# Mining reading patterns from eye-tracking data: method and demonstration

**Ioannou, Constantina; Nurdiani, Indira; Burattin, Andrea; Weber, Barbara**

*Published in:*
Software and Systems Modeling

*Link to article, DOI:*
10.1007/s10270-019-00759-4

*Publication date:*
2020

*Document Version*
Peer reviewed version

*Citation (APA):*
Ioannou, C., Nurdiani, I., Burattin, A., & Weber, B. (2020). Mining reading patterns from eye-tracking data: method and demonstration. *Software and Systems Modeling*, *19*, 345–369. https://doi.org/10.1007/s10270-019-00759-4

# Mining Reading Patterns from Eye-tracking Data

## Method and Demonstration

Constantina Ioannou[1], Indira Nurdiani[2], Andrea Burattin[1], and
Barbara Weber[1,3]

[1] DTU Compute, Software and Process Engineering,
Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark
[2] The Mærsk Mc-Kinney Møller Institute,
University of Southern Denmark, DK-5230 Odense M, Denmark
[3] University of St. Gallen, CH-9000 St.Gallen, Switzerland

**Abstract.** Understanding how developers interact with different software artifacts when performing comprehension tasks has a potential to improve developers' productivity. In this paper, we propose a method to analyze eye-tracking data using process mining to find distinct reading patterns of how developers interacted with the different artifacts. To validate our approach, we conducted an exploratory study using eye-tracking involving 11 participants. We applied our method to investigate how developers interact with different artifacts during domain and code understanding tasks. To contextualize the reading patterns and to better understand the perceived benefits and challenges participants associated with the different artifacts and their choice of reading patterns, we complemented the eye-tracking data with the data obtained from think aloud. The study used behavior driven development (BDD), a development practice that is increasingly used in Agile software development contexts, as a setting. The study shows that our method can be used to explore developers' behavior at an aggregated level and identify behavioral patterns at varying levels of granularity.

**Keywords:** process mining, eye-tracking, reading patterns, source code, behavior driven development

## 1 Introduction

One of the main concerns in the software engineering discipline is to improve developers' productivity, increase product quality, and reduce the cost of developing software [5, 45]. Different software development methods (e.g., Agile and Lean methods) [2], principles and practices (e.g., refactoring and test-driven development) [33], and tools (e.g., Integrated Development Environments or IDEs) have been put forth as means to address the aforementioned concerns [18]. In addition, in recent software engineering research, human aspects have obtained increasing attention as an important factor to improve productivity [14, 27].

Developers spend a great deal of their time understanding source code and other software artifacts, so they can add or modify functionality, fix bugs, and perform maintenance activities [31, 39]. Understanding developers' behavior during comprehension tasks with different software artifacts is important to support developers' work and as a further consequence improve their productivity [19].

In our previous work, we demonstrated the potential of using process mining to understand developers' interactions with an IDE during a programming task [18]. Using process mining we were able to identify different patterns of how developers solved programming tasks (i.e., top-down versus bottom-up approach). Our previous, however, work was limited by the fact that we could only observe behavior that required interactions with the IDE and we were not able to capture where the developers look as they interact with different software artifacts.

To overcome this limitation, we introduced eye-tracking as a tool to capture developers' eye movements when interacting with software artifacts and we propose a method to capture developers' behavior and visualize it in terms of reading patterns using process mining technology. Moreover, we complement the eye-tracking data with data from a retrospective think aloud session to contextualize the identified reading patterns.

To demonstrate the proposed method, we collected developers' eye movements while they conducted different comprehension tasks using the Eclipse plugin iTrace. As a context of the comprehension task, we chose a setting based on behavior driven development (BDD). BDD is a relatively new development practice that is increasingly used in the context of Agile software development [32, 46]. How developers use the different artifacts used in BDD as part of comprehension tasks has not been investigated up to now.

The main contribution of this paper is original regarding the following points: *(i)* it proposes a novel method to identify and visualize developer's behavior as reading patterns, *(ii)* it is one of very few works that uses eye-tracking in a way where the eye-tracking data is interlinked with the software artifacts [40], *(iii)* it demonstrates the potential of process mining to explore at different levels of granularity how developers at an aggregated level interact with software artifacts in the context of an empirical study, and *(iv)* the described empirical study is the first one investigating how software developers interact with the different software artifacts used in BDD to conduct comprehension tasks.

The remainder of this paper is structured as follows: Section 2 describes the background and related work, Section 3 explains the proposed method for mining reading patterns from eye-tracking data. Section 4 demonstrates the general method presenting an actual study with corresponding design and execution. Results of the study are reported in Section 5, whereas a broader discussion of the general method is described in Section 6. The paper is summarized and concluded in Section 7.

## 2 Background and Related Work

In this section, we discuss relevant concepts and related work. In Section 2.1 we describe process mining. In Section 2.2, we describe the existing research in code understanding. Section 2.3 describes relevant studies in software engineering using eye-tracking. Section 2.4 describes BDD and existing research in BDD.

### 2.1 Process Mining

Process mining is a research discipline bridging model-based process analysis and data oriented analysis techniques such as machine learning and data mining [48]. One of the aim of process mining is to analyze event logs, generated from executions of activities, in order to identify a process representation (i.e., a *process map* or *process model*) of the underlying process being followed.

In order to be used for process mining purposes, event logs have to conform to minimum data model requirements [17]. These require the presence of a *case id*, which determines the scope of the process; an activity, which determines the level of detail for the steps; and timestamp, which determines when each activity took place. Using process discovery, a process model explaining the behaviour of the recorded log can be inferred.

Related to the work presented in this paper is the emerging area of software process mining. With the increasing availability of software execution data, the application of process mining techniques to analyze software execution data is becoming more popular. The potential of software process mining was first demonstrated by [21, 36]. More specifically, source code repositories were analyzed to obtain insights on the software development processes that teams employed. Moreover, [1] suggests the usage of "localized" event logs where events refer to system parts to improve the quality of the discovered models. In addition, [24] proposes the discovery of flat behavioural models of software systems using the Inductive Miner [26]. In turn, [30] proposes an approach to discover a hierarchical process model for each component. An approach for discovering the software architectural model from execution data is described in [29]. Finally, [25] allows to reconstruct the most relevant state-charts and sequence diagram from an instrumented working system. The focus of all these works, however, is software development processes or the understanding of the behaviour of the software, while the focus of this paper is to define a method to identify how people (at an aggregated level) interact with different software artifacts when conducting different comprehension tasks. In the rest of this paper, process maps extracted from events referring to comprehension tasks on software artifacts will be called *reading patterns*. A reading pattern depicts how users switch between different software artifacts or parts thereof. In the literature, process mining has been used to extract reading patterns already [15]. However, in that case, the setting was very "rigid" and did not allow any interaction with the software artifacts, making the entire approach unsuitable to answer questions related to code understanding (which requires navigation over several files).

## 2.2 Code Understanding

The proposed method is suited for code comprehension tasks and thus relates to existing research on code understanding. Code understanding is one aspect of program comprehension that is vital in software development and maintenance activities [49]. Code understanding is necessary to facilitate code reuse, code inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems [39]. Code understanding has been a topic of interest in computer science as well as software engineering.

Early empirical studies focused on identifying strategies that developers use for code understanding using the think aloud approach [6, 49, 28]. The results of these empirical studies were presented as models for analyzing the cognitive processes involved in code comprehension [49]. Many of these models emphasize that developers' understanding is affected by their existing knowledge and other external documentations, like manuals or flowcharts. These models also discuss the different tactics or reading patterns used in the process of code understanding: top-down [6, 47], bottom-up [35] or both [28]. Soloway et al. [47] mention that the top-down approach is used if the code is familiar to developers. Meanwhile, Pennington [35] mentions that the bottom-up approach is used if the code is unfamiliar to developers.

One aspect of code understanding that is often studied relates to variable naming, e.g., [4, 23, 38]. Lawrie et al. [23] investigated the impact of naming variables (abbreviated versus full words) on how well participants understood the purpose of the code and could memorize it. Participants were presented with code snippets in C, C++, and Java and their ability to correctly interpreting the behavior of the snippet was used to measure their understanding. The study shows that full word variable names improve code understanding compared to abbreviated variable names.

Salviulo and Scanniello [38] performed an ethnographically-informed study to investigate the role of comments and variable names for bachelor students and professional developers in the understanding of Java source code. The data was gathered through observation of the participants when they read the source code, answer code comprehension questions, and modify the source code. The study shows students tend to spend more time on reading the comments to understand the code. Meanwhile, professional developers tend to only glance on the comments and rely on source code statements for code comprehension. The results are similar to a study conducted by Crosby and Stelovsky in 1990 [9].

Blinman and Cockburn [4] investigated how variable naming and documentation affects code understanding. The source code was written in J#. The study shows that non-descriptive variables increase study times; participants who were provided with documentation spent even more time. Furthermore, descriptive variable names increase the percentage of correct answers; participants who were provided with documentation marginally perform better than those without documentation. For participants who were provided with non-descriptive variable names, the correctness decreases significantly without documentation.

The focus of most existing work in code understanding lies on a single artifact, i.e., the source code. Very few studies have been done to investigate how developers interact with multiple artifacts to perform comprehension tasks. The method proposed in this paper, in turn, is suitable to investigate how developers interact with multiple artifacts and the paper applies the method to examine how developers, at an aggregated level, interact with different textual artifacts that are used in the context of behavior driven development (cf. Section 2.4). These artifacts include in addition to the source code, feature files describing scenarios and step definitions.

### 2.3 Use of Eye-tracking in Software Engineering

The use of eye-tracking in code understanding research has been growing, which can be seen from the number of studies included in a systematic review by Sharafi et al. [42]. One of the earlier studies in code understanding with the use of eye-tracking was done by Crosby and Stelovsky [9] to study developers' comprehension techniques using fixation-based measures like the number of fixations. The result of their study indicates that developers with low experience spend more time looking at comments, while high experienced developers spend more time on meaningful parts of the source code.

Sharafi et al. [41] investigated the impact of gender on accuracy, visual effort (measured by fixation count), and speed, given different types of identifiers, i.e. camel case versus underscore. Moreover they captured code reading strategies. The study indicates that there is no significant difference between the identifier types on accuracy of providing correct answers and visual effort. Their study supports the hypothesis that male and female subjects have different code understanding strategies. Female subjects tend to spend more effort on eliminating the wrong answer, while male subjects seem to decide more quickly on an answer.

Most tools to support eye-tracking studies can only capture the coordinates that a person is looking at without providing the contexts of what they are looking. Such a setup is not sufficient for running realistic experiments with eye-tracking in software engineering [40]. To alleviate this issue, iTrace[4] is proposed as a tool that can capture developers' eye movements as they are interacting with the IDE. It is built as a plugin to Eclipse[5] to capture more dynamic and realistic interaction between the participants and the IDE, e.g., allow participants to scroll through the code and switch between tabs or windows on the IDE. iTrace captures the eye gazes and links them to specific source code elements [40].

Walters et al. [52], used iTrace in a study to capture links between a textual bug description and source code when developers performed bug fixing tasks. The study was done as a proof-of-concept that gaze data could be used to retrieve traceability links between software artifacts.

Kevic et al. [19] used iTrace to monitor developers' eye gazes during change tasks. Their study shows that it is possible to capture both gaze and interaction

[4] See http://www.i-trace.org/.

[5] See http://www.eclipse.org/.

data. The insights gained from the gaze and interaction data could be used to determine better what developers are interested in which can be used to provide recommendations to developers on other relevant artifacts.

Most eye-tracking studies in code understanding focused on comparing two different treatments and used quantitative measures, which can be seen from the primary studies included in [42]. Only a few studies used eye-tracking to investigate how developers interact with multiple artifacts, e.g., in [52] and [19]. However, these studies are not focused on understanding developers' reading patterns. Moreover, unlike most existing work that investigated code comprehension using eye-tracking our main interest is not so much on visual effort (i.e., fixation-based measures), but rather on the reading behavior that developers employed.

### 2.4 Behavior Driven Development (BDD)

Behavior driven development (BDD) is a group of techniques that are aimed to help software development teams to identify and deliver the most valuable features for the business [44]. BDD is often perceived to be an evolution of test driven development (TDD). It was proposed by Dan North to alleviate difficulties with TDD, such as confusion and misunderstanding on how to test and what to test [32].

BDD is usually used in Agile and Lean software development [44], and it involves steps described in Figure 1. The starting point for BDD is the formulation of business goals (Step 1). In BDD, different roles like developers, testers, and business analysts work together with the customers to identify and formulate features that fulfill the business goals (Step 2). A feature is "a piece of software functionality that helps users or other stakeholders achieve some business goal" [44]. A feature is expressed in the form of scenarios. A scenario describes how the system would behave given certain preconditions and other events, like user inputs, or interaction from another system (Step 3). To describe a system behavior, each scenario uses a Given-When-Then structure. The *Given* statement describes the initial context or pre-condition, the *When* statement describes an event or an operation undertaken in the system, and the *Then* statement describes the expected outcome. This feature specification is formulated in domain specific language (i.e., Gherkin) that can be easily understood by both the developers and business analysts [3, 46]. To obtain an executable specification, in the next step, step definitions are created (Step 4) providing the translation between the scenarios and the application code. Lastly, the software under test with detailed implementation of application code is developed (Step 5). Usually, this process is supported by tools like like Cucumber, JBehave, or RSpec. Figure 1 depicts the BDD process.

In Figure 2, we provide a concrete example of a feature with one scenario (described in Gherkin), the corresponding step definition, as well as the application code in Java. Figure 2 presents a feature to reload or add a balance to a travel card at an automated kiosk. One of the scenarios is for successful travel card

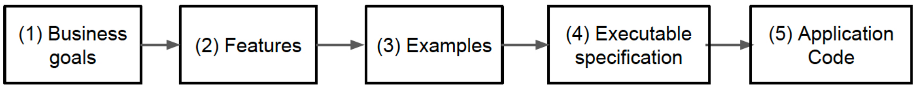
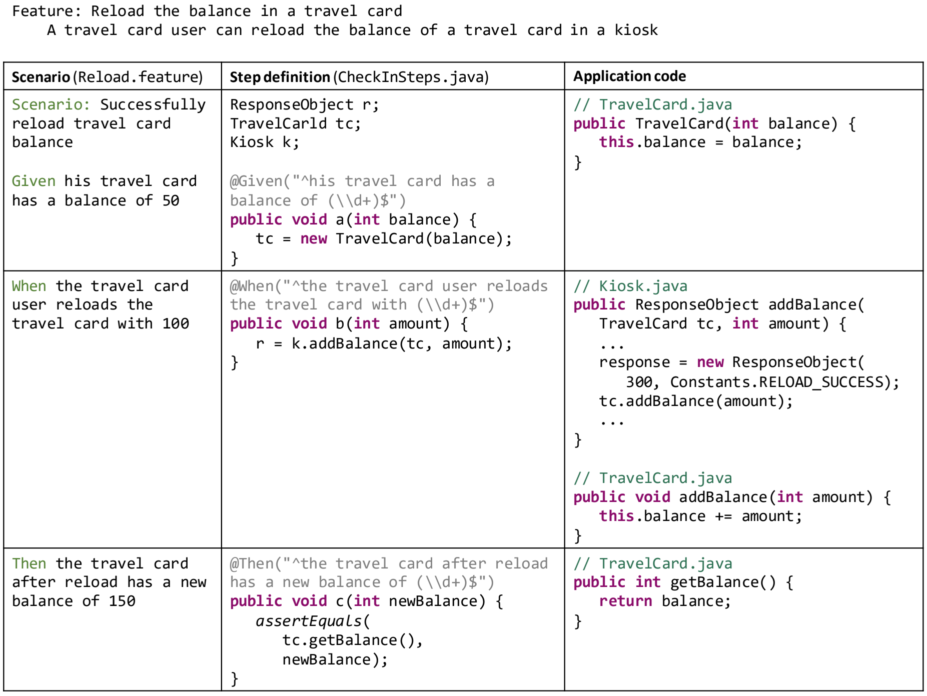


Fig. 1: BDD process adapted from [44]

reload. A concrete example is used to describe the statements or steps in the scenario, such as an initial balance of 50 in the travel card is provided (*Given*), an amount of 100 is added to the travel card (*When*), resulting in a final balance of 150 after a successful reload (*Then*). For each statement or step in the scenario, there is a corresponding method in the step definition. In the step definition, there are different methods; each method calls to the method in the application under test that implements this statement.



```
Feature: Reload the balance in a travel card
    A travel card user can reload the balance of a travel card in a kiosk
```

| **Scenario** (Reload.feature) | **Step definition** (CheckInSteps.java) | **Application code** |
|---|---|---|
| Scenario: Successfully reload travel card balance<br><br>Given his travel card has a balance of 50 | ResponseObject r;<br>TravelCarld tc;<br>Kiosk k;<br><br>@Given("^his travel card has a balance of (\\d+)$")<br>public void a(int balance) {<br>   tc = new TravelCard(balance);<br>} | // TravelCard.java<br>public TravelCard(int balance) {<br>   this.balance = balance;<br>} |
| When the travel card user reloads the travel card with 100 | @When("^the travel card user reloads the travel card with (\\d+)$")<br>public void b(int amount) {<br>   r = k.addBalance(tc, amount);<br>} | // Kiosk.java<br>public ResponseObject addBalance(<br>   TravelCard tc, int amount) {<br>   ...<br>   response = new ResponseObject(<br>      300, Constants.RELOAD_SUCCESS);<br>   tc.addBalance(amount);<br>   ...<br>}<br><br>// TravelCard.java<br>public void addBalance(int amount) {<br>   this.balance += amount;<br>} |
| Then the travel card after reload has a new balance of 150 | @Then("^the travel card after reload has a new balance of (\\d+)$")<br>public void c(int newBalance) {<br>   *assertEquals*(<br>      tc.getBalance(),<br>      newBalance);<br>} | // TravelCard.java<br>public int getBalance() {<br>   return balance;<br>} |

Fig. 2: Example of scenario, step definition, and application source code

Existing literature has identified several benefits of BDD as well as challenges. Benefits associated with BDD, particularly due to the use of plain text description in a scenario include improved traceability between requirement specification and code [22], improved understandability of the requirement specification,

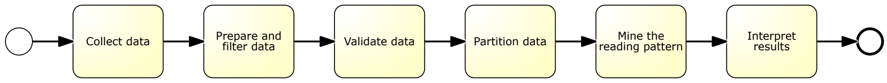


Fig. 3: Method to mine reading patterns from eye-tracking data.

improved understandability of the code intention [3], and reduced misinterpretation of the requirements specification [10]. Limitations associated with BDD include a steep learning curve, decreased team productivity, and challenges pertaining to a new way of working [3].

Our review of BDD literature shows that very few studies systematically evaluate the use of BDD artifacts in a controlled environment like an experiment. One recent study that evaluated the use of BDD artifacts was done by Wang and Wagner [53]. They performed an experiment to compare the use of BDD versus traditional User Acceptance Testing (UAT) for safety verification with respect to productivity, thoroughness, effectiveness in fault detection, and communication effectiveness. Their study shows that BDD is more effective than UAT regarding communication effectiveness. Meanwhile, BDD shows no significant difference in productivity, thoroughness, and effectiveness in fault detection.

Our study is the first empirical study that uses eye-tracking in a BDD context. We selected BDD as context because as a software development approach BDD is gaining interest and increasing in its adoption [50]. Furthermore, from the perspective of adding realism in studying reading and navigation patterns, BDD involves different types of software artifacts, which reflects what software developers in the industry are more likely to face on a daily basis, as exemplified in [52]. Currently, little is known on how BDD impacts developers' behavior when performing comprehension tasks. In this study, we define developers' behavior as their interactions with different textual artifacts like the feature file, step definition, and source code. Moreover, we aim to explore if the benefits and limitations of BDD are reflected in the developers' behavior when performing comprehension tasks.

## 3 Mining Reading Patterns from Eye-tracking Data

This section introduces a novel method based on eye-tracking in order to extract reading patterns on how developers interact with different artifacts while conducting comprehension tasks. The method was developed following the design science framework described in [54]. The general method is graphically reported in Fig. 3 and in the remainder of this section details about each step are presented and discussed.

*Collect Data.* The first activity of the approach concerns the collection of eye-tracking data, to be processed for process mining purposes. Eye gazes and fixations of subjects are captured and linked to the software artifacts being shown to

Table 1: Structure of the data collected by iTrace, after some preparation

| **Subject ID** | **Artifact** | **Timestamp (Start)** |
|---|---|---|
| SBJ1 | `Task 1.1.0.md` | 2018-09-04T13:30:03.013 |
| ⋮ | ⋮ | ⋮ |
| SBJ1 | `Kiosk.java` | 2018-09-04T13:36:05.015 |
| SBJ1 | `Kiosk.java` | 2018-09-04T13:36:05.161 |
| SBJ1 | `TravelCard.java` | 2018-09-04T13:36:17.192 |
| ⋮ | ⋮ | ⋮ |
| SBJ1 | `Task 1.1.1.md` | 2018-09-04T13:38:02.015 |
| SBJ1 | `Task 1.2.0.md` | 2018-09-04T13:38:58.043 |

the subjects. These data enable the identification of the areas of the screen being read by the subjects while answering the understanding questions. These areas, in turn, are referring to different resources being displayed. Table 1 reports an example of such data: the first column refers to the subject under examination, the second column reports the artifact being shown, and the third column reports the time stamp of when the subject started their reading. In addition, this step is also responsible for obtaining supplementary data to help the contextualization and interpretation of the results later on. Such data can be automatically collected (e.g., which task the subject is currently performing) or might require additional effort (e.g., think aloud or surveys).

*Prepare and Filter Data.* Once the data is collected for all participant, it is necessary to pre-process it in order to integrate all information needed for process mining analysis. To enable such type of analysis it is important to define 3 concepts. Specifically, an *activity* represents a unit of work that can be tracked. Each activity has to belong to a single *process instance* and process instances must refer to a certain *process*. Then, by using process mining, it is possible to synthesize a process model describing the behavior shared by the majority of the instances that are executing the same process. In the context of this paper, activities represent comprehension tasks on software artifacts, and the extracted processes are therefore reading patterns. Specifically, the activities we considered comprise reading different available artifacts (i.e., either a specific artifact or an artifact *type*) to solve the comprehension task.

A graphical representation of the way we identified activities and process instances is reported in Fig. 4. In our method, each subject needs to read a "question file" first, then navigate to and explore different artifacts available and finally, once they know the answer, they are asked to open an "answer file" before proceeding to the following question. Since fixations on all artifacts were recorded, we used the first fixation on a question file and the first fixation on the corresponding answer file as time stamps indicating the beginning and the end of a process instances (referring to the process of answering corresponding question type). An example of such process instance is depicted by the red line

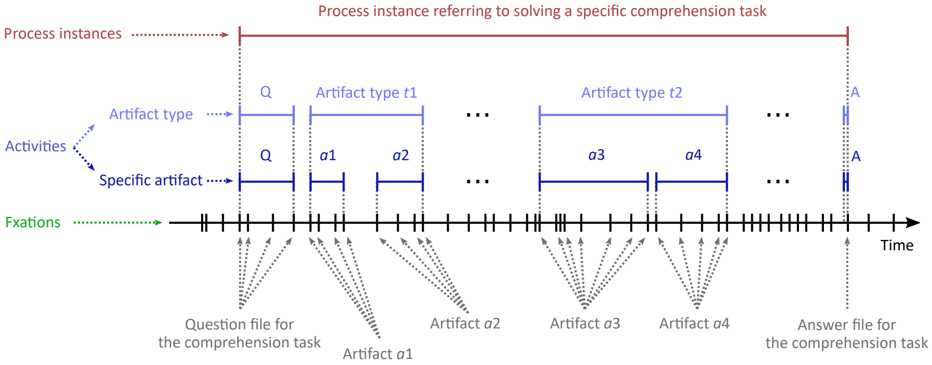


Fig. 4: Graphical representation of the sequence of fixations over time (including the artifact they refer to) and their aggregation to a process instance and to activities (whereby activities can be artifacts or artifact types).

on top of Fig. 4: it starts when the first fixation (black line, at the bottom) on a question file is detected and ends when the first fixation on the corresponding answer file is detected.

Once process instances are isolated, corresponding activities can be identified. All activities are referring to the focusing on (i.e., reading of) certain artifacts. Therefore, it is possible to group together all contiguous fixations referring to the same artifact. In principle, it is possible to group artifacts at different abstraction levels that, in turn, entails different definitions of *activities*: considering the example of Fig. 4, it would be specific artifacts (e.g., $a1$, $a2$) or artifact types (e.g., $t1$, $t2$).

*Validate Data.* In order to have reliable results, it is important to remove from the dataset used for the analysis all recordings referring to subjects (or tasks) for which some irregularities were observed during the experiment (e.g., the subject was interrupted or the instrument did not function properly).

*Partition Data.* Once the data has been prepared, it is possible to conduct the actual analysis according to two strategies. On the one hand, one might extract a reading map that encompasses all behavior in an aggregated fashion. The other possibility is to split the data according to some criteria (i.e., subject properties, task properties or answer properties) and then compare the extracted maps. This latter approach is especially useful in case of comparative analyses aiming at validating research hypotheses.

*Mine the Reading Patterns.* Once the data is properly partitioned (either resulting in one or several partitions), it is possible to mine the process models focusing on one abstraction level at the time (either specific artifacts or artifact types). For the actual mining, our method proposes to use the tool Disco.[6]. We

[6] See `http://fluxicon.com/disco/`.

recommend keeping the configuration of the tool to preserve all behavior (i.e., 100% for both "activities" and "paths"), in order to not loose any information.

*Interpret Results.* As with any knowledge discovery process [8], the results of the mining, i.e., the reading patterns, need to be interpreted in order to synthesize knowledge and therefore be useful. To help this phase, it is necessary to exploit the additional data collected during the experiment (e.g., the task that each pattern is associated with, or think aloud transcripts, or surveys), as described in the very first step.

## 4 Method Demonstration

The previous section introduced a method to identify reading patterns while conducting comprehension tasks based on eye-tracking data. In this section we demonstrate this method by applying it in an exploratory study with BDD as setting. This section describes the study that we conducted to understand developers' behavior. Section 4.1 describes the study design and execution. Section 4.2 describes how we implemented and adapted the method proposed in Section 3 to our needs.

### 4.1 Study Design and Execution

The main aim of this study is to investigate how developers engage with different software artifacts (feature, step code, source code), in terms of reading patterns, to perform different comprehension tasks (i.e., *domain understanding* and *code understanding* tasks which will be explained in paragraph **Task Description**) respectively. The study, additionally, aims to identify what benefits and challenges are associated with each artifact during different comprehension tasks to be able to better contextualize the identified reading patterns. To achieve the aim, the following research questions are formulated:

- **RQ1.1**. How do developers engage with the different artifacts (feature file, step code, source code) in terms of reading patterns to complete domain understanding tasks?
- **RQ1.2**. What were the benefits and challenges associated with the different artifacts during domain understanding tasks (i.e., how should we interpret the reading patters extracted in RQ1.1)?
- **RQ2.1**. How do developers engage with the different artifacts (feature file, step code, source code) in terms of reading patterns to complete code understanding tasks?
- **RQ2.2**. What were the benefits and challenges associated with the different artifacts during code understanding tasks (i.e., how should we interpret the reading patters extracted in RQ2.1)?

**Participants.** We used convenience sampling to recruit the participants. In recruiting the participants, we did not screen them based on their knowledge in Java programming or BDD. We only screened participants based on their vision condition. We had 11 participants who joined the study. The participants were recruited from the Technical University of Denmark (DTU) and the University of Cyprus (UCY), six and five participants respectively. All subjects had an academic background in Computer Science or related engineering field and therefore they can be seen as proxies for junior developers.

**Task Description.** In this study, we asked the participants to complete different comprehension tasks concerning the Rejsekort system. Rejsekort is a travel card system that is used in Denmark. Using a travel card, a passenger can pay for travel fares using a travel card instead of cash by checking in upon embarking and checking out upon disembarking a bus or a train.

Overall, the study consisted of four tasks. For each task, participants had to answer two questions of different type (cf. Figure 5):

1. *Domain understanding*, where we asked for high-level requirements of specific functionality. Such type of question can be answered by just reading the feature file. Alternatively, when not using the feature file the answer has to be inferred from the source code. Appendix A provides an example of a domain understanding question.
2. *Code understanding*, where the question asks how a specific functionality is implemented in the source code. The answer to such question can be found in the source code. BDD artifacts can be used to navigate to the relevant part in the source code, potentially facilitating the search of the source code that is relevant. Appendix A shows an example of a code understanding question.

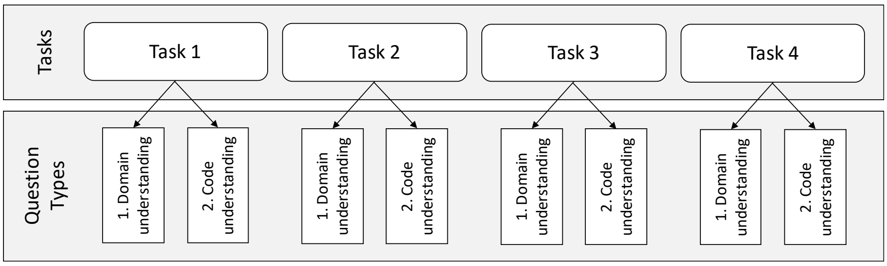


Fig. 5: Tasks structure

In addition to the questions, we also developed the artifacts and source code for the warm-up task, i.e., a simple calculator program, and the main tasks, i.e., the Rejsekort system.

In developing and formulating the artifacts for the Rejsekort tasks, we used our knowledge of using the travel card system and picked the most representative functionalities that we are familiar with. In developing the artifacts, we followed BDD principles: we started by writing the feature files in Gherkin, followed by the step definitions (to provide the glue between the feature file and the source code) and the source code in Java. The software artifacts used in the study are available at `https://github.com/CIoann/cucitrace`.

To ensure correctness of the artifacts, we also performed code inspection which was done by individuals who were not involved in developing the code. In addition, we also piloted the study with one individual who was not included as the study participants.

**Study Procedure.** We conducted the study in two locations at two universities; DTU in Denmark and UCY in Cyprus. In each location, we used a designated room to minimize interruptions.

*Training.* When a participant arrived at the designated room, we provided him/her with an introduction of the study procedure, brief introduction to the Rejsekort system, and how to interact with iTrace (e.g., calibrate the eye tracker, start and stop eye tracker). We also provided the participants with a brief introduction to BDD, its artifacts, and how they are related. At this point, we encouraged the participants to ask questions to make sure they know the tasks at hand.

*Pre-Experiment Questionnaire.* After the introduction and signing the consent form, we asked the participants to fill in the pre-experiment questionnaire to inquire about their knowledge and experience in programming and BDD. The detailed questions can be found in the external appendix at `https://zenodo.org/record/2602995`.

*Tasks.* After completing the pre-experiment questionnaire, we directed the participants to start with the main tasks. We asked the participants to sit in front of the laptop screen in a comfortable and upright posture. First, we asked the participants to perform the calibration to ensure that the gaze data will be collected properly [11]. Once calibration was completed, we asked the participants to start the eye-tracking. Then the participants proceeded with the warm-up task, followed by the main task, i.e., four tasks related to the Rejsekort system. In the end of each task, we asked the participants to perform a retrospective think aloud and walk us through the steps that they took to complete the task. This was done at the end of all four tasks. We chose retrospective think aloud because it allowed the participants to focus on the task at hand. Concurrent think aloud might influence the participants' behavior and reduce mental capacity because they divide their attention between the task and explaining what they are doing [7]. Participant had no time limit for solving their task. We also allowed the participants to ask for clarifications throughout the experiment. Once the participants completed all tasks, we asked them to stop the eye-tracking.

*Post-experiment questionnaire.* After a participant completed the tasks, we asked them for follow up questions to reflect on their experience in completing the tasks. We also asked the participants concerning the extent to which the different software artifacts helped them to answer domain understanding and code understanding tasks (using a five-point Likert scale). We also instructed the participants to indicate *Strongly disagree*, if they did not use a specific artifact to answer the domain understanding or code understanding task. The post-experiment questionnaire was also recorded and then later transcribed. Details of the questionnaire can be found in the external appendix at `https://zenodo.org/record/2602995`.

**Instrumentation.**

*Tools and settings.* The Rejsekort system was provided to the participants as an Eclipse project (including features files, step definitions, and the source code). To prevent the participants from taking their eyes away from the screen when reading the questions and answering them, we added them as files to the Eclipse IDE. As shown in Figure 5, in each task, there are two question types, i.e., domain understanding and code understanding. For each question type, there are two text files which the participants had to read. The first file contains a specific question that the participants had to answer and an instruction to open the second file once they found the answer to the question. The second file contains an instruction to the participants to say out loud their answers and how they found the answers.

For collecting eye movement data, we used a Tobii 4C[7] eye tracker with a sampling rate of 90 Hz. The device does not require the participants to wear any gear. The eye tracker is stationary at the bottom of the laptop's screen. The laptop used for the study is an HP Elite book with a 15.6-inch screen and a resolution of 1920x1080 pixels. The font of the text was increased to 14 point size to be visible from a distance of 60cm.

*Extending iTrace.* In order to map the eye-tracking data with the software artifacts, we used the iTrace plugin of Eclipse, which we had to extend to record all needed information[8]. The iTrace plugin is able to track eye gazes and fixations on source code entities. Specifically, the entities are extracted from the generated Abstract Syntax Tree (AST) of source code files. The extensions of iTrace that we needed to implement concern the mapping of eye-tracking data to feature files, which are written using the Gherkin language (cf. Section 2.4).

[7] See `https://tobiigaming.com/product/tobii-eye-tracker-4c/`.

[8] The new version of the iTrace plugin is availalable on GitHub, at `https://github.com/CIoann/cucitraceSetup`.

### 4.2 Implementation of the Proposed Method

In order to answer the research questions, we employed the method presented in Section 3. In the following we report the details concerning the actual realization of the individual activities previously described

*Collect Data.* To collect data concerning a developer's reading interactions within the IDE we used a modified version of the iTrace plugin for the Eclipse IDE as reported in Section 4.1. During the task, eye gazes and fixations of subjects were captured and linked to source code entities. The data of all 13 participants has been stored in a database, which contains a table for gazes and fixations on files. These data enabled the identification of the areas of the screen being read by the subjects while answering the understanding questions. These areas, in turn, are referring to different resources being displayed by the IDE.

*Prepare and Filter Data.* To prepare the raw data to be analyzed with process mining techniques it is important to identify activities, process instances and processes. In the context of this paper, we defined one process for each of the two question types that the subjects were asked to answer (i.e., domain understanding and code understanding, cf. Fig. 5). Therefore, each subject executed 4 instances of each process (i.e., 4 tasks, each with 2 questions, cf. Fig. 5). The activities we considered comprise reading the different resources available (i.e., either a specific artifact or file depending on the abstraction level) to answer each question.

As described in Fig. 4 to identify activities and process instances we first isolated all process instances by considering the first fixation on a question file and the first fixation on the corresponding answer file. For this study, we decided to analyze two possible abstraction levels (i.e., two definitions of *activities*): the specific *file* and the *artifact type* that a file is referring to. When abstracting at file level, all contiguous fixations on the same file are grouped together into an activity which consists of reading that specific file. As a consequence, each process can potentially have as many activities as files available (yet, only a subset of these reading activities is typically performed). Examples of such reading activities are `CheckInAutomaton.java` and `CheckIn.feature`. When focusing on artifact types, contiguous fixations that refer to files of the same type are grouped together. The possible types of artifacts in our project are: feature files, source code, and step code. Therefore, for example, all contiguous fixations on source code files (even when different files are involved) are grouped together.

*Validate Data.* During data preparation, we noticed that the data of two participants had to be excluded. One participant had to be excluded since there were prolonged periods during which the participant looked away from the screen resulting in invalid eye-tracking data. Another participant had to be excluded due to problems with the data recording. Therefore, we included 11 participants in our analysis. Additionally, we had to exclude data of Task 4 from the study since it turned out after data collection that the data for some of the participants was corrupted. Thus, in the following only data related to Task1-3 is analyzed.

Table 2: Result of data partitioning along the two dimensions considered: question type (i.e., domain/code understanding) and the combination of resources used to provide an answer. Each cell reports the subject IDs that belong to the corresponding partition of the data. Combinations never used by any subject are indicated with a dash ('-').

| | **Domain understanding** | **Code understanding** |
|---|---|---|
| **Feature** | DTU3, DTU7, DTU8, DTU9, DTU10, UCY1, UCY3, UCY4 | - |
| **Step Code** | - | - |
| **Source Code** | DTU6, UCY2 | DTU6, DTU7, DTU9, UCY1, UCY2, UCY4, UCY5 |
| **Feature + Step Code** | DTU3 | - |
| **Feature + Source Code** | - | DTU3, DTU9, UCY1, UCY3 |
| **Step Code + Source Code** | - | DTU3, DTU10 |
| **Feature + Step Code + Source Code** | DTU8, DTU3 | DTU3 ,DTU8, DTU10 |

*Partition Data.* In order to investigate the collected data, we partitioned the subjects according to the artifacts they focused on while answering the two questions types (i.e., questions about domain understanding and code understanding). Specifically, we considered all possible partitions emerging from the combinations of the three artifact types (which are feature files, source code and step code) and we grouped subjects that used the same set of artifact types to answer the questions. Orthogonally, we divided the process instances into two categories: the process being followed to answer domain understanding questions (independently from the task number) and the process of answering code understanding questions (independently from the task number).

Table 2 reports the division of the subjects among the categories used for partitioning the data. Each cell of the table reports the IDs of the subjects who answered the given question using the corresponding subset of resources. For domain understanding questions, subjects used either:

- only the feature files (eight subjects);
- only the source code (two subjects);
- the feature files and the step code (one subject);
- the feature files, the step code and the source code (two subjects).

To answer code understanding questions, subjects used either:

- only the source code (seven subjects);
- the feature files and the source code (four subjects);
- the step code and the source code (two subjects);
- the feature files, the step code and the source code (three subjects).

Please note that in the data we used each subject answered three domain understanding questions and three code understanding questions. In addition, some subjects used different subsets of resources to answer questions of the same type. This is why some subjects appear more than once in Table 2 (column-wise). In addition, one subject (UCY5) answered all domain understanding questions just by guessing, therefore they do not appear in the corresponding columns.

*Mine the Reading Patterns.* Each set of process instances generated from the partitioning devised in Table 2 has been used to mine process models considering both file and artifact abstractions as activities. Only combinations actually used by subjects have been mined and therefore, in total, we extracted 16 maps:

- Eight maps for domain understanding questions: two maps for each of the four sets of used resources, one with file abstraction and another with artifact abstraction (cf. Fig. 4);
- Eight maps for code understanding questions: two maps for each of the four sets of used resources, one with file abstraction and another with artifact abstraction (cf. Fig. 4).

For the actual mining, we used the tool Disco configured to preserve all behavior (i.e., 100% for both "activities" and "paths").

*Interpret Results* To interpret the reading patterns obtained, we employed qualitative coding on the data from the post-experiment questionnaire concerning: *1.* the participants' perception concerning the usefulness of the different software artifacts for domain and code understanding; and *2.* challenges in completing domain and code understanding tasks.

To help with the coding process, we used Atlast.ti qualitative analysis software[9]. We employed open and focused coding as described in [37]. More precisely, for open coding, we analyzed the transcriptions of the interview sessions and employed line-by-line coding. We added labels on each line of the transcripts. However, if one line does not yield to useful information, we labelled a text fragment, i.e. a sentence or more. For example, for the following text fragment: "I think it was easy to find things because of the naming, everything was really to help me understanding the code", was assigned with the labels *source code benefit* and *clear naming in the source code.*

We then followed up the results from the open coding with focused coding. We identified common themes from the data labels that we added in the open coding. The results of the focused coding can be seen in Tables 3-6.

To ensure consistency and minimize bias in the coding process, a post-hoc validation was performed. Two of the co-authors were involved in the open coding process of the post-experiment interview transcripts. One person performed the open coding process and another person conducted a post-hoc analysis to ensure consistency. Disagreements were discussed and adjustments were made on the codes applied on the text. On the focusing coding step, one person was responsible for the categorization. The result of the categorization was then

[9] See `https://atlasti.com/`.

examined by another (independent) co-author who was not involved the open coding process to minimize bias.

## 5 Results of Method Demonstration

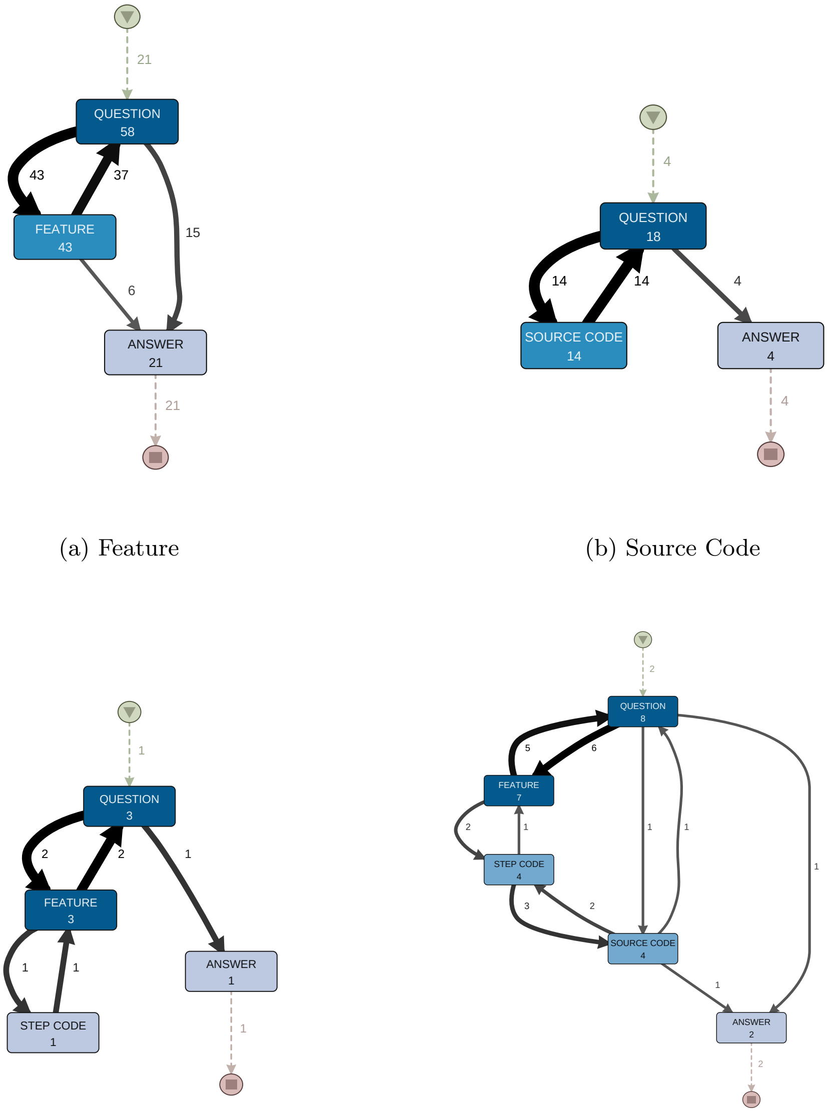
In this section we report the results of the study we conducted to demonstrate the method, as reported in Section 4. Specifically, Section 5.1 and 5.2 present the results for domain understanding tasks, while Section 5.3 and 5.4 focuses on code understanding tasks.

### 5.1 Engagement with Software Artifacts During Domain Understanding (RQ1.1)

In this section we report the data investigation needed to answer RQ1.1: we aim at understanding how developers engaged with the different artifacts in order to solve domain understanding tasks. For this, we show process maps depicting the interaction behavior at two abstraction levels: artifact types and files. For these maps, we aggregated domain understanding questions of all tasks. In five cases, however, subjects did not use any resource (w.r.t. those reported in Table 2): they decided to just guess their answers. For this reason, we ended up with a total of 28 process instances (11 subjects, each answering three domain understanding questions except in five cases; i.e., 11 * 3 - 5 = 28) among 4 categories (i.e., Feature, Source Code, Feature + Step Code, Feature + Step Code + Source Code). Please note that all raw data as well as process maps at both levels of abstraction are available online at `https://zenodo.org/record/2602995`.

**Artifact type abstraction.** First, we analyze the behavior when the artifact type abstraction is employed (cf. Fig. 4). Maps mined for the four partitioning of the subjects are reported in Fig. 6. As previously mentioned, each map shows the different reading patterns used by the subjects where each node is a fixation period (i.e., a reading activity) on the corresponding artifact. Each edge indicates direct succession in the artifacts being fixated. In addition, both nodes and edges are color-coded with their absolute frequencies (i.e., how often fixations/direct successions on artifacts were observed).

Maps depicted in Fig. 6a, 6b, 6c and 6d show, respectively, the processes followed when only feature files were investigated, when only the source code was used, when feature files and step code was read, and when all resources types were used. As expected by the study design and the data analysis technique (cf. Section 4.2), all maps always start with fixations on questions and always terminate with fixation on answers. It is relevant to note that, when subjects had been using either just the feature files (Fig. 6a) or just the source code (Fig. 6b) or on feature and step code (Fig. 6c), their reading patterns were compact, as opposed to the case when subjects used feature, step code and source code. In this latter case (Fig. 6d) the pattern is less structured, yet the thickest connections show that subjects started their reading from features files, continued with the step code and finally they went to the source code.



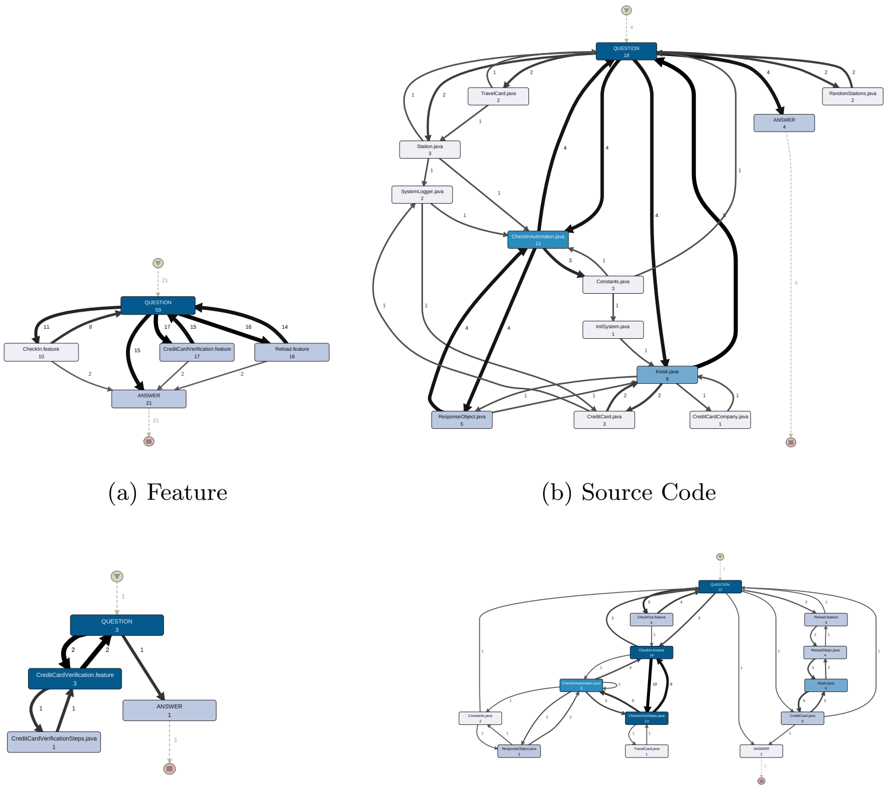
(a) Feature

(b) Source Code

(c) Feature + Step Code

(d) Feature + Step Code + Source Code

Fig. 6: Process maps mined for domain understanding questions considering the artifact type abstraction for activities (cf. Fig. 4). Larger figures are available at `https://zenodo.org/record/2602995`.



(a) Feature

(b) Source Code

(c) Feature + Step Code

(d) Feature + Step Code + Source Code

Fig. 7: Process maps mined for domain understanding questions considering the file abstraction for activities (cf. Fig. 4). Larger figures are available at `https://zenodo.org/record/2602995`.

**File abstraction.** After the high-level overview obtained with the artifact abstraction, we analyze the same process instances but, this time, with focus on file abstraction. This means that we grouped fixations referring to the same file being read (cf. "Specific artifact" in Fig. 4). Maps depicted in Fig. 7a, 7b, 7c, and 7d show, respectively, the processes followed when only feature files were investigated, when only the source code was used, when feature files and step code was read, and when all resources types were used. The semantics of the maps is the same as for the previous abstraction.

It is relevant to note that the processes followed by subjects who only used feature files (Fig. 7a) or feature files and step code (Fig. 7c) had a very compact reading patters also at this abstraction level. Instead, for subjects who used the source code, Fig. 7b unveils the actual complexity (that was not perceivable in Fig. 6b): they navigated many Java files and, as the homogeneous thickness of edges suggests, there was no clear sequence being followed (i.e., all events equally likely to happen one after the other). Finally, subjects who used feature, step code, and source code (i.e., Fig. 7d) had to navigate several resources but it is possible to note that most of the time the first files being inspected (i.e., the files directly exiting question) were feature files, followed by step code, and then the actual java files. This observation suggests that these subjects were more guided towards the identification of the Java sources to inspect.

In conclusion, we answer RQ1.1 by observing that it has been possible to group subjects into four categories. Each group showed different interaction patterns when reading the resources. In addition, the two abstractions of activities allowed to "drive" our investigations by starting from a general view (i.e., artifact abstraction) and then dig down into the more fine grained definition of the patterns (i.e., file abstraction). It is important to note that these questions (i.e., domain understanding) could be answered by just reading the feature files, which happened in the majority of the cases. For three questions, however, two subjects (i.e., DTU8 and DTU3) decided to also analyze the step code and, in two cases, the source code as well in order to "validate" their answer, as will be explained in Section 5.2.

### 5.2 Benefits and Challenges of Software Artifacts for Domain Understanding (RQ1.2)

To improve the interpretation of the results presented using the reading patterns, we analyzed the benefits and the challenges associated to the usage of different artifacts. These results are complemented with data from our retrospective think aloud session concerning the perceived benefits and challenges of the different software artifacts used in answering the domain understanding tasks.

**Feature File.** Figure 8 depicts the extent to which participants perceive the feature file useful to accomplish the domain understanding tasks respectively. The majority of the participants (8 out of 11) either strongly agree or agree on the usefulness of a feature file for answering domain related questions. This is also reflected in the choice of reading pattern (i.e., the overwhelming majority of subjects answered domain understanding questions by only relying on the feature file). Only DTU6, UCY2, and UCY5 abstained from using the feature file.

Overall, the participants agree that the feature helped them answer the *domain understanding* tasks more easily and quickly. Table 3 shows in more detail how the feature file helped them to answer domain understanding and code understanding tasks respectively. We can see from Table 3, a number of participants found a feature file useful because it provides an overview of what the system does. The concrete example outlined in a scenario in a feature file also helps the

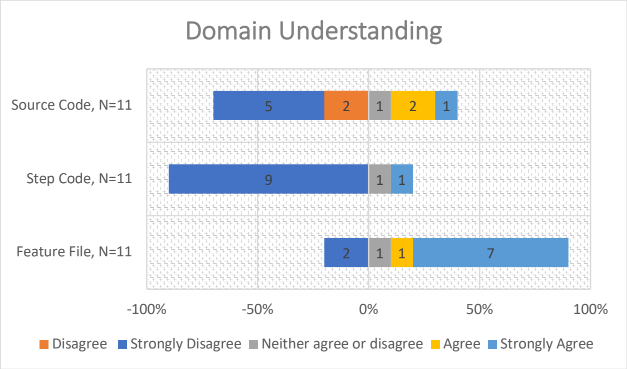


Fig. 8: Extent of usefulness of artifact types (i.e., feature file, step code, source code) for domain understanding tasks. N represents the total number of subjects. The numbers in the bars indicate the number of subjects. Strongly disagree indicates that the feature file was not used.

Table 3: Benefits of a feature file for domain understanding.

| | **Benefits of a feature file** | **Participants Quotes** |
|---|---|---|
| Domain Underst. | Provide outline of requirements and verification conditions | "*So basically the verification checks are in there [...]. it clearly stated the conditions for when a card was not valid. It had tests that this card is not valid under these conditions*" (DTU3)<br>"*It has all the scenario that I was looking for, it has something about having three requirements and there are three scenarios so I went through that*" (DTU10) |
| | Provide insight on the structure and intention of the code | "*Help [to understand] the flow of the code how it would go*" (UCY3)<br>"*It tells you how the code behind it is structured*" (DTU8)<br>"*Its important to have feature file to guide you through the source code*" (UCY4). |
| | Easy to understand due to the use of natural language | "*This file is just written in natural language, so I could understand the file and what is written in there*" (DTU9)<br>"*It was in natural language, you can understand it more easily than reading source code*" (UCY4) |
| | Easy to understand due to the Given-When-Then structure | "*Then I could easily answer, especially that I know like the structure, so we have given and then, so precondition should be in given statement*" (DTU9)<br>"*It was very useful to look up in the feature file to find the answer, I understand that it was usually the first lines is the [pre-]conditions*" (UCY4). |

participants to get an insight of what the code does and what it is expected to do without the effort of reading the code. Moreover, participants outlined that the formulation of the scenario in natural language helped them. In addition, two participants highlighted that the Given-When-Then structure of a scenario helped them to locate relevant information.

Despite the high adoption of feature files for domain understanding tasks participants were challenged by the steep learning curve, since most of the participants used BDD for the first time during the study. Remembering where the different files are located and navigating the different packages turned out to be challenging for first time users: "*[It was challenging] to find the correct feature file and the correct scenario*" (UCY3); "*I did not understand how it worked in the beginning. Then I remember that the feature file exists it was easier*" (UCY4).

**Step definition.** Figure 8 depicts the extent to which the participants perceive the step definition useful to answer the domain understanding tasks. Most participants indicate that the step definition is not useful for domain understanding. This is in line with the reading patterns, which show that only two participants used the step code, i.e., DTU3 and DTU8. These results are not very surprising, since the answer for domain questions can be found in the feature file and thus there is no need to check the step definition (given the assumption that the feature file is complete). In turn, one participant questioned the completeness of the scenario in the feature file, which led him to inspect all relevant artifacts including step code even after he looked at the correct scenario in a feature file: "*I was doubting if there was any other checks that might have been missing in the feature file*" (DTU8).

**Source Code.** Figure 8 depicts the extent of participants' agreement concerning the usefulness of the code for domain understanding tasks. We can see from Figure 8 that only three out of 11 participants found the source code useful (strongly agree and agree) for domain understanding. This is in line with the reading patterns where only four participants used the source code (two of them in combination with the step code), i.e., DTU3, DTU6, DTU8, and UCY2. This can be explained by the fact that the answer could be easily found in the feature file. The lacking familiarity with BDD could be a potential explanation of why a few participants relied on the source code rather than the feature files: "*I was not able to distinguish what are preconditions or like examples in the scenario. For some questions, I went in to the source code to have a clear idea*" (DTU9). Another participant that used the source code for domain understanding tasks mentioned that the comments in the source code were helpful for domain understanding tasks: "*The source code had like small explanation in the comments which was useful*" (DTU6).

### 5.3 Engagement with Software Artifacts During Code Understanding (RQ2.1)

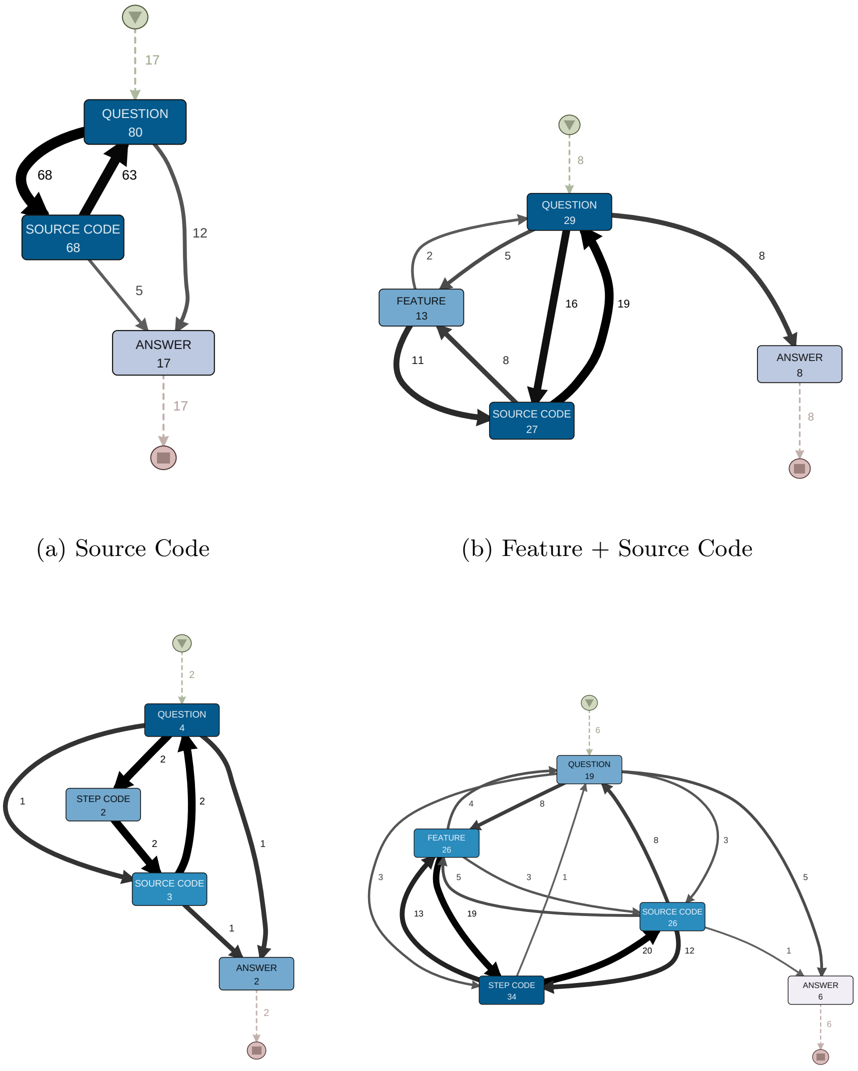
This section describes the investigations needed to answer RQ2.1 by understanding how developers engaged with different artifacts in order to answer code understanding questions. Similarly to the procedure we reported in Section 5.1, we mined the maps with activities as files and activities as artifacts for each group of subjects reported in Table 2. Since all the 11 subjects used at least one artifact to answer the three code understanding questions, we have a total of 33 process instances, distributed among the four groups of subjects. Please note that all raw data as well as larger pictures of the process maps are available online at `https://zenodo.org/record/2602995`.

**Artifact type abstraction.** The maps referring to the behavior of the subjects with artifact type abstraction (cf. Fig. 4) is reported in Fig. 9. As discussed in Section 5.1, due to the study design and to the data analysis technique (cf. Section 4.2), all maps always start with fixations on questions and always terminate with fixation on answers. In addition, it is possible to note that the complexity of the pattern increases proportionally with the number of different artifacts being used: when subjects only used the source code (Fig. 9a) we see the simplest pattern, then come the two patterns with two different artifacts each (i.e., Feature + Source Code as in Fig. 9b, and Step Code + Source Code as in Fig. 9c), and finally the pattern where all three artifacts are used to answer (Fig. 9d) which shows the most complex behavior. However, in this last case (Fig. 9d), the thickness of the edges suggests that subjects went most often from the question to the feature files, then to the step code and finally to the source code. It is important to note that this ideal behavior, in the map, is actually diluted because several subjects repeated this whole sequence (or only parts of it) several times.

**Files abstraction.** Fig. 10 reports the patterns followed when answering code understanding questions by focusing on activities emerging from file abstractions (cf. "Specific artifact" in Fig. 4). In case of subjects who just used the source code to answer code understanding questions it is possible to note that the pattern being followed (Fig. 10a) is very complicated and unstructured since most of connections have the same frequency (i.e., all source files have been inspected with no clear order). Such a lack of structure is manifested only at this level of granularity (at the artifact level, i.e., Fig. 9a it was not). For the three other groups of subjects (i.e., Feature + Source Code as in Fig. 10b, Step Code + Source Code as in Fig. 10c, and Feature + Step Code + Source Code as in Fig. 10d) the complexity of the patterns is consistent with observations at the artifact level (cf. resp. Fig. 9b, Fig. 9c, and Fig. 9d).

It is worth to mention that all subjects managed to locate the source code containing the answer to the questions. However, participants who used feature files, step code and source code appeared to have more structured way of reaching the target resource.

In conclusion, we answer RQ2.1 by observing that it has been possible to distribute subjects over four partitions based on the resources used to achieve their answers. Each group showed different interaction patterns when reading the resources. In addition, as for RQ1.1, the two abstractions of activities helped the investigation by starting from a general view and then allowing to inspect



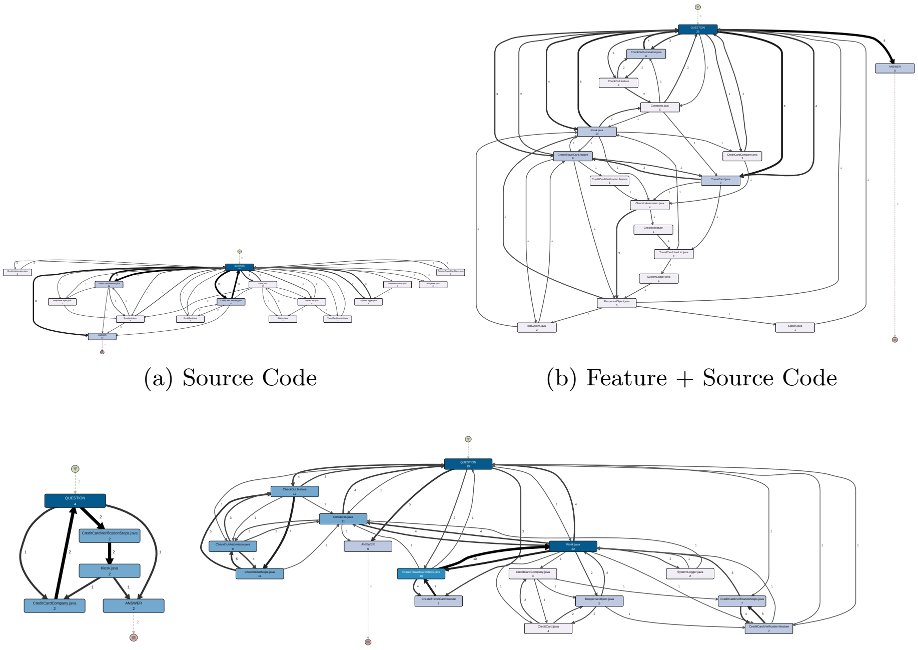
(a) Source Code

(b) Feature + Source Code

(c) Step Code + Source Code

(d) Feature + Step Code + Source Code

Fig. 9: Process maps mined for code understanding questions considering the artifact type abstraction for activities (cf. Fig. 4). Larger figures are available at `https://zenodo.org/record/2602995`.



(a) Source Code

(b) Feature + Source Code

(c) Step Code + Source Code

(d) Feature + Step Code + Source Code

Fig. 10: Process maps mined for code understanding questions considering the file abstraction for activities (cf. Fig. 4). Larger figures are available at `https://zenodo.org/record/2602995`.

more granular information. It is important to note that these questions, referring to code understanding, always required analysis of the actual source code. The navigation patterns seem to suggest that exploring the step code helped subjects to be more precise in the identification of the relevant source code (i.e., the source code with the answer to the question). However, as discussed in Section 5.4, during the post-experiment interview, several participants found the step code as not useful for code understanding as it did not contain the actual answer.

## 5.4 Benefits and Challenges of Software Artifacts for Code Understanding (RQ2.2)

To improve the interpretation of the results presented using the reading patterns, we analyzed the benefits and the challenges associated to the usage of different artifacts. These results are complemented with data from our retrospective think aloud session concerning the perceived benefits and challenges of the different software artifacts used in answering the code understanding tasks.

**Feature File.** Figure 11 depicts the extent to which participants perceive the feature file useful to accomplish the code understanding tasks. The usefulness of a feature file for code understanding is not as prominent as for domain understanding tasks, i.e., 6 out of 11 participants indicate (strong) agreement. This is in close alignment with the reading patterns that demonstrate that 6 out of 11 participants used the feature file for code understanding tasks (i.e., DTU3, DTU8, DTU9, DTU10, UCY1, and UCY3).

Benefits outlined by participants are summarised in Table 4. Feature files are perceived as helpful for code comprehension tasks because they direct them to look at relevant classes and methods in the source code. Subsequently, they were able to find the answer from the source code. However, a feature file itself does not provide the answers for code understanding tasks (unlike in domain understanding tasks). As expressed by one participant:

> "[A feature file] did not help me because it does not specify how the procedure [method] is done." (UCY4)

**Step definition.** Figure 11 depicts the extent to which the participants perceive the step definition useful to answer the code understanding tasks. For code comprehension tasks 3 out of 11 participants found the step code useful (strongly agree). As outlined in Table 5 the participants using the step code particularly found the navigational support provided by the step code helpful. As can be inferred from the reading patterns only 3 out of 11 participants used the step definition for *code understanding* (i.e., DTU3, DTU8, and DTU10). One possible explanation could be that the participants lacked the required familiarity with BDD to fully exploit the navigational support and therefore preferred to use the source code for identifying the answers. Moreover, the step code can only provide navigational support, but does not provide the answer to the question. For

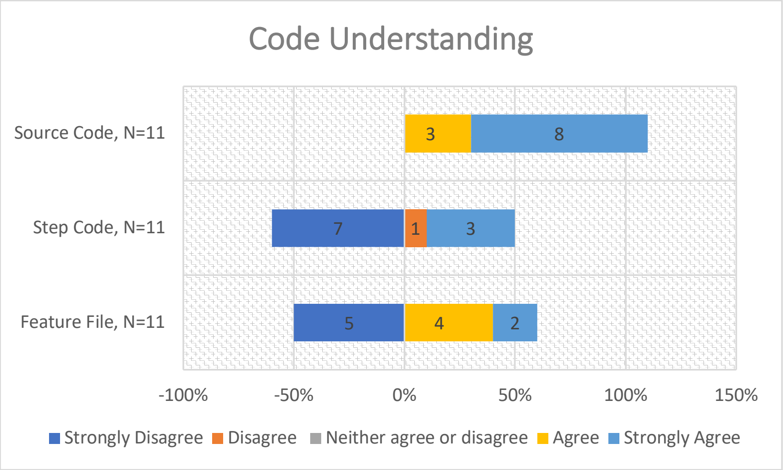


Fig. 11: Extent of usefulness of artifact types (i.e., feature file, step code, source code) for code understanding tasks. N represents the total number of subjects. The numbers in the bars indicate the number of subjects. Strongly disagree indicates that the feature file was not used.

Table 4: Benefits of a feature file for code understanding.

| | Benefits of a feature file | Participants Quotes |
|---|---|---|
| Code Underst. | Mapping to relevant method | "*The feature file helped me to find out what is the method that is being used to verify the card*" (DTU3)<br>"*I used it to find initially where I had to look into the source code*" (UCY3)<br>"*I saw there is a part where it validates, it is a condition to have a valid credit card, linked with a company provided so i checked the source code after, how it matches the credit card with the company provided*" (UCY1)<br>"*The feature file tells you where you will find which piece of code*" (DTU8) |
| | Mapping to the relevant step definition | "*It told me where to look in the step definition*" (DTU3) |

example, participant UCY4 mentioned that the step definition was not useful for code understanding because of the detailed procedure in the source code. Another participant pointed out that to fully leverage the benefits from the step definition to navigate to the corresponding implementation, the IDE should be equipped with a feature that allows you to go to a specific code automatically. However, the Eclipse IDE that we used in this study did not have such a feature: "*You need to have a good tool that allows you to go to [relevant] code*" (DTU3).

Table 5: Benefit of a step definition for code understanding

| | **Benefit of a step definition** | **Participant Quote** |
|---|---|---|
| Code Underst. | Mapping to relevant method | "*The step code [definition] was essential. The step code told me what method is being called*" (DTU3)<br>"*[A step definition] tells you actually even more clearly where exactly I have to look. It tells you in which part of the source code which function is hidden*" (DTU8)<br>"*[A step definition] told me which object and method that I need to look for in the source code* (DTU10) |

**Source Code.** Figure 11 depicts the extent of participants' agreement concerning the usefulness of the code for code understanding tasks. Meanwhile, all of the participants are in (strong) agreement concerning the usefulness of the source code for code understanding. As we can see from Table 6, the participants' responses indicate that the source code is very useful for *code understanding*, primarily because the source code provides a detailed implementation which led them to the answers. The source code is beneficial if the methods, classes, and packages are clearly named.

Table 6: Benefits of the source code for code understanding

| | **Benefits of the source code** | **Participants quotes** |
|---|---|---|
| Code Underst. | Clear naming for methods, classes, and packages | "*The wording in the functions, like by looking at it, I know that there is the answer*" (DTU9)<br>"*I think it was easy to find things because of the naming really to help me understanding the code*" (DTU10)<br>"*[The source code] helped me because [of the] keywords in the question that link to company provider credit card and I found this very easy, because there was a method that states this*" (UCY1).<br>"*[The source code] was very helpful with good structure and correct names of the classes*" (UCY2) |
| | Provide detailed implementation | "*It shows how things are implemented is only in the source code*" (DTU3).<br>"*The source code actually contains the answer*" (DTU8)<br>"*[The source code] told me what detail of what checks are done*" (DTU10)<br>"*The source code help me a lot, because it has the detail procedure*" (UCY4) |

### 5.5 Discussion and Limitations of Study Results

**Discussion of Study Results.** Analyzing our data at the artifact level provided insights into how participants navigated between different artifacts (i.e., feature, step code, and source code) for accomplishing domain and code understanding tasks. Analyzing our data at the file level, in turn, provided some first indications that, when asked for code understanding questions, participants using feature files and step code had more targeted reading patterns (i.e., focus predominantly on files relevant for performing the task at hand). A potential avenue for future work could be a confirmatory study investigating the association of different reading patterns with comprehension task performance.

Our results show that depending on the task to be addressed (domain understanding versus code understanding) different reading patterns can be observed, i.e., participants adjust their reading patterns in a task-dependent manner. Moreover, the choice of reading patterns is closely aligned with the benefits and challenges experienced by participants.

*Domain Understanding.* For domain understanding tasks the feature file was perceived as the most beneficial artifact. Mentioned benefits of a feature file are that it is easy to understand, provides insights into the code's intention, and provides an outline of the requirements. The benefits that we identified further corroborate the benefits of BDD reported in past studies [3, 10, 22]. Participants, however, also experienced a steep learning curve, a challenge that has been also reported in [3]. Moreover, one participant pointed out that a scenario described in one of the feature files had a pre-condition which did not have a corresponding implementation in the source code. Such an issue has been described in [34] as a consequence of adding unnecessary detail to a feature file. Completeness of a feature file is an attribute that practitioners found to be important for a feature file [34]. We could see in our data that one participant who did not trust the completeness of the feature file validated his answer using the step code and source code. Our study complements the study by Oliveira and Marczak [34] on the importance of conforming to quality guidelines in formulating feature files.

*Code Understanding.* For code understanding, in turn, the source code was perceived as the most useful artifact. Still both the feature file and the step code was perceived as an important navigational aid. Concerning the code understanding task, participants who exploited all BDD artifacts (i.e., feature, step code and source code) were able to complete their task by inspecting fewer irrelevant Java files. Instead, subjects who used only the source code inspected more Java files, several of them not task relevant. The reading patters are in line with the results of RQ2.2, in Section 5.4. Our participants found that BDD artifacts (i.e., feature and step code) are useful in code understanding, particularly for identifying the relevant part of the source code. This finding supports a previous study that mentions that one of the benefits of BDD is that it improves traceability between requirements and code [22].

A number of our participants also reported that clear naming of methods, classes, methods, and packages helps in code understanding. Past code understanding studies reported that intention revealing variable names improve code understanding [4, 23, 38]. The finding of our study indicates that the intention revealing variable names in the source code might have rendered the step definition less important. To investigate whether BDD uptake increases in the presence of obfuscated code or whether its benefits are more clearly seen for code understanding is subject for a future study.

A challenge that was reported by one of our participants pertains to IDE limitations. The study conducted by Solis and Wang [46] suggest that current BDD tools like Cucumber, SpecFlow, and the xBehave family, lack comprehensive support for the BDD approach. Our study identified another limitation of existing BDD tools, i.e., the lack of support in linking a feature file or a step definition and source code (in Section 5.4).

**Limitations of Conducted Study.** This subsection describes limitations concerning the conducted study. While we could demonstrate that process mining can be used to identify and visualize reading patterns from eye-tracking data during comprehension tasks, we cannot make general statements concerning the distribution of the observed patterns under varying conditions. In the rest of this section several limitation are discussed in detail.

*Task Description.* In our study participants had to read different resources like feature files, step definitions, and source code. This means that, for example, if the scenarios in a feature file are poorly formulated, participants might have difficulty in understanding it, which might, in turn, lead to a different perception of a certain artifact in terms of usefulness and subsequently to different reading patterns (e.g., decreased usage of the artifact). To minimize this threat, the artifacts were inspected by other researchers who did not participate in the study design nor in the final study. In addition, we performed a pilot study with one individual who was not included as a study participant to assess the quality of the study design in general.

The distribution of reading patterns might change depending on the quality of the different artifacts. Several of our participants outlined the high source code quality (cf. Table 6). However, one could speculate that the usefulness of the source code is perceived as lower in the absence of intentional revealing names, while, in turn, the navigational support provided by the step code (cf. Table 5) could be perceived as more useful and consequently resulting in a higher usage of the step code files. As for future work we plan to conduct an experiment where the task materials differ in terms of source code quality. We expect that feature files and step code files will be increasingly used when the source code lacks intention revealing names and we expect a positive impact on task performance through the navigational support provided.

*Selection of Participants.* To recruit the participants we used convenience sampling. This selection strategy, as we mentioned in Section 4.1, led to the inclusion of participants who were unfamiliar with BDD, which might prevent us from capturing the intended construct. To minimize sampling threat, we provided training and introduction to BDD to all participants. Our study shows that some of the participants with little to no familiarity with BDD managed to use the BDD artifacts (i.e., feature file as well as step code) during the tasks. Furthermore, our study aimed at better understanding how developers' interact with different software artifacts when answering domain and code understanding questions and not at drawing any conclusions about cause-effect relationships (e.g., how the observed patterns are associated with comprehension task performance).

*Evaluation Apprehension.* In this study, an observer was present while the participant worked on completing the task. This was done to allow proper calibration of the eye tracker and also to administer the think aloud questions. Having an observer present may bias the results, as the participants might try to perform better when being evaluated, also known as the *Hawthorne effect.* To minimize this effect, we mentioned to the participants that we are interested in the way they interact with the different software artifacts and not in judging their performance.

*Low Number of Participants.* Another threat to external validity is the rather low number of participants, i.e., 11 participants. Goldberg and Kotval performed an eye-tracking study with 12 participants and mentioned that it is a typical number in an eye-tracking study [13]. Despite the small sample size we could for both domain and code understanding tasks observe different reading patterns. The process maps provide some indication that the presence of feature files and step code could be associated with improved task performance. Due to the small sample size we analyzed our data qualitatively and did not draw conclusions using statistical inferences. As for future work we plan to conduct a confirmatory experiment to investigate the impact of feature files and step code on comprehension task performance.

## 6 Discussion and Limitations

This section discusses the results as well as limitations of the general method we proposed in Section 3.

### 6.1 Discussion of the Proposed Method

Our results show that process mining can be used to identify reading patterns from eye-tracking data and visualize how participants interact with different resources at different levels of granularity. We demonstrated the approach by conducting an exploratory study with 11 participants and complemented the analysis of reading patterns with think aloud data providing insights into perceived benefits and challenges associated with the different resources to better understand the choice of reading patterns.

With the method we demonstrated also the partitioning of the subjects based on which activities each of them engaged with to answer a questions. Corresponding reading patterns, for each partition of the subjects, can be extracted and analyzed. In addition, we show that the reading patterns can be generated for different abstraction levels: we focused on activities as reading each file or activities as reading files that belong to the same artifact type (i.e., feature, or step code, or source code).

The reading patterns identification methodology presented in this paper can represent an important data exploration tool to investigate the typical behavior aggregated by all subjects (within a certain group). This information can be used as input for confirmatory studies as well as to improve supporting tools to better reflect user needs. In addition, these maps might be useful when it is important to verify that subjects follow some prescriptive behavior (en example of such behavior is to double check different resources before providing an answer).

Please note that the visualisation of reading patterns requires the application of abstraction going beyond eye-tracking metrics (such as fixations and saccades). With eye movement data collected at a sampling rate of 90Hz (common in consumer eye-trackers, e.g., in the case of Tobii 4C) we obtain approximately 54000 gaze points for every 10 minutes of data recording. Traditional eye-tracking research usually employs the notion of Areas of Interest [16] to provide spatial abstractions. While this can be easily achieved for settings where all artifacts are visible at all time, our setting is of more dynamic nature, i.e., the artifacts visible to the participants change over time. By using iTrace and interlinking gaze points with source code entities we are able to map the data onto activities (corresponding to dynamic Areas of Interest) and visualize them using process mining in a flat 2-dimensional map. In traditional eye-tracking settings artifacts are either visible at all time or the identification of dynamic areas of interest has to be done manually [16]. Moreover, traditional visualisations for such settings require video (e.g., showing the scan-path over time).

### 6.2 Limitations of the Proposed Method

This subsection describes limitations concerning the usage of process mining to identify and visualize reading patterns derived from eye-tracking data.

*Data Collection Using Eye-Tracking.* The method presented in this paper relies on eye-tracking data. Eye-trackers are hardware equipment subject to two main problems: accuracy in space and accuracy in time. Space accuracy concerns the precision in detecting the position of a fixation. For example, if we want to have a very fine grained abstraction on the reading patterns (e.g., which methods are read, more precise than what presented in Fig. 4), then we would need to ensure that methods appear far enough from another, to compensate possible mistakes on the fixation coordinates identified by the eye tracker. The time accuracy of an eye tracker refers to the pace at which the fixations are detected. This time has to be commensurate to the amount of information we expect subjects to read. If this is not the case, it may happen that fixations on specific resources are not detected which, in turn, results in missing activities. However, for most scenarios this does not represent a problem, since even consumer eye trackers have sampling rate of 90Hz (e.g., in the case of Tobii 4C).

*Linking Eye-Tracking Data with Artifacts.* To collect the data and to link the eye movement data with the actual artifacts, there is need for specific software. In our demonstration we used the iTrace plugin, which assumes for its correct operation that the source code is not altered during a session. Therefore, with the tool being currently used, it is possible to investigate only comprehension tasks, where the resources are static throughout the duration of the whole experiment. Ideally, however, this limitation can be overcome by adopting tools which are capable of mapping fixations to artifacts which are changing over time. Then, the very same method can be used to investigate not only comprehension, but also design tasks.

## 7 Conclusions and Future Work

In this paper, we proposed a method to identify and visualize reading patterns occurring during comprehension tasks. We applied our method in an exploratory study investigating developers' behavior using gaze data collected using an eye tracker and a verbal questionnaire.

We could demonstrate that our method can be used to partition the set of subjects into different groups according to their reading patterns. The qualitative analysis of the post-experiment questionnaire revealed the benefits and challenges associated with the different artifacts providing some insights into the choice of reading patterns.

Since the current method is limited to tasks involving artifacts that remain unchanged over time, we plan to extend our work towards evolving artifacts, which would allow the application of the method to design tasks.

Moreover, we plan to use the maps elicited in our exploratory study as a starting point for future confirmatory studies. For example, the process maps provide some indication that the presence of feature files and step code could be associated with improved task performance. Moreover, there is some indication that the high quality of the source code made it less necessary for participants to rely on the navigational support provided by BDD artifacts. Both of these aspects could be investigated in the future. Additionally, it would be highly interesting to combine the method proposed in this paper with research on cognitive activities similar to [43, 12, 51, 20]. This would allow us to combine reading patterns with insights into cognitive processes.

## References

1. Wil van der Aalst. Big software on the run: In vivo software analytics based on process mining (keynote). In *Proceedings of ICSSP 2015*, pages 1–5. ACM, 2015.

2. P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile Software Development Methods – Review and Analysis. Technical Report 478, VTT Publications, 2002.
3. L. P. Binamungu, S. M. Embury, and N. Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–184, March 2018.
4. Scott Blinman and Andy Cockburn. Program comprehension: Investigating the effects of naming style and documentation. In *Proceedings of the Sixth Australasian Conference on User Interface - Volume 40*, AUIC '05, pages 73–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
5. Barry Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10:4–21, 1984.
6. Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
7. Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 1–9, New York, NY, USA, 2011. ACM.
8. Krzysztof J. Cios, Witold Pedrycz, and Roman W. Swiniarski. *Data Mining Methods for Knowledge Discovery.* Springer US, Boston, MA, 1998.
9. M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, Jan 1990.
10. Pedro Lopes de Souza, Antônio Francisco do Prado, Wanderley Lopes de Souza, Sissi Marilia dos Santos Forghieri Pereira, and Luís Ferreira Pires. Combining behaviour-driven development with scrum for software development in the education domain. In *ICEIS (2)*, pages 449–458. SciTePress, 2017.
11. Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 402–413, New York, NY, USA, 2014. ACM.
12. Andrew Gemino and Yair Wand. A framework for empirical evaluation of conceptual modeling techniques. *Requirements Engineering*, 9(4):248–260, Nov 2004.
13. Joseph H Goldberg and Xerxes P Kotval. Computer interface evaluation using eye movements: methods and constructs. *International Journal of Industrial Ergonomics*, 24(6):631 – 645, 1999.
14. Daniel Graziotin, Xiaofeng Wang, and Pekka Abrahamsson. Do feelings matter? on the correlation of affects and the self-assessed productivity in software engineering. *Journal of Software: Evolution and Process*, 27(7):467–487, 2015.
15. Jens Gulden, Andrea Burattin, Amine A. Andaloussi, and Barbara Weber. From analytical purposes to data visualizations: a decision process guided by a conceptual framework and eye tracking. *Software & Systems Modeling*, Jul 2019.
16. Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. *Eye tracking: A comprehensive guide to methods and measures.* OUP Oxford, 2011.
17. IEEE Task Force on Process Mining. Process Mining Manifesto. *Business Process Management Workshops*, pages 169–194, 2011.
18. Constantina Ioannou, Andrea Burattin, and Barbara Weber. Mining developers' workflows from ide usage. In Raimundas Matulevičius and Remco Dijkman, editors, *Advanced Information Systems Engineering Workshops*, pages 167–179, Cham, 2018. Springer International Publishing.

19. Katja Kevic, Braden M. Walters, Timothy R. Shaffer, Bonita Sharif, David C. Shepherd, and Thomas Fritz. Tracing software developers' eyes and interactions for change tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 202–213, New York, NY, USA, 2015. ACM.
20. Jinwoo Kim, Jungpil Hahn, and Hyoungmee Hahn. How do we understand a system with (so) many diagrams? cognitive integration processes in diagrammatic reasoning. *Information Systems Research*, 11(3):284–303, 2000.
21. Ekkart Kindler, Vladimir Rubin, and Wilhelm Schäfer. Incremental workflow mining based on document versioning information. In *Software Process Workshop*, volume 3840 of *LNCS*, pages 287–301. Springer, May 2005.
22. T.M. King, G. Nunez, D. Santiago, A. Cando, and C. Mack. Legend: An agile dsl toolset for web acceptance testing. pages 409–412, 2014.
23. Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, Dec 2007.
24. M. Leemans and W. M. P. van der Aalst. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *Proceedings of MODELS*, pages 44–53, Sept 2015.
25. Maikel Leemans, Wil M. P. van der Aalst, and Mark G. J. van den Brand. Recursion aware modeling and discovery for hierarchical software event log analysis (extended). *CoRR*, abs/1710.09323, 2017.
26. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency.* Springer, 2013.
27. Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, 2015.
28. Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987.
29. Cong Liu, Boudewijn F. van Dongen, Nour Assy, and Wil M. P. Aalst. Software architectural model discovery from execution data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 03 2018.
30. Cong Liu, B. van Dongen, N. Assy, and W. M. P. van der Aalst. Component behavior discovery from software execution data. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Dec 2016.
31. S. C. Mller and T. Fritz. Stakeholders' information needs for artifacts and their dependencies in a real world context. In *2013 IEEE International Conference on Software Maintenance*, pages 290–299, Sept 2013.
32. Dan North. Introducing BDD, 2006. `http://dannorth.net/introducing-bdd/`. Last Accessed: October 2018.
33. Indira Nurdiani, Jürgen Börstler, and Samuel Fricker. The Impacts of Agile and Lean Practices on Project Constraints: A Tertiary Study. *Journal of Systems and Software*, 119:162–183, 2016.
34. Gabriel Oliveira and Sabrina Marczak. On the understanding of bdd scenarios' quality: Preliminary practitioners' opinions. In Erik Kamsties, Jennifer Horkoff, and Fabiano Dalpiaz, editors, *Requirements Engineering: Foundation for Software Quality*, pages 290–296. Springer International Publishing, 2018.
35. Nancy Pennington. Empirical studies of programmers: Second workshop. chapter Comprehension Strategies in Programming, pages 100–113. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

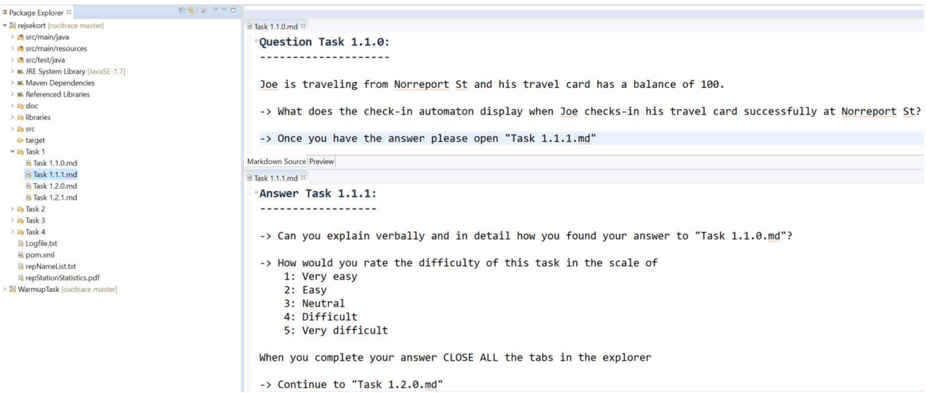
36. Vladimir Rubin, Christian W. Günther, Wil M. P. van der Aalst, Ekkart Kindler, Boudewijn F. van Dongen, and Wilhelm Schäfer. Process mining framework for software processes. In *Proceedings of ICSP 2007*, pages 169–181. Springer, 2007.
37. Johnny Saldaña. *The Coding Manual for Qualitative Researchers.* SAGE Publications Limited, 2012.
38. Felice Salviulo and Giuseppe Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 48:1–48:10, New York, NY, USA, 2014. ACM.
39. I. Schröter, J. Krüger, J. Siegmund, and T. Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311, May 2017.
40. Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 954–957, New York, NY, USA, 2015. ACM.
41. Z. Sharafi, Z. Soh, Y. Guéhéneuc, and G. Antoniol. Women and men – different but equal: On the impact of identifier style on source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 27–36, June 2012.
42. Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, 67:79–107, 2015.
43. Keng Leng Siau. *Empirical studies in information modeling: Interpretation of the object relationship.* PhD thesis, University of British Columbia, 1996.
44. John Ferguson Smart. *BDD in Action.* Manning Publications, 2014.
45. IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.* IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.
46. C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, Aug 2011.
47. Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. *The nature of expertise*, pages 129–152, 1988.
48. W.M.P. van der Aalst. *Process Mining: Data Science in Action.* Springer, 2016.
49. A. M. Vans and A. von Mayrhauser. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 08 1995.
50. Version One. 12th Annual State of Agile$^{\mathrm{TM}}$Report, 2017. Last Accessed: December 2018 .
51. Iris Vessey and Sue A. Conger. Requirements specification: learning object, process, and data methodologies. *Communications of the ACM*, May 1994.
52. Braden Walters, Timothy Shaffer, Bonita Sharif, and Huzefa Kagdi. Capturing software traceability links from developers' eye gazes. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 201–204, New York, NY, USA, 2014. ACM.
53. Yang Wang and Stefan Wagner. Combining stpa and bdd for safety analysis and verification in agile development: A controlled experiment. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering*

*and Extreme Programming*, pages 37–53, Cham, 2018. Springer International Publishing.

54. Roelf J. Wieringa. *Design science methodology for information systems and software engineering.* Springer, 2014. 10.1007/978-3-662-43839-8.

## A Task Description

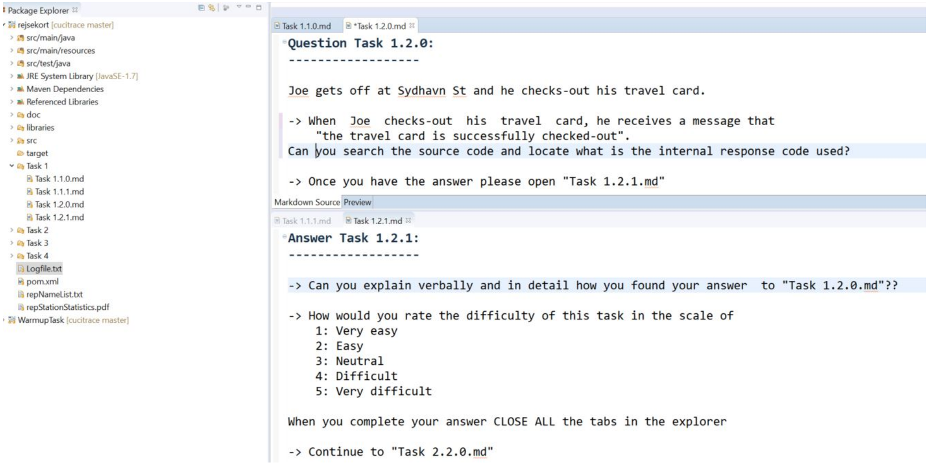
Tasks consist of answering questions regarding domain understanding and code understanding. In this appendix, we present two screenshots with question and corresponding answer files along with the structure of the software artifacts. We choose to present Task 1.1.0 and Task 1.2.0 as example. Task 1.1.0 is a domain understanding question (see Fig. 12) whereas Task 1.2.0 is a code understanding question (see Fig. 13). The remaining tasks are available at `https://github.com/CIoann/cucitrace`.



```
Question Task 1.1.0:
--------------------

Joe is traveling from Norreport St and his travel card has a balance of 100.

-> What does the check-in automaton display when Joe checks-in his travel card successfully at Norreport St?

-> Once you have the answer please open "Task 1.1.1.md"
```

```
Answer Task 1.1.1:
------------------

-> Can you explain verbally and in detail how you found your answer to "Task 1.1.0.md"?

-> How would you rate the difficulty of this task in the scale of
    1: Very easy
    2: Easy
    3: Neutral
    4: Difficult
    5: Very difficult

When you complete your answer CLOSE ALL the tabs in the explorer

-> Continue to "Task 1.2.0.md"
```

Fig. 12: The domain understanding question for Task 1.1.0 and the answer Task 1.1.1 files are shown



```
Question Task 1.2.0:
------------------

Joe gets off at Sydhavn St and he checks-out his travel card.

-> When  Joe  checks-out  his  travel  card, he receives a message that
    "the travel card is successfully checked-out".
Can you search the source code and locate what is the internal response code used?

-> Once you have the answer please open "Task 1.2.1.md"
```

```
Answer Task 1.2.1:
------------------

-> Can you explain verbally and in detail how you found your answer  to "Task 1.2.0.md"??

-> How would you rate the difficulty of this task in the scale of
    1: Very easy
    2: Easy
    3: Neutral
    4: Difficult
    5: Very difficult

When you complete your answer CLOSE ALL the tabs in the explorer

-> Continue to "Task 2.2.0.md"
```

Fig. 13: The code understanding question for Task 1.2.0 and the answer Task 1.2.1 files are shown