AN ABSTRACT OF THE THESIS OF

Marjan Adeli for the degree of Master of Science in Computer Science presented on March 19, 2020.

Title: Facilitating Code Comprehension by Annotations in Canvas-Based IDE

Abstract approved: ______________________________

Anita Sarma

Developers spend a considerable amount of time comprehending code and building accurate mental models of the code. Understanding the relationships between software features within IDEs is difficult, with information split across different visual hierarchies making navigation cumbersome. Canvas-based IDEs mitigate some of the navigation costs by allowing relevant information to be presented in groups. However, these groups have no explicit way of capturing and sharing the meaning of different spatial layouts. In this thesis, we present annotations in a canvas-based IDE called Synectic to address this concern. Synectic allows users to arrange relevant information in groups, attach meaning to the arrangement, and externalize thoughts and relationships between artifacts through annotations. To study the effects of these annotations on comprehension, we conducted a user study of newcomers performing code comprehension tasks comparing Synectic and Eclipse. The results show annotations in Synectic increase the developer's accuracy, while reducing cognitive load during newcomer comprehension tasks.

Facilitating Code Comprehension by Annotations in Canvas-Based IDE

by

Marjan Adeli

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented March 19, 2020
Commencement June 2020

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my advisor, Dr. Anita Sarma, without whose constant guidance and support I would not have been able to achieve this. I am also indebted to Dr. Amir Nayyeri for his support throughout my early graduate years, and through many tough times. I would also like to extend my gratitude to my committee members Dr. Rakesh Bobba and Dr. Leonard Coop.

A very special gratitude goes out to my team members who I worked closely with. Thank you Souti Chattopadhyay and Nicholas Matthew Nelson.

I also express my gratitude to my family and friends for their understanding and encouragement throughout my years of study.

Finally, I appreciate the National Science Foundation for partially supporting this project under the NSF Cooperative Agreement No. 1560526.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

Code comprehension comprises a large portion of developers activities. For example, developers were found to spend 58% of their effort on code comprehension [44]. The effort to comprehend code is especially higher in large software projects [11]. To help with comprehension, researchers have conducted several studies of developers' process of code comprehension [37, 16, 3]. For example, Sillito et al. [37] identified four categories of questions developers ask during maintenance tasks. These categories represent a model for understanding how developers explore and understand code and highlight the process of finding a focus point, expanding understanding around that point, and contextualizing that newly discovered knowledge within the larger codebase. Throughout each of these categories of questions, developers must "forage" for relevant information in order to build their mental model of the system. Prior research has shown that foraging accounts for 35% of the developer's time during maintenance tasks [20, 8].

Moreover, developers face tremendous barriers while foraging for information. One study found that 50% of the navigation yielded less information than developers expected and 40% of navigation required more effort than the developer predicted [30]. The cost of foraging would likely be considerably higher for newcomers to a project, who haven't yet formed a mental model of the feature set and the different relationship between those features.

In particular, developers have difficulty understanding the relationships between software features within traditional IDEs[12, 7]. A possible reason is that the features in IDEs for managing multiple artifacts is unordered tab panes. Moreover, when required information is split across different hierarchies (e.g. file explorer, tabs, panes, etc.), navigation becomes cumbersome. This problem is further exacerbated when we consider that software development requires artifacts beyond the codebase (e.g. documentation, code examples, debugging outputs).

In this thesis, towards addressing these concerns, we focus our interest on three needs of developers during software development: (1) using artifacts that are more than just code, (2) placing relevant information closer together (either spatially or in groups), and (3) externalizing and recording relationships between artifacts through annotations. Past canvas-based approaches have been created to ease navigation and understand project structure [2, 7, 6, 5, 12]. These systems have addressed (1) to some extent (e.g. via augmenting code with debugger output inline [6]), (2) based primarily on syntactic linking (e.g. bubbles containing code linked together by call graph relationships [2]), but none have directly addressed (3).

To the best of our knowledge, no in-depth studies have been conducted to determine whether and how freedom to group and arrange artifacts, and link relevant information patches in-situ aid in code comprehension.

To address this gap in knowledge, we presented annotations in a canvas-based IDE, Synectic, which allows relevant information to be arranged and group according to the user needs, and externalizes relationships through annotations. To evaluate how annotations help with code comprehension, we conducted a user study comparing how newcomers to a project perform code comprehension tasks within a traditional IDE (Eclipse) and our canvas-based IDE (Synectic). In particular, the evaluation investigated four research questions:

RQ1: Do annotations affect the accuracy of code comprehension during newcomer's onboarding tasks?

RQ2: Do annotations affect the time of code comprehension during newcomer's onboarding tasks?

RQ3: Do annotations affect newcomer's cognitive load during onboarding tasks that require code comprehension?

RQ4: Do users perceive annotations to be useful for code comprehension?

We chose newcomers for our user study since they require significant effort towards comprehending and building understanding of a codebase. Additionally, newcomers have not previously built mental models or developed navigation habits for working with a specific codebase. We asked participants to complete four comprehension tasks; each task comprised two parts–to locate code relevant to a particular feature, followed by a question requiring deeper understanding of that feature.

We analysed our observations using two lenses. We identified the hurdles that newcomers encounter in traditional IDEs (Eclipse) as compared to canvas-based IDEs (Synectic) using the Information Foraging Theory (IFT) [32] lens. We then examined support for newcomers' code comprehension using the four categories of comprehension questions defined by Sillito et al. [37] to see whether annotations and links can alleviate some of the difficulties.

The rest of this thesis is organized as follows. Chapter 2 presents a review of the background and prior research related to this work. Chapter 3 provides a brief description of Synectic and its annotations feature. Chapter 4 describes the methodology of the user study we conducted. In Chapter 5, we present the results of the user study. Chapter 6 builds upon the collected metrics from users doing the code comprehension tasks and relates them to (i) the hurdles to foraging that newcomers encounter in traditional IDEs, and (ii) the support for code comprehension in our canvas-based IDE. Finally, Chapter 7 concludes the thesis with a brief summary of the key takeaways.

Chapter 2: Background

2.1 Information Foraging Theory

Information Foraging Theory (IFT) explains and predicts how humans seek information within information-rich environments. This provides a theoretical foundation to investigate why some software engineering tools fail, or succeed, at supporting software developers' work. IFT [32] explains how humans seek information is analogous to how animals forage for prey in the wild. Originally applied to user-web interactions, researchers expanded IFT to suitably explain human behavior during software development [20, 21, 8, 27, 28]. IFT has also be applied to design tools supporting development activities [8, 29, 20].

IFT describes a human seeking information as akin to a *predator* (person seeking information) pursuing *prey* (valuable sources of information) through a collection of *patches* of information in an informational environment. Patches are connected by traversable *links* that can lead to other patches of information. Each patch contains *information features* that the predator can process.

The information features have value, as well as cost (in the form of time for the human to read and process them). Traversing a link also has a cost (time to go from one patch to the other). Figure 2.1 graphically depicts an example of two information patches. The central proposition of IFT is that a predator tries to maximize the value of information gained from a patch over the cost of traversing to the patch and processing the information [30]. However, predators cannot accurately determine a patch's value and cost prior to processing, so they make choices based on their expectations of value and cost. These expectations are based on the previously processed information, and perceived potential for future patches.

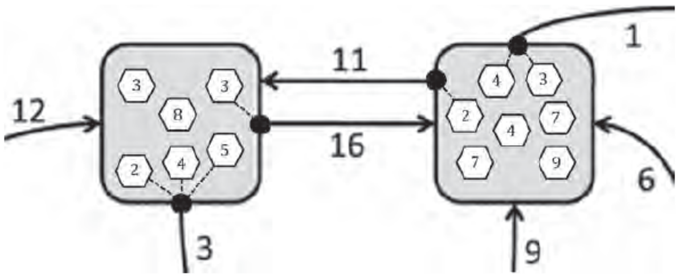


Figure 2.1: An example of two information patches

The directed edges is a navigable link from one patch to another, and the weight on each link indicates the cost of traversing. Each patch contains a set of information features, depicted as hexagons and the number inside each hexagon shows the procesing cost of the information feature. Figure adapted from [33].

2.2 Code Comprehension Questions

Developers decide on the value of particular patches of information during foraging, and in particular they must read and comprehend the underlying code in order to evaluate whether it is relevant to their specific task. This comprehension process requires that developers formulate and ask questions of the underlying information (primarily comprised of code in the case of software development).

Several studies have examined the types of information that developers seek during development tasks [16, 37, 36, 18, 22, 40]. In particular, we use the four categories of code comprehension questions identified by Sillito et al. [37]. These categories represent a model for understanding how developers explore and understand code during change tasks, and consist of: (1) Questions related to locating an initial focus point, (2) questions about expanding relevant entities through exploring relationships, (3) questions used for building understanding of concepts that span multiple entities, and (4) contextual questions that build upon knowledge that spans multiple groups of related entities. Figure 2.2 illustrates these four categories.

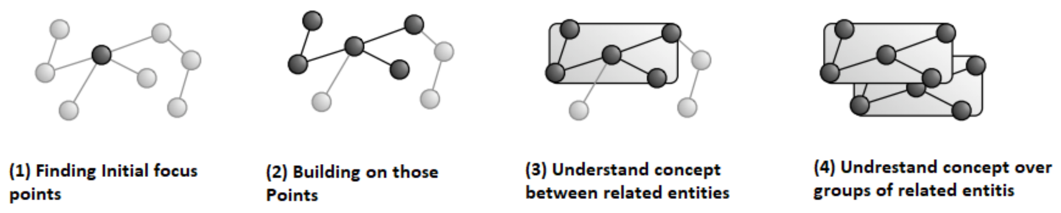

(1) Finding Initial focus points
(2) Building on those Points
(3) Understand concept between related entities
(4) Undrestand concept over groups of related entitis

Figure 2.2: An overview of four categories of code comprehension questions
An overview of the four categories of code comprehension questions illustrated by diagrams depicting source code entities along with connections between those entities. Figure adapted from [37].

These questions enable developers to build mental models of the underlying code and facilitate the ability to confidently make changes to that code [19]. Combined with IFT, these models provide a foundation for examining the effectiveness and efficiency of developer tools during change tasks.

2.3 Alternative user interfaces in IDEs

Traditional file-based IDEs use individual code files as the core component that all interactions and interfaces are designed around. This model creates barriers to effortless coding. Prior literature has shown that developers working in traditional IDEs spend considerable time foraging for relevant information [31], navigating code [15], and managing context when switching tasks and environments [14].

Alternative IDEs attempt to tackle some of these deficiencies through novel user interfaces. Code Bubble replaced the IDE's typical set of tabbed windows with a pan-and-zoom interface [2]. Code Bubble presented the code as a collection of lightweight, editable fragments called bubbles (see Figure 2.3). It allowed the relevant bubble to be clustered in a group of bubbles. Code Bubble also linked the relevant code bubbles together (through call graph relationship) during debugging. Another IDE, Code Canvas, Also was presented with the similar idea to Code bubble, and at the same time [7]. Later Code Bubble and Code Canvas teams collaborated to release Debugger Canvas, an industrial version of the Code Bubbles paradigm [6].

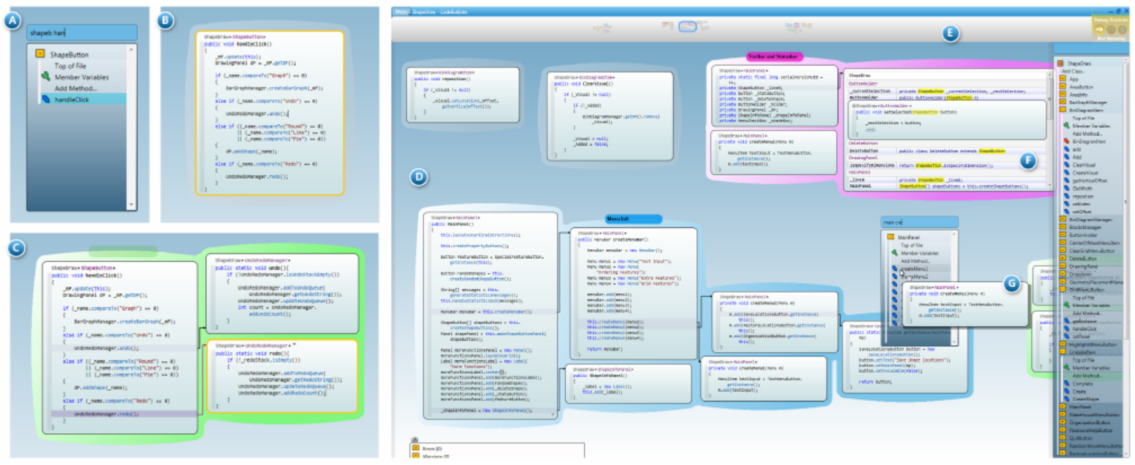


Figure 2.3: Code Bubble interface

An example of a working set of bubbles. (a) User opens a bubble via the pop-up search box, (b) resulting bubble, (c) user opens definition of two more bubbles side-by-side (automatically grouped); (d) a large working set of bubbles, including a (f) bubble stack of references; (e) an overview is shown in the panning bar; (g) hover preview. Figure adapted from [2]. An example of a working set of bubbles.

The Patchworks code editor was another design that based on the idiom of a sliding grid strip (called ribbon) of code fragments (called patches) [12]. The programmer can use the ribbon view to adjustthe visible patch grid. It was aimed to ease navigation among open patches. Figure 2.4 depicts the Patchworks interface.

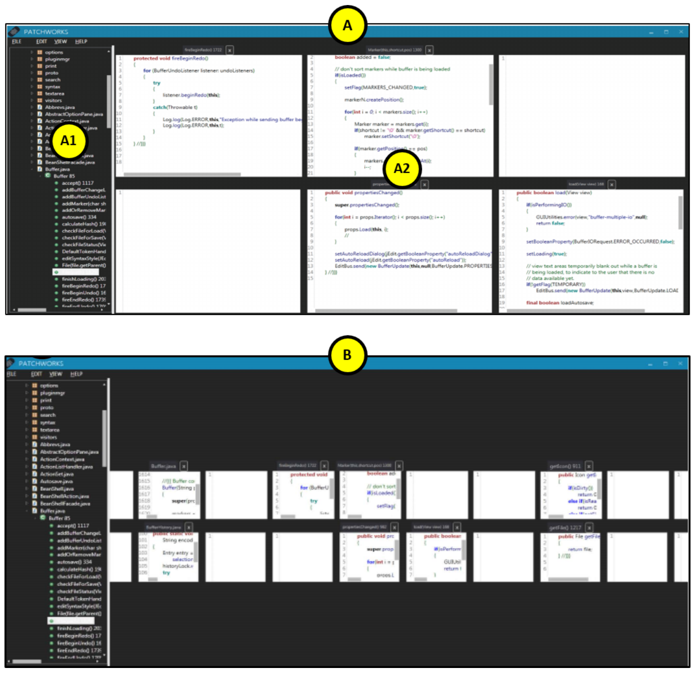


Figure 2.4: Patchworks interface

A Patchwork editor including, the patch-grid view (A) and the ribbon view (B). The patch-grid view includes the package explorer (A1) and a 2 × 3 patch grid (A2). Four of the patches contain code fragments and two are empty. The displayed patch grid is actually a view into a larger patch ribbon (B) and the programmer can navigate by sliding the view left or right along the ribbon. Figure adapted from [12].

All of these alternative IDEs have sought to reduce the costs of context switching when navigating to relevant code, and lower the cognitive load required to understand and operate on that code. However, improvements in these dimensions are still possible and further work can help developers using both traditional and alternative IDEs (since features developed in alternative IDEs don't have to remain exclusive to those IDEs).

Chapter 3: Annotations in Synectic

Synectic is an IDE designed as a canvas-based environment, similar to Code Bubbles [2] and Moldable Debugger [5], with spatial cognition as the central interaction paradigm. Spatial cognition involves the human capabilities of object recognition, object search, and navigation through space to enhance learning and knowledge acquisition [17]. Conventional user interfaces impede the power of spatial cognition and reduces the amount of usable information that can be embedded within a user interface.

Synectic provides a spatially-oriented interface that mimics cards on a canvas in order to allow contextually relevant information to be arranged and grouped according to the needs of the user. To further capture the relationships between these cards, we include an annotation overlay that includes annotations and links that can be attached to cards placed on the canvas. Figure 3.1 presents a snapshot of the Synectic interface, including three cards containing code editors and a browser (displaying API documentation in this example), as well as annotations (shown as yellow boxes) with links between cards and annotations (shown as black lines). These annotations can attach to a single card (as shown in the bottom half of Figure 3.1), or linked between two cards in order to annotate relevant relationships between information contained within those cards (as shown in the top half of Figure 3.1). This annotation overlay is provided in order to capture and express the previously hidden relationships that developers intuitively create in their minds [19].

3.1 Support for Foraging

Information Foraging Theory (IFT) [32, 21, 30] provides a model for understanding the needs of developers navigating for relevant information patches within an IDE (see Section 2.1). And according to IFT, the costs of foraging are directly related to the level of navigation affordances provided in the user interface. Maintaining code often requires revisiting the same prey multiple times and gathering information that spans multiple information features [8, 16]. The navigation pathways undertaken to locate a particular prey or information feature are typically left to the memory capacity of individual developers, and if deemed valuable, might remain open on the screen for future use; this process is ad hoc and temporary.

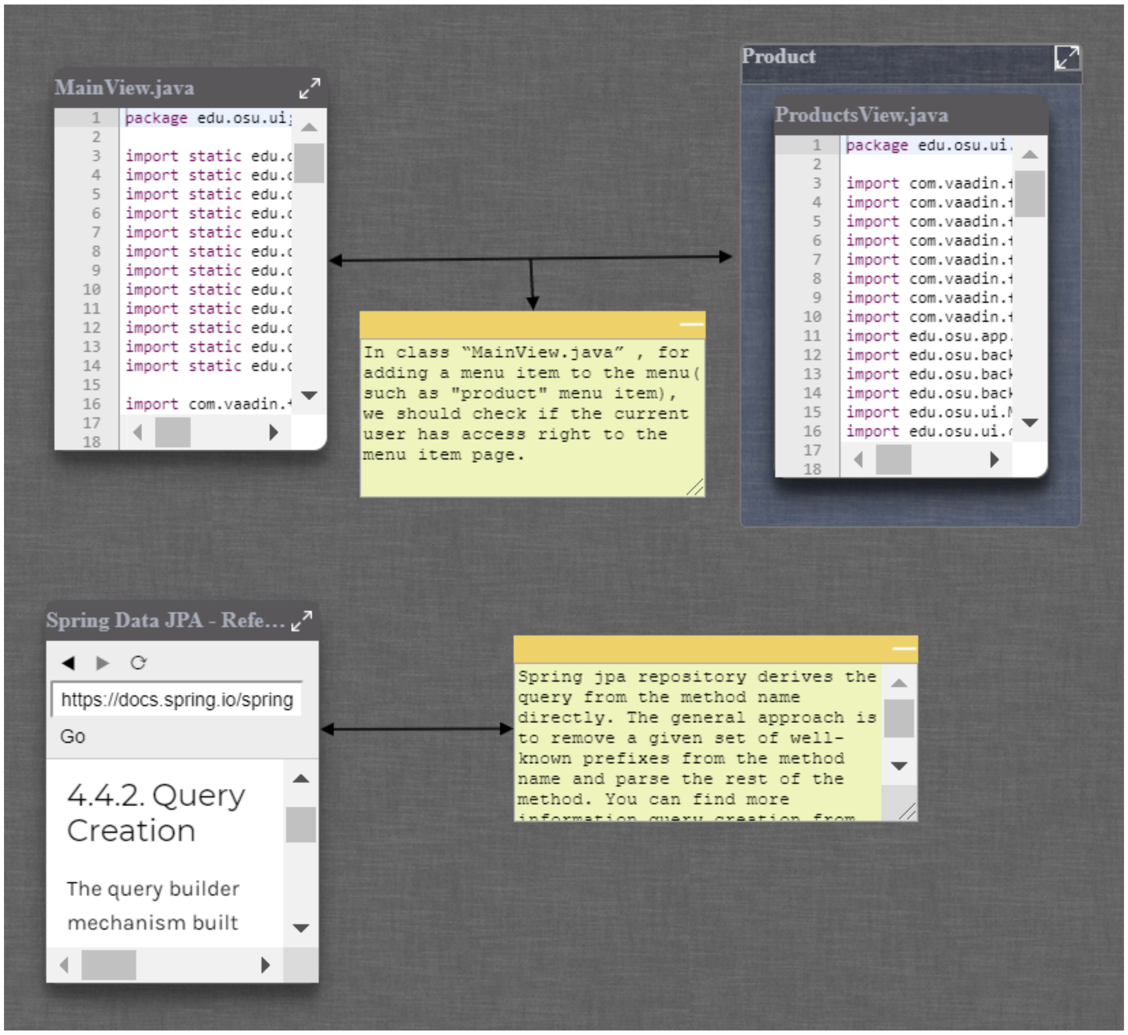
The annotation features within Synectic provide a system for exposing and archiving these pathways so that revisiting prey and information features for similar tasks does not incur the same costs as the first foraging session. The annotations within Synectic allow for both individual notes that are attached to a single card (i.e. prey containing possibly relevant information features), and notes that are attached to two cards in order to visually represent relationships that span multiple information sources. Although these features do not reduce the initial costs of foraging when a task is being undertaken for the first time, all subsequent foraging sessions can benefit from the presence of notes that indicate the scope of tasks, value of information features, and navigation pathways for particular cards (i.e. prey containing information features).

3.2 Support for Understanding

Developers use IDEs (and code editors) to solve problems that expand beyond a monolithic model of code. This requires developers to deal with different versions of files, information stored in a variety of formats, and layers of abstractions (e.g. hierarchical, syntactic, semantic) that reduce cohesion in order to accommodate the needs of different software systems (e.g. compilers, build systems, testing, etc.) [24].

The conventional design for dealing with multi-dimensional relationships in IDEs has been to add tabbed or multi-pane user interfaces that individually represent a lens under which we examine code (e.g. a debugger pane for examining run-time state, a version control pane for reconciling different versions of code files, and tabs of editors for operating across multiple code files). However, these interfaces limit the ability to visually describe relationships between different entities. The relationship between different tabs of code is not immediately ascertainable by looking at the arrangement of tabs, and often conveys no information beyond the order in which they were opened [30].

Synectic attempts to expose these interdependent relationships through cards that can be rearranged and grouped according to the specific lens under which the developer is examining the code. For example, a developer attempting to locate and resolve a bug can open a series of syntactically related code files into individual cards. The developer can then create a group of cards that contain code that modifies the code elements involved in the bug, and another group of the non-modifying code cards (just in case they contain information relevant at a later time). Additionally, annotations within Synectic allow the developer to add notes that specifically call out the relevant information found in each card (or group of cards).



```
package edu.osu.ui;

import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c

import com.vaadin.
```

```
package edu.osu.ui.

import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import edu.osu.app.
import edu.osu.back
import edu.osu.back
import edu.osu.back
import edu.osu.ui.M
import edu.osu.ui.
```

In class “MainView.java” , for adding a menu item to the menu(such as "product" menu item), we should check if the current user has access right to the menu item page.

https://docs.spring.io/spring

4.4.2. Query Creation

The query builder mechanism built

Spring jpa repository derives the query from the method name directly. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. You can find more

Figure 3.1: Synectic interface

Synectic provides a canvas-based environment containing spatially arranged cards of relevant information (code, websites, etc.) with annotations that can be linked to individual cards (bottom) or between cards (top).

3.3 Support for Maintaining

Mental models are constructed representations of real world that mirror a working understanding of observed phenomena [13]. Within software development, mental models contain a developers' knowledge and insights into both code and external constraints on the use of that code [19, 41]. Synectic provides direct representation of these mental models through spatially arranged cards of information, and through the annotating them which allow the intrinsic knowledge of developers to be extrinsically archived in their IDE.

Research has shown that maintaining mental models incurs a cognitive cost on developers [19]. This cost affects locating relevant information (see Section 3.1), and sorting through that information to create a mental model that is relevant to the current task (see Section 3.2). After incurring these costs, developers try to reduce or remove these costs from future work by saving the relevant information in code, comments, and documentation that maintains as much of the mental model as possible.

During maintenance tasks, developers often seek to understand (or remember) different aspects of individual entities (known as *information features* in IFT), building understanding of concepts that span multiple entities, and expanding to the larger context of concepts that encompass groups of entities [37, 8]. These relationships are often valuable for a variety of maintenance tasks, but left to each individual developer to explore and build their mental model through direct experience with reading and manipulating the code. The annotation features within Synectic provide a simplified method for capturing and storing this information so that future developers (or the same developer working on future tasks) can quickly recover their mental model; leveraging it to potentially reduce maintenance time and effort.

We further expand upon the use and benefits of annotations within a canvas-based environment through user studies described in the Study Design and Results chapters.

Chapter 4: Study Design

In this chapter, we describe the experimental design of the user study we conducted to compare the annotations in Synectic with the notes functionality in Eclipse.

Our study has two treatments – Synectic (with annotations) and Eclipse (with notes). Participants were randomly assigned to one of the two treatments (between-subjects) and asked to complete four program comprehension task using the assigned IDE. Each task required participants to answer questions about the code. After completing each task, participants were asked to rate their perceived cognitive loads for that task. At the and of all tasks, they were asked to rate the usability of annotations/notes within their assigned IDE.

We describe the components of the user study in details below.

4.1 Participants and Treatments:

Our participants comprised graduate level computer science students recruited through convenience and snowball sampling [10]. These participants represent our target population of newcomers to some project and help our objective of studying how appropriate annotations are for onboarding newcomers in.

22 participants were recruited through university mailing lists; Table 4.1 shows the distribution of our participants (13 participants were men, 8 were women and one participant preferred not to disclose their gender). The median of programming experience was 5 years for both groups (with mean=7.4 years and SD=5.5 years for

Eclipse, and mean=7.0 years and SD=5.0 years for Synectic).

Ptc.[i]	**Gnd.**[ii]	**Exp.**[iii]	**Ptc.**[i]	**Gnd.**[ii]	**Exp.**[iii]
E1	M	15	S1	M	16
E2	M	8	S2	F	3
E3	M	3	S3	M	5
E4	M	2	S4	M	5
E5	F	3	S5	M	3
E6	M	19	S6	M	12
E7	F	8	S7	F	8
E8	F	5	S8	F	2
E9	M	10	S9	P	4
E10	M	5	S10	M	15
E11	F	3	S11	F	4

Table 4.1: Study Participant Demographics

[i] Participant (E for Eclipse, S for Synectic) [ii] Gender (M for Male, F for Female, P for Prefer not to disclose) [iii] Years of software development experience

11 participants were randomly assigned to each treatment. This assignment was balanced based on the participant's programming experience to keep mean experience consistent. All 11 participants from the Eclipse treatment were familiar with Eclipse to some degree, whereas, none of the 11 Synectic participants were familiar with Synectic.

Each study session was time-boxed to two hours. We obtained participant's consent and walked them through their assigned IDE, the task project, and procedures. Following these, participants were asked to complete a warm-up task to get used to the study protocol. Participants were asked to think aloud and we recorded the screen and audio of the them during the tasks. After the tasks, participants completed a usability survey related to the annotation/notes feature. At the end of the study, participants were offered US$20 as compensation for their time.

4.2 Project and Tasks:

The tasks were based on a Java project ($LOC \approx 5000$) designed for a bakery to manage their orders and customers. It includes functions to keep inventory of products, customers, employees, and a visual dashboard to summarize all transactions [1]. The project was implemented using Vaadin framework. None of our participants was familiar with this framework, making this project a good choice.

A senior developer from the project provided the necessary information aimed to help newcomers understand the code base. The information was presented in two formats: through the annotations in Synectic, and as a word document to be loaded in Eclipse; both contained the same information.

Participants were given four comprehension tasks. These tasks were designed to be representative of common problems that newcomers experience when onboarding [1]. Each task comprised two parts. Part A involved locating the element (method, class etc.) in the codebase related to a specific feature. Finding the initial focus point of a programming task is a well known problem discussed by many researchers[cite]. Part B included an in-depth question regarding the Part A feature, such as, how a specific aspect of the feature has been implemented, or what needs to be modified to change an aspect of the feature. see Table 4.2 for the specific prompts and questions given to participants in the study prompt.

After each task, participants reported their perceived cognitive load for the task

[1]`https://vaadin.com/start/latest/full-stack-spring`

Task	Part	Question
1	A	Name the class(es) and method(s) in which we put the ”product” menu item in the list of system menus.
	B	To add a menu item in the body of ”configure” method, an instance of ”AppLayoutMenuItem” has been created. Which parameters are needed to create an ”AppLayoutMenuItem” for the “product” menu item? Explain what each parameter means.
2	A	In which class(es) have “product” validations (e.g. not blank, acceptable format for a field,...) been added?
	B	How did we limit the maximum price of a product?How does the system limit the maximum price of a product?
3	A	In which class(es) do we add the code to get the user access to the “Product” pages?
	B	We want only the user with role “Manager” be able to have access to the “product” page. What changes would you apply?
4	A	Which class(es) are responsible for implementing a ”product”-related search?
	B	We want to be able to search the products by product Name and Price. What changes would you apply?

Table 4.2: Tasks

List of the code comprehension tasks. Each tasks comprised of two parts (A and B) by answering "how mentally demanding was the task?" (using a balanced Likert-scale response, where 1 is *very low* and 7 is *very high*) [25]. After completing all four tasks, participants provided overall usability ratings for the annotations/notes features of the assigned IDE by completing a questionnaire based on the System Usability Scale [4]. Table 4.3 shows the six questions from our usability survey.

SUS	**Question**
SUS.1	I think that I would use these on-boarding notes/annotations when working on programming tasks.
SUS.2	I would imagine that most developers would like to use these on-boarding notes/annotations when programming.
SUS.3	I found using these on-boarding notes/annotations unnecessarily time-consuming.
SUS.4	I found these on-boarding notes/annotations helpful when completing the tasks.
SUS.5	I found these on-boarding notes/annotations very cumbersome/awkward to use.
SUS.6	I felt confident about completing my tasks when using these on-boarding notes/annotations.

Table 4.3: System Usability Scale (SUS) questions

4.3 Measurements and Constructs:

To answer our research questions, we measured *time* and evaluated the *accuracy* of responses for each prompt/question. We provide definitions of all relevant constructs used in our results and discussions below:

Time. The time taken to answer each question was the duration between the time the participant switched to the IDE after reading the question and the time they hit the “next” button on the task form to proceed to the next question. Time spent in each task is the sum of the time spent in the two parts of the tasks. The overall time for each participant is the average time taken in all four tasks.

Accuracy ($\mathcal{A}$): Accuracy of a response is dependent on the completeness and correctness of individual elements within that response. We use the balanced Sørensen–Dice coefficient (F_1-score) [39] to calculate accuracy:

$$\mathcal{A} = \frac{2\mathcal{TP}}{2\mathcal{TP} + \mathcal{FP} + \mathcal{FN}}$$

Where *True Positive* (TP) is the number of elements (e.g. class, method, etc.) correctly identified in an individual response, *False Positive* (FP) is the number of elements incorrectly identified in the response, and *False Negative* (FN) is the number of elements missing from the response. For a concrete example, if a prompt asks for the names of relevant classes for a specific feature and that feature has three relevant classes. Then a response that correctly names two relevant classes (TP) and failed to mention the third one (FN), but includes three other irrelevant classes (FP) in the response, would result in an accuracy $\mathcal{A} = \frac{2\times2}{2\times2+3+1} = 0.5$ (or 50%).

Since each task is comprised of two parts, we calculate the accuracy of a task such that $\mathcal{A}_T = \frac{\mathcal{A}_{T_A}+\mathcal{A}_{T_B}}{2}$ (where $\mathcal{A}_{T_A}$ represents accuracy from Part A, and $\mathcal{A}_{T_B}$ represents accuracy from Part B). The overall accuracy for each participant is the average accuracy in all four tasks.

Cognitive load. The perceived cognitive load was reported by participants after each task using a seven-point Likert scale (where 1 is *very-low*, and 7 is *very-high*). The overall cognitive load for a participant is the average over the four tasks.

Usability. The perceived usability of the onboarding document/annotations were reported by participants at the end of the study using a seven-point Likert scale (where 1 is *strongly-agree*, and 7 is *strongly-disagree*) and standardized System Usability Scale (SUS) prompts [4]. Table 4.3 shows the questions. SUS.3 and SUS.5 were negatively-worded prompts, which required inverting the Likert scale (where 1 is *strongly-disagree*, and 7 is *strongly-agree*) to maintain consistency with positively-worded prompts. The perceived usability score for each participant was the sum of the scores across all question. The overall usability score of each participant converted to a 0-100 scale.

Each participant was given four code comprehension tasks. This means we obtained four measurements for each participants. These multiple measurements are not independent data points and generally assumed to be correlated. One approach to address this issue is calculating a summary measure for each participants which is often the mean value of the measurements [26]. Therefore, we defined the overall time, accuracy, and cognitive load for each participant as the summary measure for each participant.

Chapter 5: Results

In this chapter, we present the results of the our analysis of the data obtained from 22 participants of the user study.

5.1 RQ1: Accuracy

RQ1: How do annotations affect the accuracy of a newcomer's comprehension of the code during onboarding tasks?

We measured *accuracy* of responses for each task using the balanced Sørensen–Dice coefficient (F_1-score) [39]. The overall accuracy for each participant is the average accuracy in all four tasks.

Participants using Syntectic had a higher overall accuracy score, as shown in Figure 5.1. A Wilcoxon Rank-Sum test indicated that accuracy was higher for Synectic participants (Median: 0.75) than for Eclipse participants (Median: 0.45), (W = 9.5, p-value < 0.001, two-sided Wilcoxon rank-sum test). The analysis also showed a large effect size (Cliff's Delta $\delta = 0.84$). Note that in the case of Task-3 the spread for Synectic participants is much lower than Eclipse, this was because finding the information for Part-B of the task was easily done because of the annotations. Section 6.2 discusses this further.

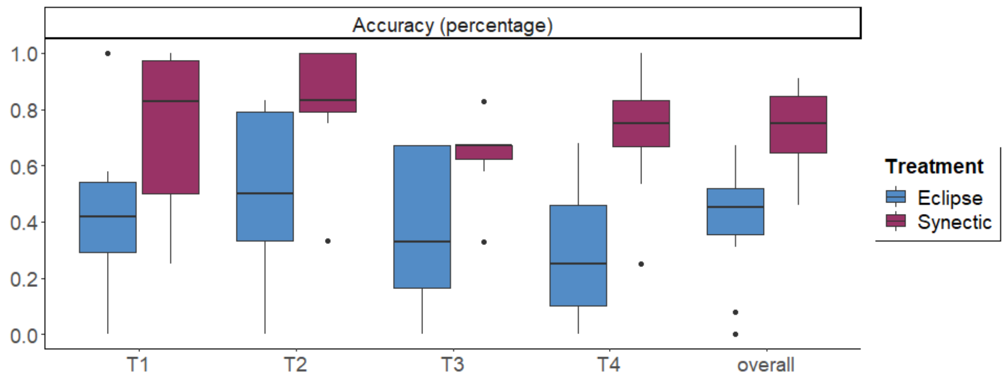


Figure 5.1: Boxplots of accuracy scores

The figure shows boxplots for accuracy scores for each task T1–T4 (left bars), as well as for overall accuracy scores(the right-most pair of bars) grouped by treatment (Synectic or Eclipse).

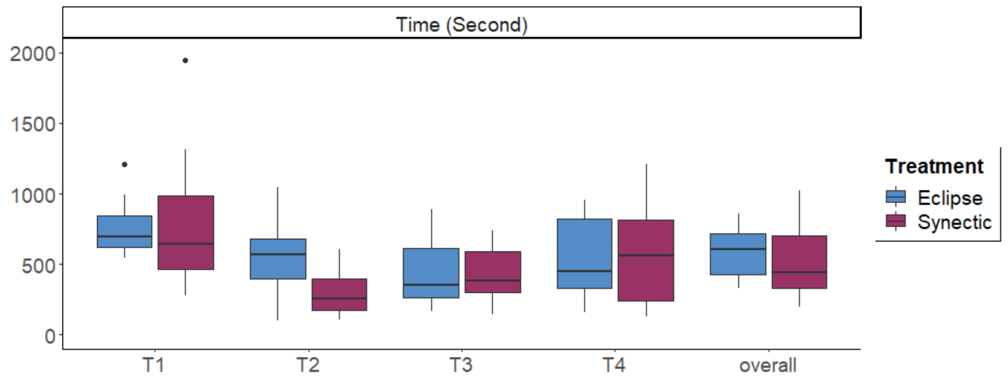


Figure 5.2: Boxplots of time to completion

The figure shows the boxplots for time spent for each task T1–T4 (left bars), as well as for overall time (the right-most pair of bars) grouped by treatment (Synectic or Eclipse).

5.2 RQ2: Time

RQ2: Do annotations affect the time of code comprehension during newcomer's onboarding tasks?

We measured the time taken to complete each task. The overall time for each participant is the average of time over all four tasks.

The median of overall time taken by Synectic group (440 seconds) was smaller than that of Eclipse group (607 seconds), as shown in Figure 5.2. However, the difference failed to achieve statistical significance (W = 74, p-value = 0.40, two-sided Wilcoxon rank-sum test).

5.3 RQ3: Cognitive Load

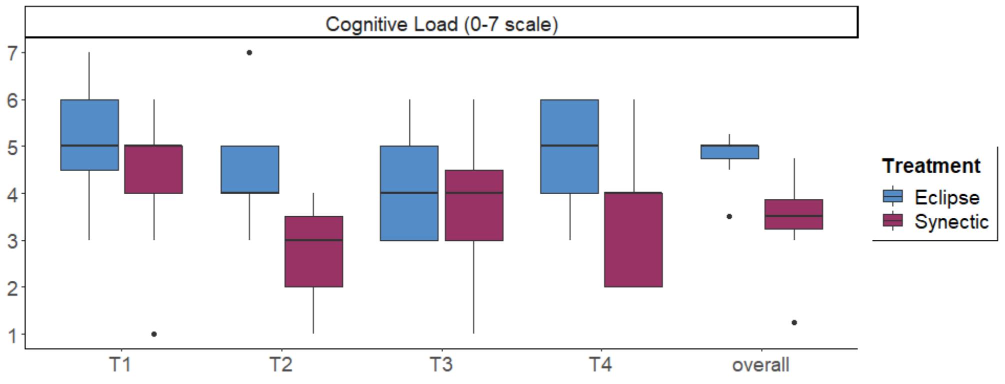


Figure 5.3: Boxplots of cognitive load

The figure shows the boxplots for reported cognitive load for each task T1–T4 (left bars), as well as for overall cognitive load (the right-most pair of bars) grouped by treatment (Synectic or Eclipse).

RQ3: Do annotations affect newcomer's cognitive load during onboarding tasks that require code comprehension?

The perceived cognitive load was reported by participants after each task using a seven-point Likert scale. The overall cognitive load for a participant is the average over the four tasks.

Synectic participants reported less overall cognitive load than Eclipse participants as shown in Figure 5.3. A Wilcoxon rank-sum test indicated that Synectic participants reported lower cognitive load cognitive load (Median: 3.5) than the Eclipse participants (Median: 5), (W = 107, p-value = 0.002, two-sided Wilcoxon rank-sum test). The analysis also showed a large effect size (Cliff's Delta $\delta = 0.77$).

5.4 RQ4: Usability

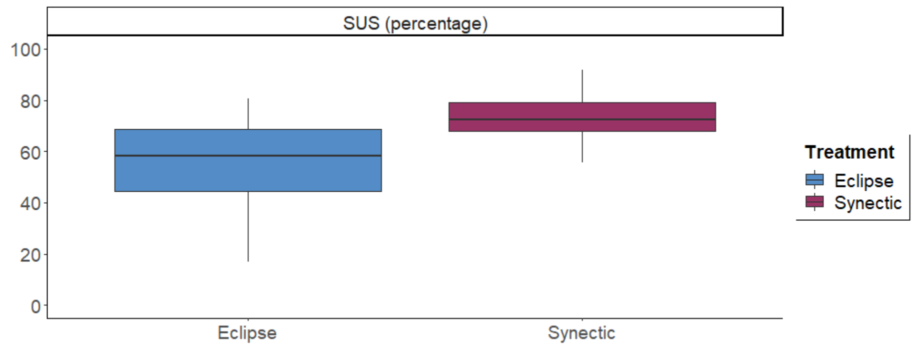


Figure 5.4: Boxplot of system usability scale

Boxplot of System Usability Scale (SUS) percentages grouped by treatment(Synectic, Eclipse)

RQ4: Do users perceive annotations to be useful for code comprehension?

The perceived usability of the onboarding annotations/notes were reported by participants at the end of the study using a seven-point Likert scale. The perceived usability score for each participant was the sum of the scores across all question converted to a 0-100. scale.

Participants rated Synectic as more usable than Eclipse. The average of usability scores reported by Synectic participants (73.74) was higher than the average of usability score reported by Eclipse participants (53.79). Figure 5.4 shows boxplot of the usability score reported by participants. Also, a Wilcoxon rank-sum test showed that the groups differ in usability score (p-value < 0.0123, two-sided Wilcoxon rank-sum test) with the usability of Synectic (Median: 72.22) was higher than Eclipse (Median: 52.78). The difference between the groups was large (Cliff's Delta $\delta = 0.64$).

Chapter 6: Discussion

We observed participants in the Synectic group answered questions with higher accuracy than participants in the Eclipse group. The Synectic group also incurred less cognitive load when answering the study task questions.

In this chapter, to understand the observed differences between the Synectic group and the Eclipse group, we will first look at how and where participants struggled with navigation and foraging for information from the perspective of Information Foraging Theory (IFT) [33]. Next, we examine code comprehension using the four categories of questions that developers ask during change tasks in Silito et al. [37]. Combining these models allows us to compare Eclipse and Synectic support for locating relevant information and comprehending the code necessary to undertake code maintenance tasks.

6.1 Using IFT to understand navigation and foraging behavior

Information Foraging Theory (IFT) describes how people seek information [32]. IFT is a suitable theory for explaining and predicting developers behaviors, and can be applied to tool design [8, 29, 20].

As mentioned in the study design (Chapter 4), participants were presented with an onboarding document displayed as a set of annotations within Synectic, and as a text document loaded within Eclipse; the information provided was identical for both groups. In IFT terminology, each paragraph of the onboarding document provided to the Eclipse group is an information *patch*, and the full document is a collection of *patches* arranged in a sequential topology. For the Synectic group, each annotation contains an information *patch*, but is arranged to be in proximity to the relevant code discussed within the *patch*.

6.1.1 Foraging in the document

Participants in the Eclipse group had to forage for the right *prey* by navigating across *patches* arranged sequentially in the onboarding document. Eclipse participants followed two strategies to avoid having to read the entire document; visually *skimming* the document and using keyword *searching*. Although successful some of the time, these strategies also posed problems for participants.

6.1.1.1 Skimming can backfire:

All Eclipse participants, at least once, *skimmed* through the *patches* in an attempt to mitigate the full costs of foraging across the entire document. However, skimming still involved some cost as participants had to navigate through irrelevant *patches* (i.e. paragraphs not relevant to the task at hand). Moreover, skimming backfired when participants overlooked the *patch* containing the *prey*.

For example, E11 overlooked the information she needed while skimming the onboarding document to complete Task-1 (see Table 4.2 for task descriptions). When navigating the document, E11 tried to identify the class and method used to add an item to the system menu. *"I'm looking in the document to find something related to the product"*; she skimmed the document from bottom to top, passing by the relevant *patch* several times. After 1.5 minutes of scrolling within the onboarding document (i.e. foraging), she found the correct *patch* and began thoroughly reading through the text within the *patch*. E11 was not the only participant who struggled with skimming. Other participants (e.g., E10 during Task-3) were unable to find the relevant *patch* even after multiple skimming attempts, and eventually gave up on completing portions of the study tasks.

6.1.1.2 Searching by keyword can make or break:

Some participants used keyword search to reduce the number of potential *patches*. However, this strategy was not always successful. Some participants used incorrect keywords (or synonyms of relevant keywords), which returned no *patches* or irrelevant *patches*.

During Task-4, which asks participants to locate and prepare to update the `product` page with additional search functionality, E7 used the keyword "search" for searching within the onboarding document *"I'm just keep searching this document to see if there is anything to do with `search`"*. The search results included documentation describing the implementation of search functionality in the `Storefront` page, but nothing in regards to the `product` page. Other participants (E4 and E5) also faced the same problem and became stuck. In these cases, the keyword mismatch led participants astray, and arose because the author of the onboarding document had used the word "find" instead of "search" in the *patches* related to the `product` page. This variation in terminology is a common problem for newcomers. Furnas et al. [9] mention that people use a surprisingly large variety of words to refer to the same thing, and new users often use the wrong words. This disparity causes newcomers to fail more often in achieving the actions or information they want, which is known as the "vocabulary problem".

Among Synectic participants, information foraging was less of a challenge. *Patches* were contained within annotations that were linked to cards containing directly related code. This interweaving of code and information *patches* takes advantage of the Gestalt laws of perceptual organization, specifically proximity and continuity, which allowed participants to understand information is spatially proximate to related information and following the link-directed pathways to find additional relevant information [34, 43]. Synectic participants were able to visually skim groups of related cards to quickly narrow in on the information they needed. And once located, participants were able to use annotations to connect relevant information to code, which reduced their overall foraging costs.

6.1.2 Foraging across code and document

Foraging in the onboarding document was only the start for Eclipse participants. After finding the right information in the document *patch*, participants had to refer to the appropriate code *patches* in order to comprehend the code. Navigating between the document and code *patches* imposed additional costs on foraging.

6.1.2.1 Hierarchy hides prey.

Code is hierarchical in nature; a method (*patch*) is contained inside a class (*patch*), which is further contained in packages or components (*patch*). A relevant piece of code required for a task could be at any level of this hierarchy. However, the Eclipse user interface does not provide direct links between code and documentation (except in the case of Javadocs and similar code comment standards that allow URLs for linking sections of code to outside documentation). For example, Eclipse participants relied on the Project Explorer view in order to navigate to potentially relevant code *patches* during our study.

The Project Explorer view itself is a *patch*, providing cues about existing classes and packages as well as the hierarchical structure of code (e.g. the structure of classes contained in packages). Well-established projects can have many hierarchical levels, and require developers to forage for *prey* buried deep within the project hierarchy. The hierarchy of the project used in our study was 6-levels deep, which renders foraging strategies involving systematic searches as non-trivial and cost-prohibitive.

For example, when working on Task-1 (see Table 4.2 for task descriptions), E6 believed that the relevant piece of code (*prey*) should be in a *patch* that discusses the user interface; *"I imagine it should be a UI kinda area."* However, the *prey* (`Mainview.java`) was six levels deeper than the file that E6 was currently examining. He had to guess which path to further drill down into, and at some point E6 complained, *"why is this folder structure so deep? It's horrible!"* This process of drilling down was made more difficult by the lack of cues indicating which paths were more likely to yield the desired prey. When E6 became frustrated, he switched strategy and began expanding each level of the tree hierarchy, saying, *"let's explore them all."* Once all paths were expanded, he skimmed through the file names in order to locate the right *patch*.

6.1.2.2 Prey scattered across hierarchy.

Foraging cost was higher when *prey* was scattered across hierarchy (e.g. relevant classes were spread across file structure and packages). Participants had to navigate across different levels of the hierarchical structure several times for each *prey*. This imposed high cognitive loads, reduced their focus, and lengthened the time to complete a task.

For example, in Task-2, E11 used the Project Explorer to locate two classes mentioned in the document–`ProductsView.java` and `product.java`. She first found `ProductsView.java` in the Project Explorer and read the code *patch* contained within it. She returned to the Project Explorer in order to locate the second class (`product.java`) as well. She struggled with finding the second *prey* for more than 2.5 minutes:

> *"I don't know where this* ***`product.java`*** *is. The only thing that I can do is to go through every package and search."*

In total, she spent 4.5 minutes searching through the Project Explorer to find both class files. During this search, she lost track of her original goal, *"OK, what was the question?"* After rereading the task instructions, she had to reorient herself within the Eclipse user interface before restarting her foraging. This process was repeated for both class files.

In general, context switches between code and document patches incurs a cognitive load [42]. Within our study, this cost increased as participants attempted to answer task questions that were increasingly complex. First, when foraging for relevant patches by navigating through the file hierarchy, participants had to manage the context switches between examining a potentially relevant file and searching for new potentially relevant files. Second, when attempting to correlate the information found in particular files, participants had to switch between interfaces and information formats (reading source code and reading text documentation have both similar and different constraints for human comprehension and learning [35]). And finally, when a sufficient number of relevant patches have been located, participants had to switch between the files in the process of building understanding and developing a mental model of the code.

The costs of these context switches impede comprehension and put a high cognitive load on participants who may become disappointed and discourage if the cost of extracting or locating information is higher than the expected costs[30]. Several participants in our study reported becoming disappointed (and even abandoning particular tasks) when the high cost of locating prey across boundaries in user interfaces and information formats exceeded their expectations.

In Synectic, however, the documentation *patches* were placed adjacent to the relevant code *patches*. And visually joined through the use of annotations and links. This combined interface creates a unitary source of information that requires less attention splitting, and thus leads to substantially enhanced performance [42]. For participants in our Synectic group, this reduced the cost of foraging across *patches* and ultimately allowed for faster comprehension and reduced cognitive load.

6.2 Using Sillito's four stages of questions to understand code comprehension behavior

To understand why Synectic participants outperformed Eclipse participants in code comprehension tasks, we use the four categories of code comprehension questions described by Sillito et al. [37] to evaluate affordances that effect developer comprehension. We find that annotations in Synectic provide additional affordances that developers can use to more quickly answer code comprehension questions in each category.

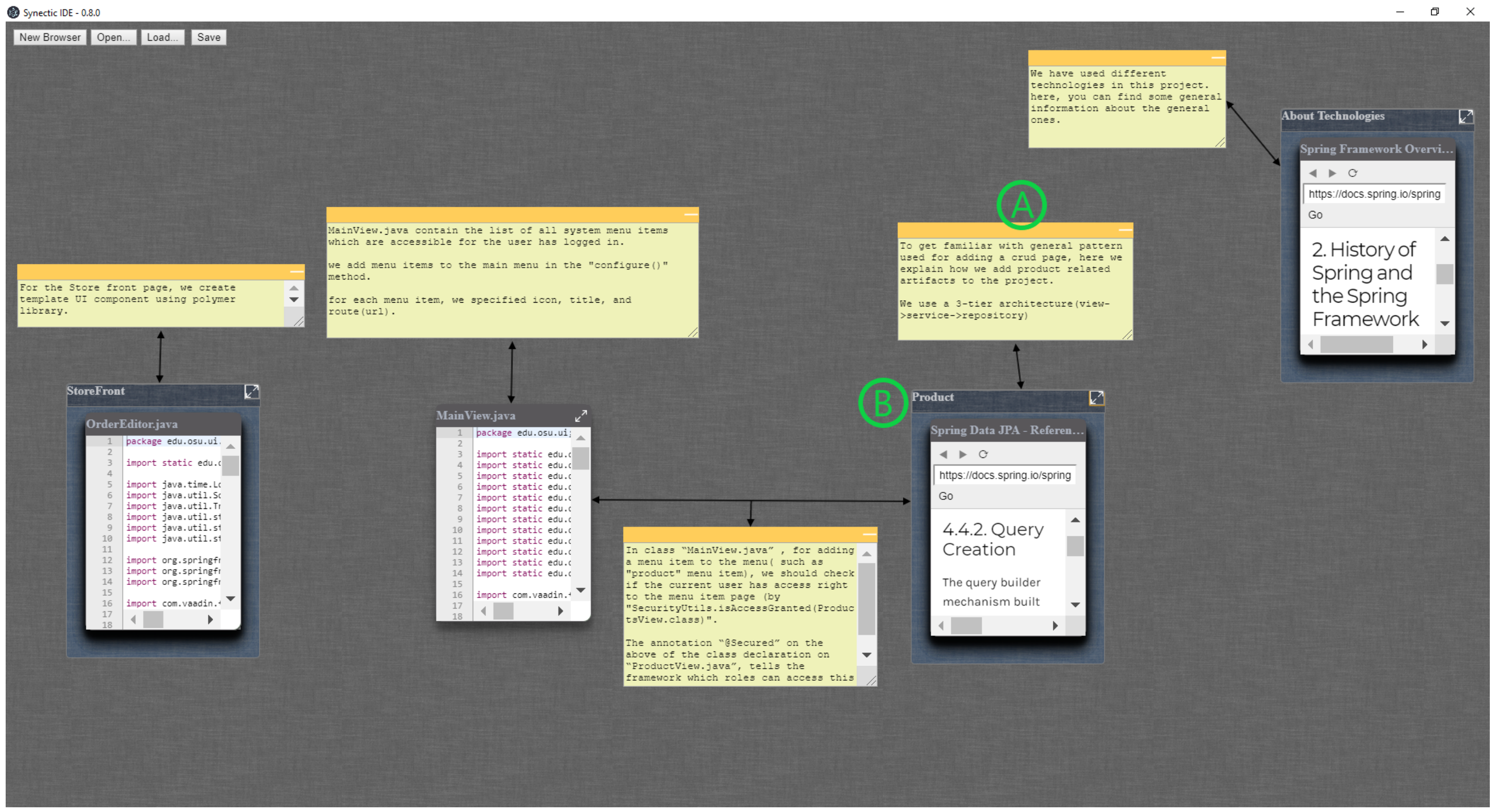
According to Sillito et al. [37], developers ask comprehension questions in four categories during code change tasks: (1) questions about finding points in the code that were relevant to the task, (2) questions that explore relationships of an entity believed to be related to the task, (3) questions that build an understanding of concepts that involve multiple relationships and entities, and (4) questions that build an understanding over groups of related entities (contextual questions).

Exemplifying the full spectrum of comprehension questions, we examine participant S5 during Task-4 through each of the four categories. Task-4 required participants to develop an understanding of the search functionality implemented in the target Java project (see Table 4.2); understanding the code enough to make changes to the search functionality so that search can use the `product name` and `product price` fields in addition to the previous implemented search fields.

6.2.1 Finding an initial focus point

. Developers start their task by finding points/entities that are relevant to the task [37]. In Synectic, starting from the main canvas, annotations can help developers narrow down the search and direct them to the group(of cards) related to the task, and from there to the related cards inside the group.

For example, S5 started from the main canvas (Figur 6.1 and looked at the annotations to see which one was related to the task i.e. 'product-related search':


Synectic IDE - 0.8.0
New Browser
Open...
Load...
Save
StoreFront
OrderEditor.java
MainView.java
Product
Spring Data JPA - Referen...
About Technologies
Spring Framework Overvi...
https://docs.spring.io/spring
Go
2. History of Spring and the Spring Framework
4.4.2. Query Creation
The query builder mechanism built
A
B

For the Store front page, we create template UI component using polymer library.

MainView.java contain the list of all system menu items which are accessible for the user has logged in.

we add menu items to the main menu in the "configure()" method.

for each menu item, we specified icon, title, and route(url).

To get familiar with general pattern used for adding a crud page, here we explain how we add product related artifacts to the project.

We use a 3-tier architecture(view->service->repository)

We have used different technologies in this project. here, you can find some general information about the general ones.

In class "MainView.java" , for adding a menu item to the menu(such as "product" menu item), we should check if the current user has access right to the menu item page (by "SecurityUtils.isAccessGranted(ProductsView.class)".

The annotation "@Secured" on the above of the class declaration on "ProductView.java", tells the framework which roles can access this

```
package edu.osu.ui.

import static edu.c

import java.time.Lc
import java.util.Sc
import java.util.Tr
import java.util.st
import java.util.st
import java.util.st

import org.springfr
import org.springfr
import org.springfr

import com.vaadin.
```

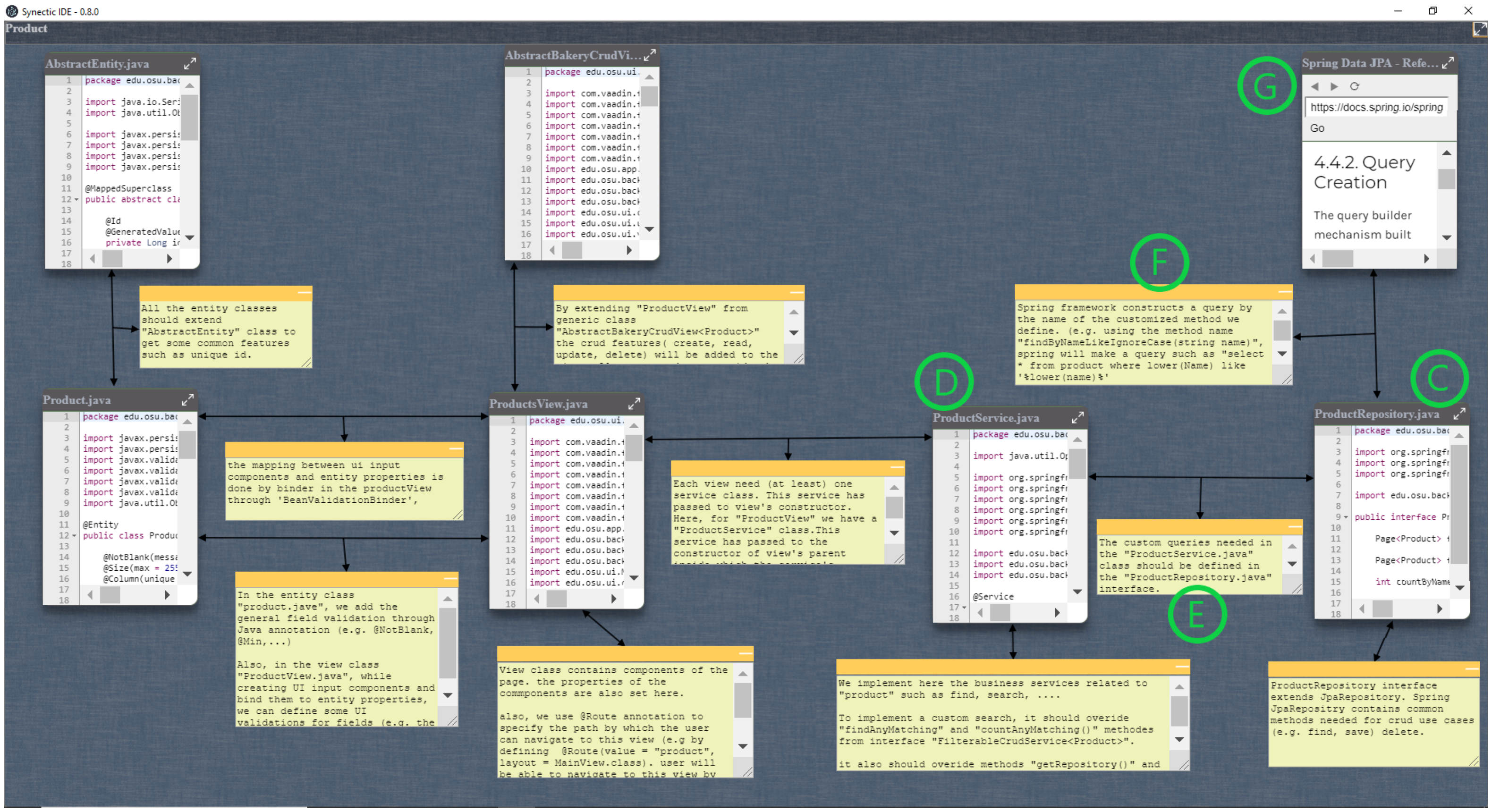
```
package edu.osu.ui;

import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c
import static edu.c

import com.vaadin.
```

Figure 6.1: Synectic IDE, project canvas

A snapshot of the the main canvas opened in the Synectic IDE during the user study. See section 6.2.



```
package edu.osu.bac

import java.io.Seri
import java.util.Ob

import javax.persis
import javax.persis
import javax.persis
import javax.persis

@MappedSuperclass
public abstract cla

    @Id
    @GeneratedValue
    private Long id
```

```
package edu.osu.ui.

import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import edu.osu.app.
import edu.osu.back
import edu.osu.back
import edu.osu.back
import edu.osu.ui.c
import edu.osu.ui.u
import edu.osu.ui.v
```

```
package edu.osu.bac

import javax.persis
import javax.persis
import javax.valida
import javax.valida
import javax.valida
import javax.valida
import java.util.Ob

@Entity
public class Produc

    @NotBlank(messa
    @Size(max = 255
    @Column(unique
```

```
package edu.osu.ui.

import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import com.vaadin.f
import edu.osu.app.
import edu.osu.back
import edu.osu.back
import edu.osu.back
import edu.osu.ui.M
import edu.osu.ui.c
```

```
package edu.osu.bac

import java.util.Op

import org.springfr
import org.springfr
import org.springfr
import org.springfr
import org.springfr
import org.springfr

import edu.osu.back
import edu.osu.back
import edu.osu.back

@Service
```

```
package edu.osu.bac

import org.springfr
import org.springfr
import org.springfr

import edu.osu.back

public interface Pr

    Page<Product> f

    Page<Product> f

    int countByName
```

4.4.2. Query Creation

The query builder mechanism built

All the entity classes should extend "AbstractEntity" class to get some common features such as unique id.

By extending "ProductView" from generic class "AbstractBakeryCrudView<Product>" the crud features(create, read, update, delete) will be added to the

Spring framework constructs a query by the name of the customized method we define. (e.g. using the method name "findByNameLikeIgnoreCase(string name)", spring will make a query such as "select * from product where lower(Name) like '%lower(name)%'

the mapping between ui input components and entity properties is done by binder in the productView through 'BeanValidationBinder',

Each view need (at least) one service class. This service has passed to view's constructor. Here, for "ProductView" we have a "ProductService" class.This service has passed to the constructor of view's parent

The custom queries needed in the "ProductService.java" class should be defined in the "ProductRepository.java" interface.

In the entity class "product.jave", we add the general field validation through Java annotation (e.g. @NotBlank, @Min,...)

Also, in the view class "ProductView.java", while creating UI input components and bind them to entity properties, we can define some UI validations for fields (e.g. the

View class contains components of the page. the properties of the commponents are also set here.

also, we use @Route annotation to specify the path by which the user can navigate to this view (e.g by defining @Route(value = "product", layout = MainView.class). user will be able to navigate to this view by

We implement here the business services related to "product" such as find, search, ...

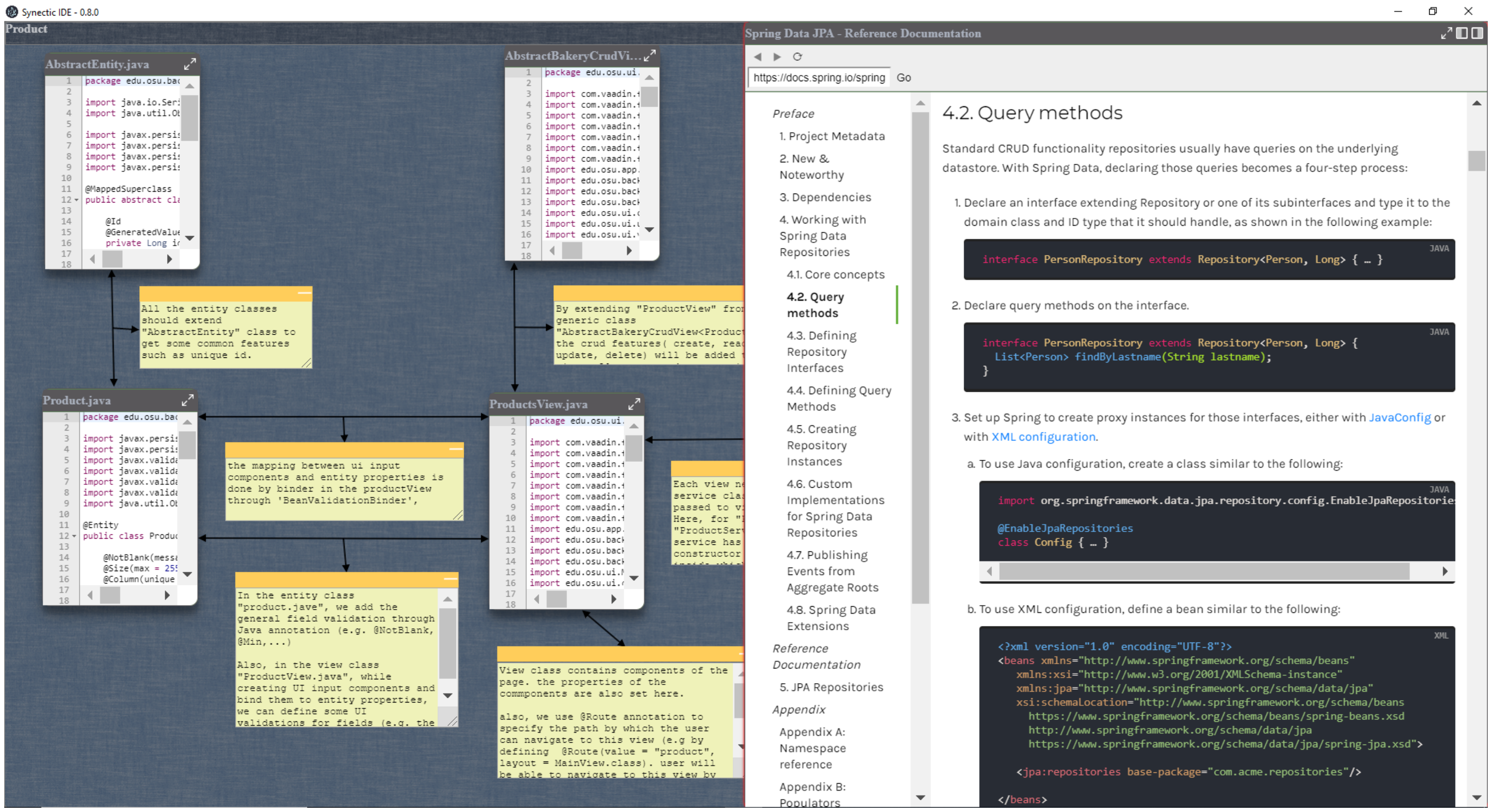
To implement a custom search, it should overide "findAnyMatching" and "countAnyMatching()" methodes from interface "FilterableCrudService<Product>".

it also should overide methods "getRepository()" and

ProductRepository interface extends JpaRepository. Spring JpaRepositry contains common methods needed for crud use cases (e.g. find, save) delete.

Figure 6.2: Synectic IDE, Product group

A snapshot of the the Product group during the user study. See section 6.2.



```
package edu.osu.ba

import java.io.Seri
import java.util.Ob

import javax.persis
import javax.persis
import javax.persis
import javax.persis

@MappedSuperclass
public abstract cla

    @Id
    @GeneratedValue
    private Long id
```

```
package edu.osu.ui.

import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import edu.osu.app.
import edu.osu.back
import edu.osu.back
import edu.osu.back
import edu.osu.ui.
import edu.osu.ui.
import edu.osu.ui.
```

All the entity classes should extend "AbstractEntity" class to get some common features such as unique id.

By extending "ProductView" from generic class "AbstractBakeryCrudView<Product the crud features(create, read update, delete) will be added

```
package edu.osu.ba

import javax.persis
import javax.persis
import javax.valida
import javax.valida
import javax.valida
import javax.valida
import java.util.Ob

@Entity
public class Produc

    @NotBlank(messa
    @Size(max = 255
    @Column(unique
```

the mapping between ui input components and entity properties is done by binder in the productView through 'BeanValidationBinder',

```
package edu.osu.ui.

import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import com.vaadin.
import edu.osu.app.
import edu.osu.back
import edu.osu.back
import edu.osu.back
import edu.osu.ui.M
import edu.osu.ui.
```

Each view ne service cla passed to vi Here, for " "ProductSer service has constructor

In the entity class "product.jave", we add the general field validation through Java annotation (e.g. @NotBlank, @Min,...)

Also, in the view class "ProductView.java", while creating UI input components and bind them to entity properties, we can define some UI validations for fields (e.g. the

View class contains components of the page. the properties of the commponents are also set here.

also, we use @Route annotation to specify the path by which the user can navigate to this view (e.g by defining @Route(value = "product", layout = MainView.class). user will be able to navigate to this view by

Preface
1. Project Metadata
2. New & Noteworthy
3. Dependencies
4. Working with Spring Data Repositories
4.1. Core concepts
4.2. Query methods
4.3. Defining Repository Interfaces
4.4. Defining Query Methods
4.5. Creating Repository Instances
4.6. Custom Implementations for Spring Data Repositories
4.7. Publishing Events from Aggregate Roots
4.8. Spring Data Extensions
Reference Documentation
5. JPA Repositories
Appendix
Appendix A: Namespace reference
Appendix B: Populators

4.2. Query methods

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

```java
interface PersonRepository extends Repository<Person, Long> { … }
```

2. Declare query methods on the interface.

```java
interface PersonRepository extends Repository<Person, Long> {
  List<Person> findByLastname(String lastname);
}
```

3. Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.

a. To use Java configuration, create a class similar to the following:

```java
import org.springframework.data.jpa.repository.config.EnableJpaRepositorie

@EnableJpaRepositories
class Config { … }
```

b. To use XML configuration, define a bean similar to the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     https://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     https://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

Figure 6.3: Synectic IDE, Product group with an opened web-browser card.

A snapshot of Product group with an opened web-browser card during the user study. See section 6.2.

"I'm in the main canvas and search for product-related search. I'm reading the notes". He found an annotation (Figur 6.1 A) connected to the group of cards titled `product` (Figur 6.1 B). The annotation explained 'how to add a CRUD page using product page as an example'. He decided to expand this group; *"I think I should go to this group"*. S5 found that the `product` group contained several cards and annotations (Figur 6.2). He was hunting for the annotation which *"gave [him] a clue to product search"*. After skimming through the annotations for two minutes, he decided to open the code card `ProductRepository.java` (Figur 6.2 C) and read the methods within this class.

6.2.2 Building on those points

. From the initial focus point, developers begin asking questions that explore relationships to expand entities believed to be related to the task [37]. Synectic provided a set of connections between relevant cards (or annotations), which allowed participants to explore beyond their initial focus point (i.e. their initial card).

Once S5 had located `ProductRepository.java` as his initial focus point, he said, *"It contains a* ***`find`*** *method which seems to be interesting"*. He minimized the card and began to search for the location in which this 'find' class/method is called *"there should be some implementation related to the* ***`ProductRepository.java`****"*. Using the annotation link between the `ProductRepository` card and the `ProductService.java` (Figure 6.2 D) card, he decided to open the `ProductService.java` card.

6.2.3 Understanding concepts between related entities

. Using the relevant entities, developers ask questions to build an understanding of concepts in the code that involve multiple relationships and entities [37]. In Synectic, annotations linked to the entities (i.e cards/groups) provide expert description of the concept and relationships between them. This documentation helps to build an understanding of the concepts spanning related cards and groups.

To understand how the two classes `ProductService.java` and `ProductRepository.java` interact with each other to implement a search, S5 read the annotation connecting these two classes (Figure 6.2 E). He then opened the two cards side-by-side to examine them simultaneously, and noticed that the find method in `ProductRepository` was being called in `ProductService`. This helped him understand the relationship between these two classes.

6.2.4 Questions over groups of related entities

. Finally, developers ask questions regarding related groups of entities, and the relationships between those groups. The information within Synectic's annotations, and the links between cards, help to relate different concepts across the system to build an overall understanding of the larger context.

Within our study, we asked participants to develop an understand of the overall software in order to add additional functionality to the search feature. In order to accomplish this task, participants need to know how search has been implemented, and combine this understanding with how to make custom queries by method name (using the Springs' JPARepository API).

For S5, he checked the annotations to figure out how to adjust the implementation to support search by both the `name` and `price` of the product. The annotation connecting the `ProductRepository` card to a web-browser card explained how to make custom query by method name (using `Springs' JPARepository`)(Figur 6.2 F). He read the annotation and opened the web-browser card (Figur 6.2 G) which lead to the API documentation related to custom queries (Figur 6.3) *"I found the documention for it"*. The documentation explained how to create a custom query by defining a method name. S5 was able to accomplish the task, even though he was unfamiliar with the technology *"Oh! I just should define a query method. I haven't used this before, but I think this example here shows how to do it."*. He combined the the knowledge of creating custom queries with his previous knowledge of how and where the search method works (`ProductRepository` and `ProductService` classes).

In summary, developers must gather relevant information at multiple levels in order to constructively work with large codebases. Typical IDEs (e.g. Eclipse) provide limited capabilities for expressing relationships at the conceptual levels (phases 3 and 4 from Sillito et al. [37]), when context becomes important. Synectic provides annotations and groupings that point toward good initial focus points, groupings that highlight important related entities, and annotated links that explain the relationships between these entities and even the larger context within a software project. With these features, Synectic facilitates all four phases of code comprehension questions described in Sillito et al. [37].

6.3 Threats to Validity

Our user study has several limitations inherent to laboratory studies of programmers. Our participants were graduate students and may not be representative of professional developers. However, all participants had at least two years of software development experience, and would likely be considered newcomers to any software projects they contribute to now or in the near future. The task and code bases were related to a single Java-based framework project, which may not be representative of large software projects. However, our participants' tasks were examined by a senior developer on the project to verify that they represent onboarding tasks that do occur in real-world development scenarios. Additionally, we did not ask participants to implement new features or change the code directly, which is actually a common practice for newcomers that are just beginning to learn about a project [38].

As with any empirical research involving participant observation, responses could have been affected by the Hawthorne effect [23]. To mitigate biases in participant responses, we were careful not to disclose the comparisons we were making during the study. Additionally, participants might have previous experience using Eclipse, but none had previously used Synectic. Participants could have been aware of advanced features in Eclipse that are not present in Synectic, which would reduce some of the navigational costs. However, even with this potential disadvantage, we observed participants using Synectic generally performed better than participants using Eclipse.

Chapter 7: Conclusion

In this Thesis, we have presented annotations in Synectic, a canvas-based IDE with spatially-oriented interface which allows relevant information to be arranged and group according to the user needs, as well as externalizes relationships through annotations and links. Our aim was to provide developers with support for foraging information, code comprehension, and code maintenance. To validate annotations in Synectic, we conducted a user study comparing newcomer task support for foraging and comprehension with a traditional IDE (Eclipse) and our canvas-based IDE. The results of our user study show promising evidence that Synectic fulfills these goals: We observed participants using annotations in Synectic were able to answer navigation and comprehension questions with significantly higher accuracy (RQ1) and efficiency (RQ2) than those using Eclipse. Also, participants using Synectic reported less cognitive load (RQ3) and rated Synectic as more usable (RQ4) than Eclipse on average. However, we was not able to find a statistically significant differences in the time (RQ2).

These findings also suggest promising directions for future research. One open question is how developers might use annotations in Synectic to externalize their own mental model, and how revisiting their own annotations might impact future performance during software development tasks.

Bibliography

[1] Sogol Balali, Igor Steinmacher, Umayal Annamalai, Anita Sarma, and Marco Aurelio Gerosa. Newcomers' barriers... is that all? an analysis of mentors' and newcomers' barriers in oss projects. *Computer Supported Cooperative Work (CSCW)*, 27(3-6):679–714, 2018.

[2] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512, 2010.

[3] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Frequently asked questions in bug reports. Technical report, University of Calgary, 2009.

[4] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[5] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44:89–113, 2015.

[6] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073. IEEE, 2012.

[7] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 207–210, 2010.

[8] Scott D Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):14, 2013.

[9] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[10] Leo A Goodman. Snowball sampling. *The Annals of Mathematical Statistics*, pages 148–170, 1961.

[11] Nathan Hawes, Stuart Marshall, and Craig Anslow. Codesurveyor: Mapping large-scale software to aid in code comprehension. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 96–105. IEEE, 2015.

[12] Austin Z Henley and Scott D Fleming. The patchworks code editor: toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2511–2520, 2014.

[13] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness.* Number 6. Harvard University Press, 1983.

[14] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, 2006.

[15] Andrew J Ko, Htet Aung, and Brad A Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering*, pages 126–135, 2005.

[16] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, (12):971–987, 2006.

[17] Barbara Landau and Ray Jackendoff. “what” and “where” in spatial language and spatial cognition. *Behavioral and brain sciences*, 16(2):217–238, 1993.

[18] Thomas D LaToza and Brad A Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194, 2010.

[19] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[20] Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1323–1332, 2008.

[21] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2010.

[22] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31, 2014.

[23] Rob McCarney, James Warner, Steve Iliffe, Robbert Van Haselen, Mark Griffin, and Peter Fisher. The hawthorne effect: a randomised, controlled trial. *BMC medical research methodology*, 7(1):30, 2007.

[24] Nicholas Nelson, Anita Sarma, and André van der Hoek. Towards an ide to support programming as problem-solving. In *PPIG*, page 15, 2017.

[25] Fred Paas, Juhani E Tuovinen, Huib Tabbers, and Pascal WM Van Gerven. Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist*, 38(1):63–71, 2003.

[26] Nick R Parsons, M Dawn Teare, and Alice J Sitch. Science forum: Unit of analysis issues in laboratory-based research. *Elife*, 7:e32486, 2018.

[27] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? how production bias affects developers' information foraging during debugging. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 11–20, Sep. 2015.

[28] D. Piorkowski, S. Penney, A. Z. Henley, M. Pistoia, M. Burnett, O. Tripp, and P. Ferrara. Foraging goes mobile: Foraging while debugging on mobile devices. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–17, Oct 2017.

[29] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1471–1480, 2012.

[30] David Piorkowski, Austin Z Henley, Tahmid Nabi, Scott D Fleming, Christopher Scaffidi, and Margaret Burnett. Foraging and navigations, fundamentally: developers' predictions of value and cost. In *Proceedings of the 2016 24th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, pages 97–108. ACM, 2016.

[31] David J Piorkowski, Scott D Fleming, Irwin Kwan, Margaret M Burnett, Christopher Scaffidi, Rachel KE Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3063–3072, 2013.

[32] Peter Pirolli and Stuart Card. Information foraging. *Psychological review*, 106(4):643, 1999.

[33] Peter L. T. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information.* Oxford University Press, Inc., New York, NY, USA, 1 edition, 2007.

[34] Philip T Quinlan and Richard N Wilton. Grouping by proximity or similarity? competition between the gestalt principles in vision. *Perception*, 27(4):417–430, 1998.

[35] Darrell R Raymond. Reading source code. In *CASCON*, volume 91, pages 3–16, 1991.

[36] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE, 2012.

[37] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[38] Susan Elliott Sim and Richard C Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th international conference on Software engineering*, pages 361–370. IEEE, 1998.

[39] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *Kongelige Danske Videnskabernes Selskab*, 5(4):1–34, 1948.

[40] Jamie Starke, Chris Luce, and Jonathan Sillito. Searching and skimming: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 2009.

[41] M-AD Storey, F David Fracchia, and Hausi A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[42] John Sweller, Paul Chandler, Paul Tierney, and Martin Cooper. Cognitive load as a factor in the structuring of technical material. *Journal of experimental psychology: general*, 119(2):176, 1990.

[43] Max Wertheimer. Laws of organization in perceptual forms. 1938.

[44] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.