLANTATION plant also experiences a 50% harvest reduction?

7. During an Animal Attack, what happens to PLANTATION tiles where other plants are growing?

8. Where did you add code to create instances of the animal attack event?

Figure A.9: Post-Questionnaire for the Copilot group (Page 1)

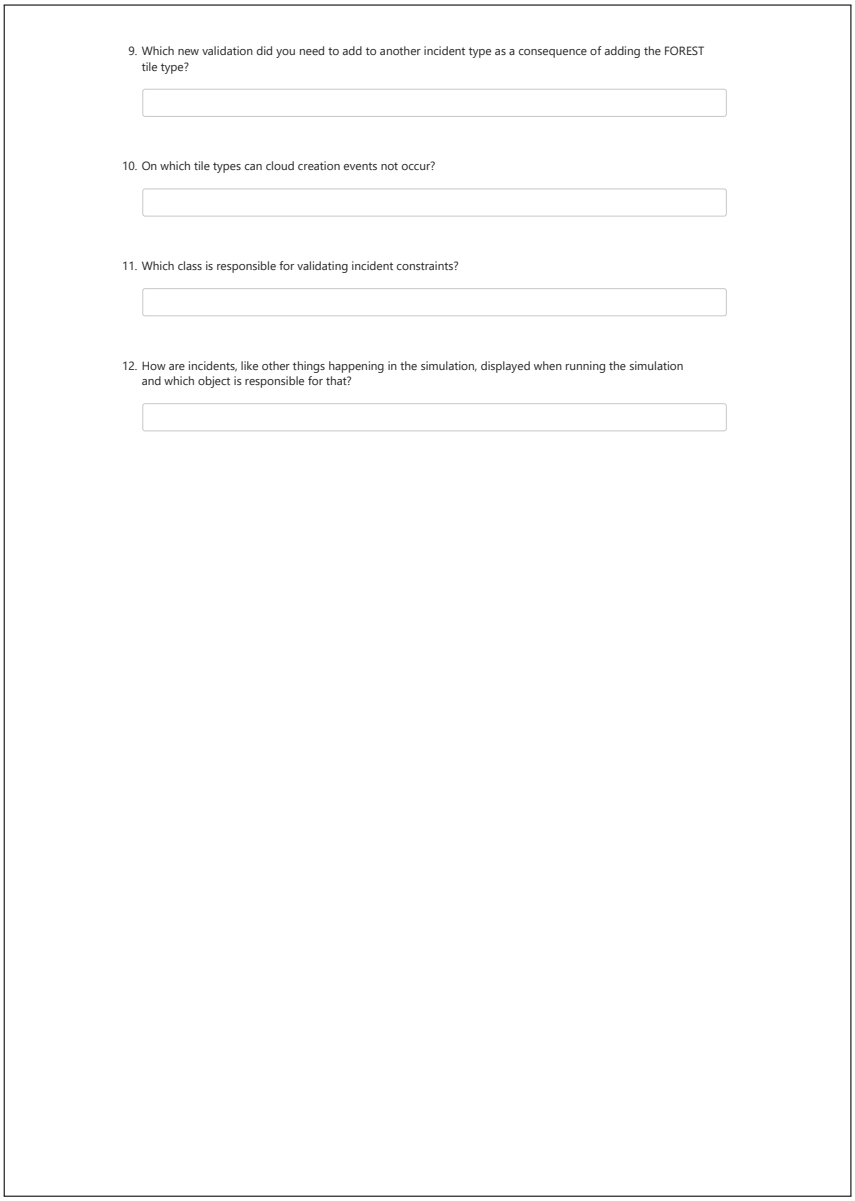
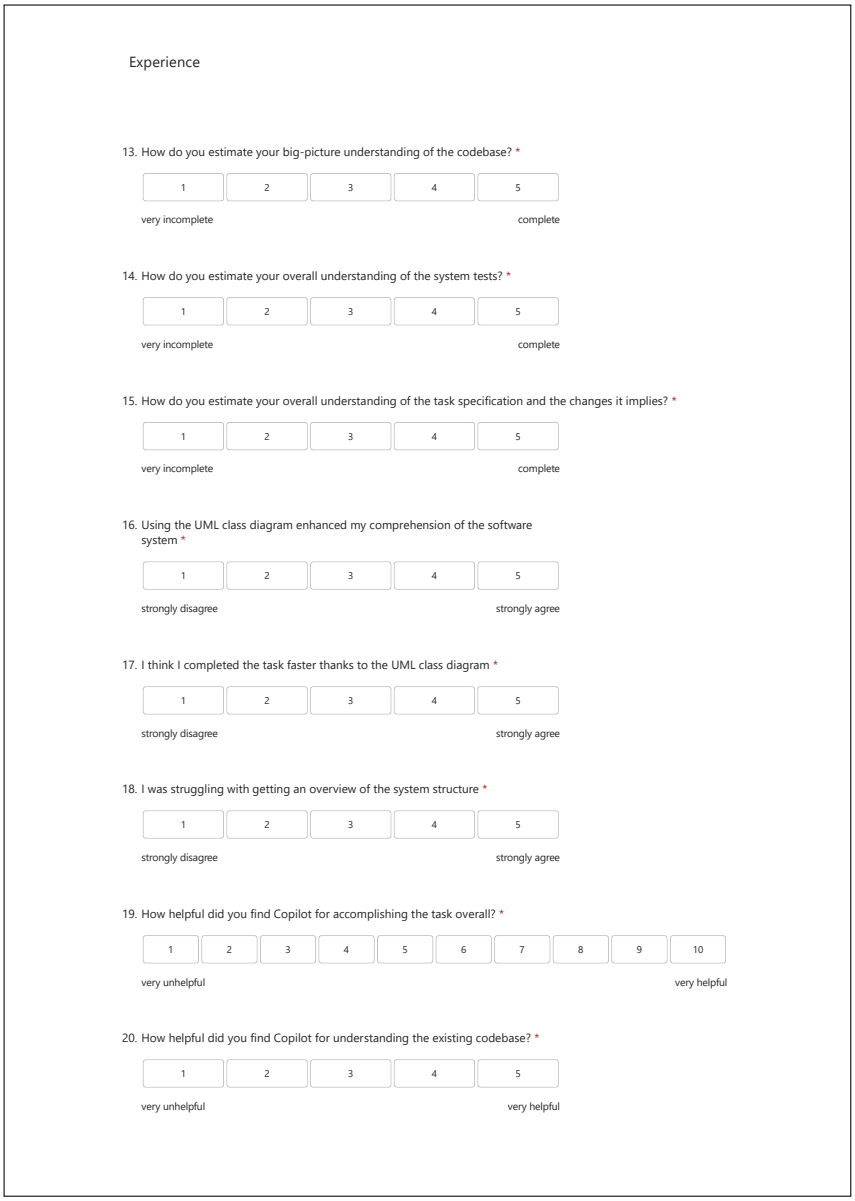


Figure A.10: Post-Questionnaire for the Copilot group (Page 2)



Experience

13. How do you estimate your big-picture understanding of the codebase? *

1 2 3 4 5

very incomplete complete

14. How do you estimate your overall understanding of the system tests? *

1 2 3 4 5

very incomplete complete

15. How do you estimate your overall understanding of the task specification and the changes it implies? *

1 2 3 4 5

very incomplete complete

16. Using the UML class diagram enhanced my comprehension of the software system *

1 2 3 4 5

strongly disagree strongly agree

17. I think I completed the task faster thanks to the UML class diagram *

1 2 3 4 5

strongly disagree strongly agree

18. I was struggling with getting an overview of the system structure *

1 2 3 4 5

strongly disagree strongly agree

19. How helpful did you find Copilot for accomplishing the task overall? *

1 2 3 4 5 6 7 8 9 10

very unhelpful very helpful

20. How helpful did you find Copilot for understanding the existing codebase? *

1 2 3 4 5

very unhelpful very helpful

Figure A.11: Post-Questionnaire for the Copilot group (Page 3)

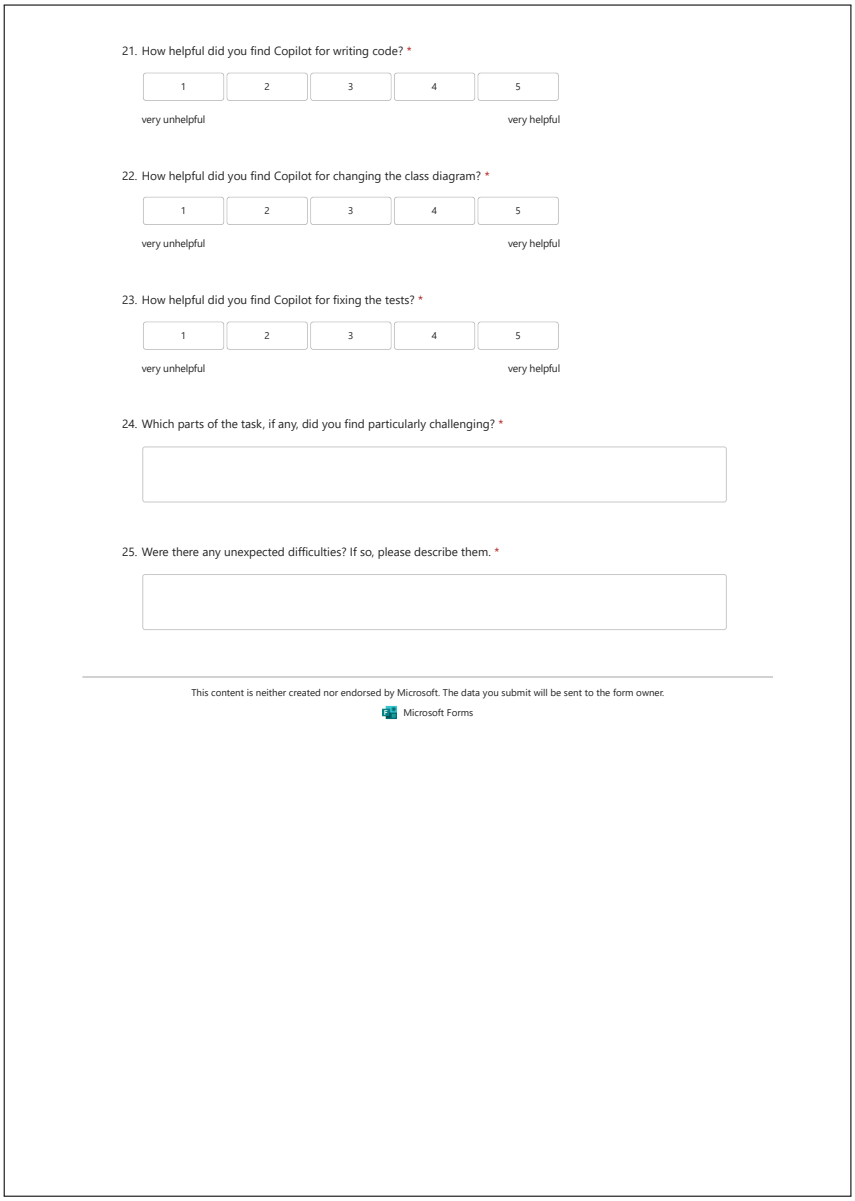
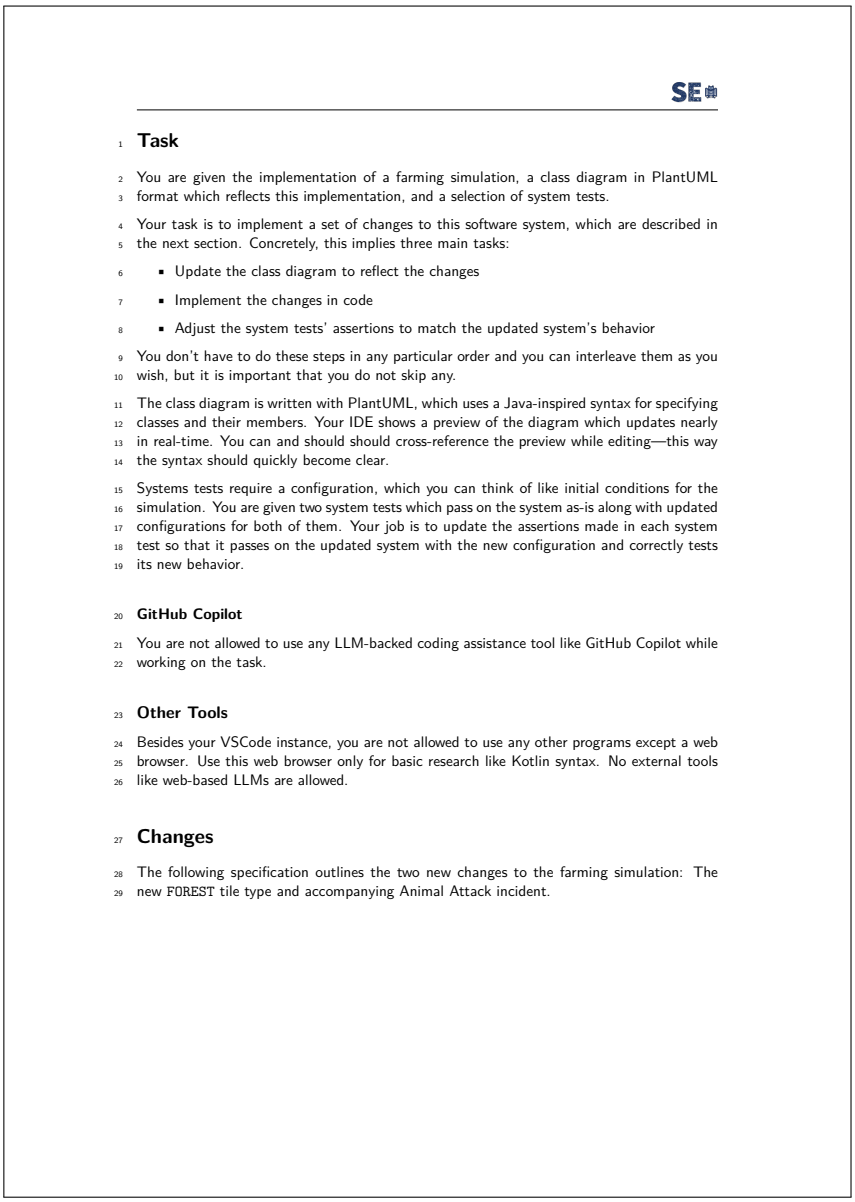


Figure A.12: Post-Questionnaire for the Copilot group (Page 4)

a.3 Task Specification



Task

You are given the implementation of a farming simulation, a class diagram in PlantUML format which reflects this implementation, and a selection of system tests.

Your task is to implement a set of changes to this software system, which are described in the next section. Concretely, this implies three main tasks:

- Update the class diagram to reflect the changes
- Implement the changes in code
- Adjust the system tests' assertions to match the updated system's behavior

You don't have to do these steps in any particular order and you can interleave them as you wish, but it is important that you do not skip any.

The class diagram is written with PlantUML, which uses a Java-inspired syntax for specifying classes and their members. Your IDE shows a preview of the diagram which updates nearly in real-time. You can and should should cross-reference the preview while editing—this way the syntax should quickly become clear.

Systems tests require a configuration, which you can think of like initial conditions for the simulation. You are given two system tests which pass on the system as-is along with updated configurations for both of them. Your job is to update the assertions made in each system test so that it passes on the updated system with the new configuration and correctly tests its new behavior.

GitHub Copilot

You are not allowed to use any LLM-backed coding assistance tool like GitHub Copilot while working on the task.

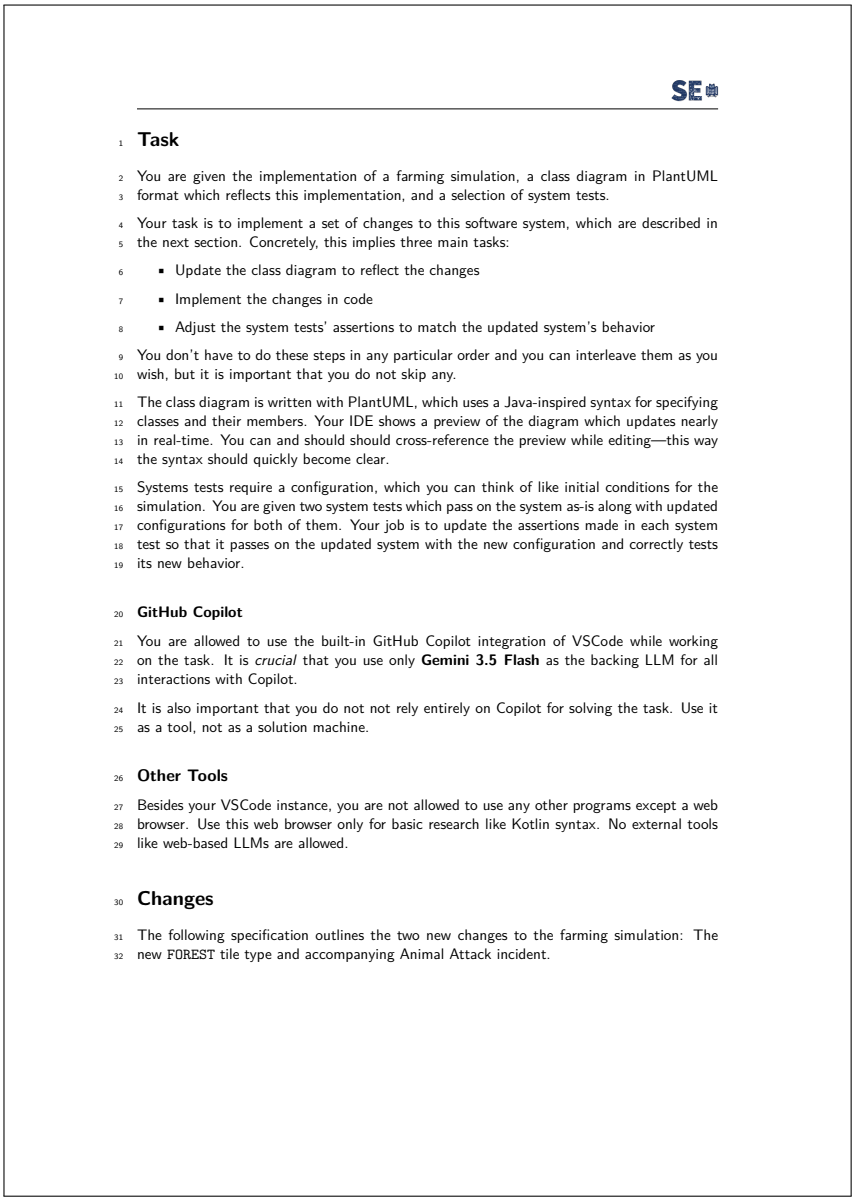
Other Tools

Besides your VSCode instance, you are not allowed to use any other programs except a web browser. Use this web browser only for basic research like Kotlin syntax. No external tools like web-based LLMs are allowed.

Changes

The following specification outlines the two new changes to the farming simulation: The new `FOREST` tile type and accompanying Animal Attack incident.

Figure A.13: Task specification for participants in the Non-Copilot Group (Page 1)



SE

Task

You are given the implementation of a farming simulation, a class diagram in PlantUML format which reflects this implementation, and a selection of system tests.

Your task is to implement a set of changes to this software system, which are described in the next section. Concretely, this implies three main tasks:

- Update the class diagram to reflect the changes
- Implement the changes in code
- Adjust the system tests' assertions to match the updated system's behavior

You don't have to do these steps in any particular order and you can interleave them as you wish, but it is important that you do not skip any.

The class diagram is written with PlantUML, which uses a Java-inspired syntax for specifying classes and their members. Your IDE shows a preview of the diagram which updates nearly in real-time. You can and should should cross-reference the preview while editing—this way the syntax should quickly become clear.

Systems tests require a configuration, which you can think of like initial conditions for the simulation. You are given two system tests which pass on the system as-is along with updated configurations for both of them. Your job is to update the assertions made in each system test so that it passes on the updated system with the new configuration and correctly tests its new behavior.

GitHub Copilot

You are allowed to use the built-in GitHub Copilot integration of VSCode while working on the task. It is *crucial* that you use only **Gemini 3.5 Flash** as the backing LLM for all interactions with Copilot.

It is also important that you do not not rely entirely on Copilot for solving the task. Use it as a tool, not as a solution machine.

Other Tools

Besides your VSCode instance, you are not allowed to use any other programs except a web browser. Use this web browser only for basic research like Kotlin syntax. No external tools like web-based LLMs are allowed.

Changes

The following specification outlines the two new changes to the farming simulation: The new `FOREST` tile type and accompanying Animal Attack incident.

Figure A.14: Task specification for participants in the Copilot Group (Page 1)

Parsing and Validation

All the new and changed functionality which will subsequently be introduced must, in addition to their explicitly specified features, also support being parsed—including schema and constraint validation.

`FOREST` Tiles

The world of the farming simulation is separated into tiles. These tiles can be of different types, which each have different properties. For instance, clouds (part of the simulation's weather system) dissipate when they move onto a `VILLAGE` tile and plants can only exist on `FIELD` or `PLANTATION` tiles. You need to add a new type of tile into the simulation: `FOREST` tiles.

`FOREST` tiles are places where forests grow, and forest animals can originate and cause havoc on adjoining `FIELD` and `PLANTATION` tiles. Both square and octagonal tiles can be `FOREST` tiles. They can adjoin all but `VILLAGE` tiles.

Farm machines cannot traverse `FOREST` tiles as the trees stand too close together.

Like most other tile types, `FOREST` tiles can have airflow.

Animal Attack

Another feature of the simulation is incidents. These are events which happen at predetermined ticks in the simulation (as defined in their configuration) and can change any number of things about the simulation state. For instance, a cloud creation incident causes new clouds to form on its assigned tile. You need a new type of incident into the simulation: animal attacks.

Animal attacks originate on `FOREST` tiles. Many animals living on these tiles struggle with finding enough food and water to survive. As their habitat shrinks, they become desperate to find food and encroach on the vast `FIELD` and `PLANTATION` tiles. An animal attack is an incident that impacts a tile and a specified radius around the tile, for which it has a `location` and `radius` property. It must be validated during the initialization of the simulation that at least one of the impacted tiles is a `FOREST` at the moment of incident. It lasts only for a single tick, in which it causes animals to break out from all impacted `FOREST` tiles and attack adjoining `FIELD` and `PLANTATION` tiles. They stampede through `FIELD` tiles and eat the plants, which causes massive damage for each incident. Therefore, the harvest in `FIELD` tiles is reduced to half. As `GRAPE`s are hanging low on the plants, the effects are as devastating, so the harvest is reduced to half. Other plantation trees such as apples, almonds and cherries are less affected. Here, the animals eat the grass, which resets the need for `MOWING` in this tick and the next, and eat 10% of the fruits. The effects occur only once per incident and per `FIELD` and `PLANTATION` tile, no matter how many affected `FOREST` tiles are adjoining.

Figure A.15: Task specification for participants (Page 2, both groups)

a.4 Consent Form

Welcome to Our Study on Multi-Artifact Software Maintenance

Thank you for your interest in participating in our research study! This study investigates how software developers evolve and maintain a software system consisting of multiple artifacts (like code, tests and diagrams). In this context, we are especially interested in how the developer's mental model is formed and adapted. Our overall goal is to understand how the different artifacts contribute to this process and how developers apply their mental model to them.

What You Will Do

In this study, you will be given an existing software system and have to implement a change to it. Concretely this means you have to:

- Make changes to the system's existing class diagram to reflect new functionality
- Understand and update the existing code to match a new specification
- Adjust the assertions of some end-to-end tests to match this new behavior

The study will take approximately **60 minutes** to complete.

Data Collection

We collect the following data:

- Your answers in the pre- and post-questionnaire
- The state of the codebase after you completed the task
- During the study, we record your screen and the audio of the room you'll be working in. This is to let us capture your thoughts about the task and conversations with us.
- We also use an eye tracker during the study, which lets us record timestamped data of where your gaze was focused on the screen for the duration of the task.
- Depending on your group, we also record timestamped data of precisely how and when you interact with GitHub Copilot.

Data Protection and Privacy

Your privacy is important to us. Here is how we handle your data:

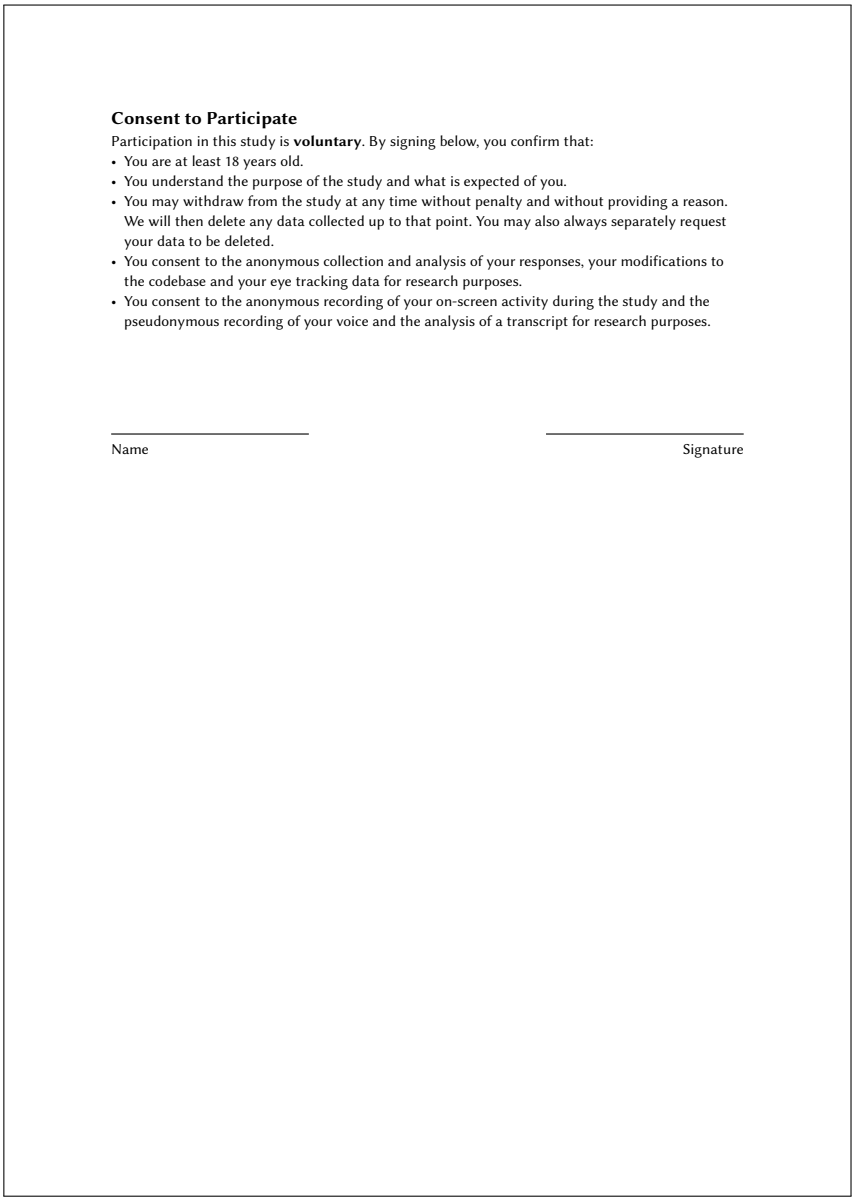
- All data is stored anonymously and securely on servers controlled by the university.
- The data is only accessible to the research team and will be used solely for academic research.
- The anonymized data, with the exception of your audio recordings, may be published in academic papers or presentations.
- The only personally identifiable information (PII) that will be collected is the room audio recording. We will transcribe this with a tool which we run locally—none of your data will leave our control.

  For analyzing your data, we rely on this transcript where possible. The recordings themselves will only be kept as backup and never shared with anyone outside our research team.

If you have any questions about this study, your rights as a participant, or how your data will be used, you can always contact us:

Chair of Software Engineering
Saarland University
lily.busche@uni-saarland.de

Figure A.16: Consent form for participating in the experiment (Page 1)



Consent to Participate

Participation in this study is **voluntary**. By signing below, you confirm that:

- You are at least 18 years old.
- You understand the purpose of the study and what is expected of you.
- You may withdraw from the study at any time without penalty and without providing a reason. We will then delete any data collected up to that point. You may also always separately request your data to be deleted.
- You consent to the anonymous collection and analysis of your responses, your modifications to the codebase and your eye tracking data for research purposes.
- You consent to the anonymous recording of your on-screen activity during the study and the pseudonymous recording of your voice and the analysis of a transcript for research purposes.

Name

Signature

Figure A.17: Consent form for participating in the experiment (Page 2)

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used:

- Google Scholar Labs[1] was used as a starting point during some research sessions. The sources suggested were thoroughly verified for relevance and quality just like any other.
- Google Gemini 3.6 Flash[2] was used for basic code completion while writing the Python code for data analysis. Gemini was used with its separate chat interface, not as an agent. It did not generate any of the scripts in their entirety (not even close) and was used primarily to offset unfamiliarity with library APIs. All of Gemini's output was verified thoroughly just like I would have proceeded for code review on a pull request.

No further generative digital assistants were involved in the development of the thesis' content or the writing of its text. We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

1 https://scholar.google.com/scholar_labs/search
2 https://gemini.google.com/app

Bibliography

[1] Tarek Alakmeh, Norman Anderson, Victoria Jackson, Guilherme Vaz Pereira, Umit Akirmak, Anthony Estey, Rafael Prikladnicki, Andre Van Der Hoek, Margaret-Anne Storey, and Thomas Fritz. "Grasping AI reliance in program comprehension and coding through the AIRELI persona taxonomy." In: *Proceedings of the 2026 34th IEEE/ACM International Conference on Program Comprehension*. 2026, pp. 123–134.

[2] Sebastian Baltes and Stephan Diehl. "Sketches and diagrams in practice." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, 530–541. ISBN: 9781450330565. DOI: 10.1145/2635868.2635891. URL: https://doi.org/10.1145/2635868.2635891.

[3] Hans Christian Benestad, Bente Anda, and Erik Arisholm. "Understanding software maintenance and evolution by analyzing individual changes: a literature review." In: *Journal of Software Maintenance and Evolution: Research and Practice* 21.6 (2009), pp. 349–378.

[4] Michelle Brachman, Siya Kunde, Sarah Miller, Ana Fucs, Samantha Dempsey, Jamie Jabbour, and Werner Geyer. "Building Appropriate Mental Models: What Users Know and Want to Know about an Agentic AI Chatbot." In: *Proceedings of the 30th International Conference on Intelligent User Interfaces*. 2025, pp. 247–264.

[5] Giuseppe Desolda, Andrea Esposito, Francesco Greco, Cesare Tucci, Paolo Buono, and Antonio Piccinno. "Understanding User Mental Models in AI-Driven Code Completion Tools: Insights from an Elicitation Study." In: *arXiv preprint arXiv:2502.02194* (2025).

[6] Yash Mohan Dhanore. *The Impact of Generative AI Tools in Open-Source Software Development*. 2025.

[7] Wojciech J Dzidek, Erik Arisholm, and Lionel C Briand. "A realistic empirical evaluation of the costs and benefits of UML in software maintenance." In: *IEEE Transactions on software engineering* 34.3 (2008), pp. 407–432.

[8] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. "Measuring programming experience." In: *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE. 2012, pp. 73–82.

[9] Ana M Fernández-Sáez, Danilo Caivano, Marcela Genero, and Michel RV Chaudron. "On the use of UML documentation in software maintenance: Results from a survey in industry." In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2015, pp. 292–301.

[10] Ana M. Fernández-Sáez, Michel R. V. Chaudron, Marcela Genero, and Isabel Ramos. "Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance? a controlled experiment." In: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. EASE '13. Porto de Galinhas, Brazil: Association for Computing Machinery, 2013, 60–71. ISBN: 9781450318488. DOI: 10.1145/2460999.2461008. URL: https://doi.org/10.1145/2460999.2461008.

[11] Ana M Fernández-Sáez, Marcela Genero, Danilo Caivano, and Michel RV Chaudron. "Does the level of detail of UML diagrams affect the maintainability of source code?: a family of experiments." In: *Empirical Software Engineering* 21.1 (2016), pp. 212–259.

[12] Olivier Friard and Marco Gamba. "BORIS: a free, versatile open-source event-logging software for video/audio coding and ive observations." In: *Methods in Ecology and Evolution* (2016), pp. 1324–1330. ISSN: 2041-210X. DOI: 10.1111/2041-210X.12584. URL: http://dx.doi.org/10.1111/2041-210X.12584.

[13] Carmine Gravino, Giuseppe Scanniello, and Genoveffa Tortora. "Source-code comprehension tasks supported by UML design models: Results from a controlled experiment and a differentiated replication." In: *Journal of Visual Languages & Computing* 28 (2015), pp. 23–38.

[14] Dean Hendrix, James H Cross, and Saeed Maghsoodloo. "The effectiveness of control structure diagrams in source code comprehension activities." In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 463–477.

[15] Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. "Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025." In: *arXiv preprint arXiv:2512.14012* (2025).

[16] Kai-Kristian Kemell, Matti Saarikallio, Anh Nguyen-Duc, and Pekka Abrahamsson. "Still just personal assistants?–A multiple case study of generative AI adoption in software organizations." In: *Information and Software Technology* 186 (2025), p. 107805.

[17] Hatice Koc, Ali Mert Erdoğan, Yousef Barjakly, and Serhat Peker. "UML diagrams in software engineering research: a systematic literature review." In: *Proceedings*. Vol. 74. 1. MDPI. 2021, p. 13.

[18] 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*. 2006, pp. 492–501.

[19] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. "Analyzing forty years of software maintenance models." In: *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*. IEEE. 2017, pp. 146–148.

[20] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. "Mental models and software maintenance." In: *Journal of Systems and Software* 7.4 (1987), pp. 341–355. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(87)90033-1. URL: https://www.sciencedirect.com/science/article/pii/0164121287900331.

[21] Ariadi Nugroho. "Level of detail in UML models and its impact on model comprehension: A controlled experiment." In: *Information and Software Technology* 51.12 (2009), pp. 1670–1685.

[22] Norman Peitek. "A neuro-cognitive perspective of program comprehension." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018, pp. 496–499.

[23] Yunhan Qiao, Christopher Hundhausen, Summit Haque, and Md Istiak Hossain Shihab. "Comprehension-performance gap in GenAI-assisted brownfield programming: A replication and extension." In: *arXiv e-prints* (2025), arXiv–2511.

[24] Giuseppe Scanniello, Carmine Gravino, Marcela Genero, José A. Cruz-Lemus, Genoveffa Tortora, Michele Risi, and Gabriella Dodero. "Do software models based on the UML aid in source-code comprehensibility? Aggregating evidence from 12 controlled experiments." In: *Empirical Softw. Engg.* 23.5 (Oct. 2018), 2695–2733. ISSN: 1382-3256. DOI: 10.1007/s10664-017-9591-4. URL: https://doi.org/10.1007/s10664-017-9591-4.

[25] Stephanie Seupke, Sarukan Segar, and Martin Baumann. "Development of a mental model questionnaire framework: a systematic approach to measuring mental models in automated driving." In: *Transportation Research Part F: Traffic Psychology and Behaviour* 114 (2025), pp. 686–700.

[26] Md Istiak Hossain Shihab, Christopher Hundhausen, Ahsun Tariq, Summit Haque, Yunhan Qiao, and Brian Wise Mulanda. "The effects of GitHub copilot on computing students' programming effectiveness, efficiency, and processes in Brownfield coding tasks." In: *Proceedings of the 2025 ACM Conference on International Computing Education Research V. 1*. 2025, pp. 407–420.

[27] Sandhya Tarwani and Anuradha Chug. "Agile methodologies in software maintenance: A systematic review." In: *Informatica* 40.4 (2016).

[28] *UML® - Unified Modeling Language | Object Management Group*. URL: https://www.omg.org/uml/ (visited on 03/10/2026).

[29] Birgit Vogel-Heuser, Frieder Loch, Sarah Hofer, Eva-Maria Neumann, Frank Reinhold, Sarah Scheuerer, Jonas Zinn, and Kristina Reiss. "Analyzing Students' Mental Models of Technical Systems." In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. 2019, pp. 1119–1125. DOI: 10.1109/INDIN41052.2019.8972071.

[30] Lynn Westbrook. "Mental models: a theoretical overview and preliminary study." In: *Journal of Information Science* 32.6 (2006), pp. 563–579.

[31] Jeong Yang, Young Lee, Deep Gandhi, and Sruthi Ganesan Valli. "Synchronized UML diagrams for object-oriented program comprehension." In: *2017 12th International Conference on Computer Science and Education (ICCSE)*. IEEE. 2017, pp. 12–17.