Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place

Marjan Adeli*, Nicholas Nelson*, Souti Chattopadhyay*, Hayden Coffey†, Austin Z. Henley†, and Anita Sarma*
* Oregon State University,
Corvallis, OR, USA
{adelima, nelsonni, chattops, anita.sarma}@oregonstate.edu
† University of Tennessee-Knoxville,
Knoxville, TN, USA
hcoffey1@vols.utk.edu, azh@utk.edu

***Abstract*—Code comprehension, especially understanding relationships across project elements (code, documentation, etc.), is non-trivial when information is spread across different interfaces and tools. Bringing the right amount of information, to the place where it is relevant and when it is needed can help reduce the costs of seeking information and creating mental models of the code relationships. While non-traditional IDEs have tried to mitigate these costs by allowing users to spatially place relevant information together, thus far, no study has examined the effects of these non-traditional interactions on code comprehension. Here, we present an empirical study to investigate how the right information at the right time and right place allows users—especially newcomers—to reduce the costs of code comprehension. We use a non-traditional IDE, called Synectic, and implement link-able annotations which provide affordances for the accuracy, time, and space dimensions. We conducted a between-subjects user study of 22 newcomers performing code comprehension tasks using either Synectic or a traditional IDE, Eclipse. We found that having the right information at the right time and place leads to increased accuracy and reduced cognitive load during code comprehension tasks, without sacrificing the usability of developer tools.**

***Index Terms*—Code comprehension, annotation, information foraging, Integrated development environment (IDE), user study**

I. Introduction

Code comprehension is essential to software developement and comprises a large portion of developers' activities. For example, in a field study, Xia et al. found that software professionals spent 58% of their development efforts on code comprehension [1]. Given the large role that code comprehension plays, research has sought to map the processes through which developers understand a code base. For instance, Sillito et al. identified four categories of questions developers ask during maintenance tasks [2], which map to the code comprehension processes of: find an initial focus point, expand understanding around that point, and understand the concepts that connect related code entities, and build a mental model of multiple groups of related entities.

A common thread through each of these steps is that developers "forage" for relevant information in order to create a mental model of the system and the relationships between its various components. In fact, prior research has shown that foraging accounts for up to 35% of a developer's time during maintenance tasks [3], [4].

However, foraging for information is not easy. One study found that 50% of navigation (when foraging) yielded less information than developers expected and 40% of navigation required more effort than the developer predicted [5]. This cost is likely considerably higher for newcomers to a project, who haven't yet formed a mental model of the feature set and the different relationships between code elements.

Newcomers rely on tools to come up-to-speed on large, varied sources of relevant project information. However, the way traditional IDEs are set up limits direct management of relevant information [6], [7], which in turn creates barriers in understanding the relationship between software features [8]. For example, to fully understand the codebase, developers may refer to other artifacts such as documentation, issue reports, sample code, and design rationale [9]. Traditional IDEs treat code as the primary artifact, often relegating other artifacts out of scope for direct support. Additionally, the primary mechanism for managing multiple code documents in traditional IDEs is a combination of single-purpose window panes, un-ordered tabs, and one-off dialog boxes. These "bento-box" style interfaces partition relevant information without providing affordances for understanding the relationships between them. This increases navigational overhead and inhibits developers from organizing information to fit their current task (or mental model) [10], [11], [12], [13].

We focus on three aspects that can ease code comprehension: (A1) allowing interaction with artifacts that are more than just code, (A2) placing relevant information closer together (either spatially or in groups), and (A3) externalizing and recording relationships between artifacts. Past work on non-traditional IDEs—using a canvas or ribbon-based approach—have sought to ease navigation and understanding of project structure [14], [15], [16], [17], [18]. These studies have addressed aspect (A1) to an extent (e.g. via augmenting code with debugger output inline [16]) and aspect (A2) based primarily on syntactic linking (e.g. bubbles containing code linked together by call graph relationships [14]), but none have directly addressed aspect (A3), which is an important element of code comprehension.

Further, no in-depth studied investigate how these aspects facilitate code comprehension among developers. Without an understanding of the effects of these aspects, we miss

978-1-7281-6901-9/20/$31.00 ©2020 IEEE

out on valuable insights to design processes and tools that intend to improve code comprehension. To address this gap in knowledge, we conducted a user study to compare the effects of a traditional IDE (Eclipse) with a canvas-based IDE (Synectic) that facilitates these three aspects (A1–A3) of code comprehension through annotations. Annotations allow developers to gather documentation relevant to the code (A1), juxtapose these code and documentation on a canvas (A2), and link the code artifacts with the relevant documentations (or other code artifacts) (A3) to capture and annotate the relationship between artifacts.

In our between-subject study of 22 participants, we analyzed how project newcomers used annotations to complete a set of four code comprehension tasks. We focused on newcomers as they haven't yet built a mental model of the codebase and need significant effort to comprehend it [19], [20].

To understand participants' code comprehension activities, we analyzed the accuracy, time, and cognitive load of each participant and examined the results through two lenses. First, we used an Information Foraging Theory lens [21] to analyze how participants "foraged" for the right information to complete each comprehension task. Second, we investigated how annotations helped participants comprehend code using the four categories of comprehension questions described by Sillito et al. (i.e., finding focus points, expanding understanding around that focus point, connecting related elements, and contextualizing the information to the larger codebase) [2].

The results of our study show that the current "bento-box" style arrangement of code in traditional IDEs create hurdles to foraging for the right information. Conversely, annotations helped participants in finding and arranging the right information at the right place.

II. Background & Related Work

A. 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 about the underlying information (which is primarily code in the case of software development).

Several studies have examined the types of information that developers seek during development tasks [6], [2], [22], [23], [24], [25]. In particular, we use the four categories of code comprehension questions identified by Sillito et al. [2]. 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 that examine how this initial focus point relates to other entities, (3) questions that explore the combined behavior of related entities, and (4) contextual questions that build upon knowledge spanning multiple groups of related entities.

These questions enable developers to build mental models of the underlying code and facilitate the ability to confidently make changes to that code [10]. We use comprehension questions that span these four categories in order to capture the complete code comprehension process of newcomers building understanding within a codebase.

B. 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 [21] explains how humans seeking 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 [3], [26], [4], [27], [28]. IFT has also been applied to design tools supporting development activities [4], [29], [3].

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). 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 [5]. 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 patches, and perceived potential for future patches.

We use an IFT lens to examine the navigational aspects of comprehending code within an IDE, since the scale of information available within such tools requires "foraging" in order to locate and comprehend relevant information.

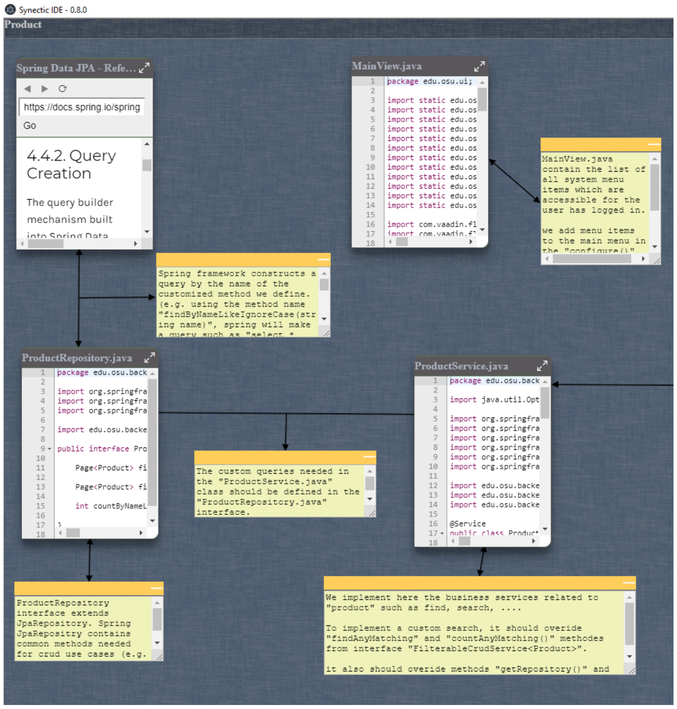
C. Alternative user interfaces in IDEs

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 [30], navigating code [31], [32], and managing context when switching tasks and environments [33].

Alternative IDEs attempt to tackle some of these deficiencies through novel user interfaces that allow related information to be clustered in bubbles [14], that span infinitely expandable canvases [15], and maintain relevant information in close proximity [16]. These visual metaphors have also spawned efforts to constrain the amount of information into ribbons of sequentially related code snippets [18].

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 are not exclusive to those IDEs.

III. Annotations in Synectic



4.4.2. Query Creation

The query builder mechanism built into Spring Data

```
package edu.osu.ui;

import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os
import static edu.os

import com.vaadin.fl
import com.vaadin.fl
```

```
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()"
```

```
Spring framework constructs a
query by the name of the
customized method we define.
(e.g. using the method name
"findByNameLikeIgnoreCase(str
ing name)", spring will make
a query such as "select *
```

```
package edu.osu.back

import org.springfra
import org.springfra
import org.springfra

import edu.osu.backe

public interface Pro

    Page<Product> fi

    Page<Product> fi

    int countByNameL

}
```

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

```
package edu.osu.back

import java.util.Opt

import org.springfra
import org.springfra
import org.springfra
import org.springfra
import org.springfra
import org.springfra

import edu.osu.backe
import edu.osu.backe
import edu.osu.backe

@Service
public class Product
```

```
ProductRepository
interface extends
JpaRepository. Spring
JpaRepositry contains
common methods needed
for crud use cases (e.g.
```

```
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
```

Fig. 1: Synectic provides a canvas-based environment containing spatially arranged cards of relevant information (code, webpages, etc.), along with annotations that link information to individual cards and between related cards. This example groups cards and annotations related to implementing CRUD operations on the `product` page in Vaadin Bakery App (task T4).

Synectic is an IDE designed as a canvas-based environment, similar to Code Bubbles [14] and Moldable Debugger [17], with spatial interactions as the central interaction paradigm. 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. Cards can be of many types; e.g. code cards, web browser cards, etc.

To further capture the relationships between these cards, we also include an annotation overlay that includes annotations and links that can be attached to one or more cards placed on the canvas. Figure 1 presents a snapshot of the Synectic interface, including three cards containing code and a browser card (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 contain arbitrary text, including a mix of code and documentation, and exist as long as the linked card(s) exist on the canvas. Sharing annotations among developers on the same project can be accomplished through backup files that record the state of the canvas within Synectic. The annotation overlay is provided in order to capture and express the previously hidden relationships that developers intuitively create in their minds [10], [34].

A. *Support for Foraging*

The annotations within Synectic provide a system for exposing and archiving navigation pathways of developers. This ensures that revisiting information for similar tasks does not incur the same costs as the first foraging session [21], [26], [5]. All subsequent foraging sessions can benefit from the presence of annotations that indicate the value of information features, and navigation pathways for particular cards.

B. *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 between IDEs and other software systems (e.g. compilers, build systems, testing, etc.) [35].

The conventional bento-box 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 understandable by looking at the arrangement of tabs, and often conveys no information beyond the order in which they were opened [5].

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 related code files into individual cards. The developer can then create a group of syntactically-related cards that contain code involved directly in the bug, and another group of semantically-related code cards that provide further context when developing a fix for the bug. 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).

C. *Support for Maintaining*

Mental models are constructed representations of real world that mirror a working understanding of observed phenomena [36]. Within software development, mental models contain a developers' knowledge and insights into both code and external constraints on the use of that code [10], [37]. Synectic provides direct representation of these mental models through spatially arranged cards of information, annotated between and within sources, that 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 [10]. This cost affects locating relevant information (see Section III-A), and sorting through that information to create a mental model that is relevant to the current task (see Section III-B). 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 [2], [4]. 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 examine the benefits of annotations for code comprehension through user studies described in the Study Design and Results sections below.

IV. Study Design

To understand the effects of annotations on code comprehension, we conducted a controlled lab study comparing annotations in Synectic to the notes functionality included in Eclipse–a traditional IDE. We specifically aim to answer the following research questions:

RQ1: How do annotations affect code comprehension among newcomers?

- a: Do annotations increase the accuracy of responses?
- b: Do annotations reduce the time to task completion?
- c: Do annotations reduce cognitive load?

RQ2: How usable are annotations for newcomers?

Participants were randomly assigned to either Synectic or Eclipse and asked to complete four program comprehension tasks using the assigned IDE. Between tasks, participants were assessed for perceived cognitive load. Finally, at the conclusion of all four tasks, participants were asked to complete a brief questionnaire on the perceived usability of their assigned IDE.

A. Participants and Treatments:

Our participants comprised graduate-level computer science students recruited through convenience and snowball sampling [38]. These participants represented our target population of project newcomers and allowed us to assess the appropriateness of annotation features in onboarding tasks.

We recruited 22 participants using university mailing lists. Table I illustrates the demographic distribution of participants (13 men, 8 women, and one participant preferred not to disclose gender). The median level of programming experience among all participants was 5 years ($mean$ = 7.4 yrs, $S.Dev.$ = 5.5 yrs for Eclipse group; $mean$ = 7.0 yrs, $S.Dev.$ = 5.0 yrs for Synectic group).

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

TABLE I: Study Participant Demographics

[i] Participant (E for Eclipse, S for Synectic) [ii] Gender (M for Man, W for Woman, P for Prefer not to disclose) [iii] Years of soft. dev. experience

We assigned 11 participants to each treatment group using a stratified sampling [39] based on participants' programming experience; pairing similarly experienced participants and assigning one participant to each treatment. All participants in the Eclipse group reported some degree of familiarity with Eclipse, and none of the participants in the Synectic group reported familiarity with Synectic.

Each study session was time-boxed to two hours. First, we obtained participant's consent and provided a walk-through of the assigned IDE, the target project for the study, and the study protocol. Participants were asked to think aloud during the study, which was captured using audio recordings and screen capture software. Prior to starting the study tasks, participants were asked to complete a brief warm-up task in order to become comfortable with the study protocol. After the study tasks, participants were asked to complete a usability questionnaire related to the annotation/notes features within the assigned IDE. Participants were offered US$20 in compensation at the conclusion of each session.

B. Project and Tasks:

The tasks focused on maintenance tasked within the Vaadin Bakery App[1], which is an open-source Java project ($LOC \approx 5000$) designed for bakery shop sales and orders management. The project includes functionality to keep track of product inventory, customers, employees, and a visual dashboard to summarize all transactions.

A senior developer on the project provided documentation and information that would typically be conveyed to newcomers attempting to understand the codebase. For Synectic, he laid out the cards and added onboarding documentation as annotations. For Eclipse, he created an onboarding document file and linked to the code. The documentation information in both treatments were identical.

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

Task	Part	Prompt/Question
T1	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.
T2	A	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?
T3	A	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?
T4	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 II: User study tasks; each task is divided into a navigation prompt (Part A) and a comprehension question (Part B).

We gave participants four comprehension tasks, which we created. These tasks were designed to be representative of common problems that newcomers experience when onboarding [40]. The senior developer then verified these tasks were typical onboarding tasks, and the ordering (in terms of complexity) was appropriate for newcomers to attempt.

For each task, we asked participants to first locate elements in the codebase that are relevant to a particular feature (Part A), and then ask them to answer comprehension questions that require in-depth understanding of that portion of the codebase (Part B); see Table II 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 by completing a one-question survey that asked *“how mentally demanding was the task?”* (using a balanced Likert-scale response, where 1 is *very low* and 7 is *very high*) [41]. 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 [42].

C. Measurements and Constructs:

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

Accuracy ($\mathcal{A}$): Accuracy of a response is dependent on the completeness and correctness of individual elements within that response. Therefore, we use the balanced Sørensen–Dice coefficient (F_1-score) [43] 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. Since each task is comprised of two parts, we calculate and report the mean of accuracy scores for each task.

Time: The time between when a participant switched to the IDE from the question to when they pressed the `next` button on the task prompt.

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*).

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 [42].

D. Analysis:

Our study design is factorial. Each participant was assigned to a treatment (Eclipse or Synectic)—the between-subject factor—and performed four tasks T1 to T4 (in order)—the within-subject factor. This constitutes an F1-LD-F1 design, an instance of Fx-LD-Fy family of designs, where x refers to the number of between-subjects factor (i.e. treatment) and y are the number of within-subjects factors within each x (i.e. tasks) [44], [45], [46], [47].

This design gave us 88 observations across all participants, treatments, and tasks (11 × 2 × 4). Our analysis is bounded by two constraints: measures from each task are not independent across participants, and sample sizes per measurement are small. Therefore, we chose a rank-based (RB) non-parametric ANOVA (NP). Thus, we compare the the accuracy, time-to-completion, and cognitive load measurements between treatments using the Rank Based Non-Parametric (RBNP) ANOVA-type tests [48], [49].

We use RTE [44] to explain the magnitude of differences between treatments since calculating effect size (a mean-based analysis) is not appropriate for Rank-based ANOVA. RTE provides the probability of one group having a higher observed value than the other. For example, an RTE value of 60% for Group A (for accuracy), means that in a random sampled observation from each group, there is 60% probability that Group A observations has a higher accuracy than Group B.

We evaluate usability by comparing the normalized SUS score of the participants (sum of the scores across all four tasks transformed to a $0-100$ scale) using the Wilcoxon Rank-Sum test [50].

V. Results

Here we present the descriptive statistics and results of the statistical tests comparing the effect of annotations on the following constructs:

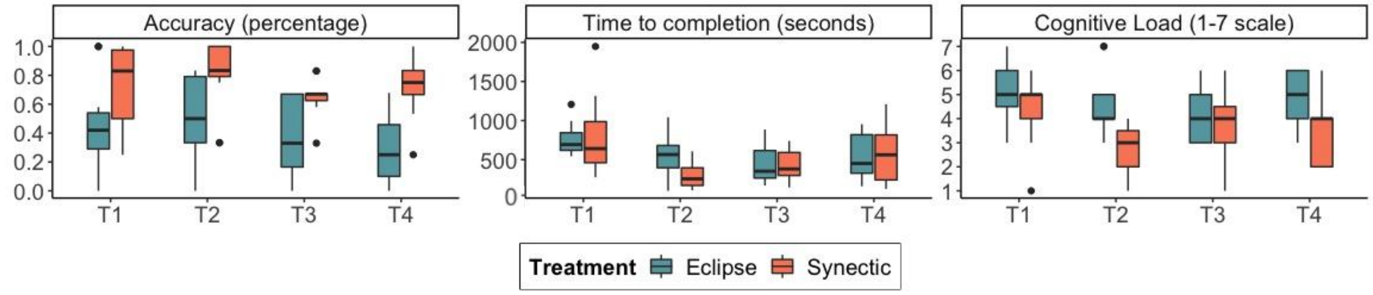


Fig. 2: Boxplots with error bars for each task T1–T4 grouped by treatment (Synectic or Eclipse) and included for each measured dimension: accuracy of participant task responses as a percentage (left); time to task completion in seconds (center); participant perceptions of cognitive load on a 1–7 scale, where 1 is *very-low* and 7 is *very-high* (right)

A. *RQ1a: Accuracy*

The RBNP ANOVA-type test showed that accuracy of participants were significantly different across the two treatments (p-value < 0.001, statistic $= 19.46488$). The RTE statistics showed that Synectic (RTE $= 64.80\%$) has a higher effect on the accuracy of the responses compared to Eclipse (RTE $= 35.20\%$). Figure 2 (*left*) shows that Synectic participants had a higher median accuracy for all four tasks.

B. *RQ1b: Time*

The mean and median of time taken by Synectic group (mean = 575, Mdn = 440 seconds) was smaller than that of Eclipse group (mean = 517, Mdn =607 seconds), however the RBNP ANOVA-type test failed to show a significant difference across the two treatments in the time to complete a task (p-value $= 0.22$, statistic $= 1.607723$). Figure 2 shows the distribution of time taken for all four tasks.

C. *RQ1c: Cognitive Load*

The RBNP ANOVA-type test showed that the cognitive load perceived by participants were significantly different across the two treatments (p-value $= 0.003$, statistic $= 11.52591$). The RTE statistics shows that Synectic (RTE $= 38.75\%$) has a lower effect on the cognitive load of the participants compared to Eclipse (RTE $= 61.25\%$). Figure 2 shows that Synectic participants had lower median cognitive load than Eclipse participants for tasks T2 and T4, but similar median cognitive load for tasks T1 and T3.

D. *RQ2: Usability*

The average usability score reported by Synectic participants (73.74) was higher than the average usability score reported by Eclipse participants (53.79). Figure 3 shows the boxplot distribution of usability scores reported by participants in each treatment group. Also, a Wilcoxon rank-sum test showed that the groups differ in median usability score (W=22, p-value < 0.0123, two-sided Wilcoxon rank-sum test). The difference in median usability scores between Synectic participants (median: 72.22) and Eclipse participants (median: 52.78) was also shown to be statistically large (Cliff's Delta $\delta = 0.64$).

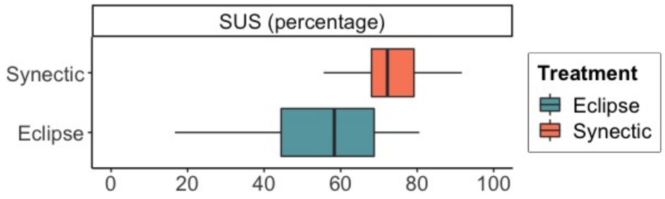


Fig. 3: Boxplot of System Usability Scale (SUS) percentages grouped by treatment; 72.22% median for Synectic, 52.78% for Eclipse

VI. Discussion

Overall, the quantitative results show that annotations helped participants answer comprehension questions more accurately and with significantly less cognitive load. Although the Synectic group took less time overall, the differences were not statistically significant. To delve deeper into participants' performance in the code comprehension tasks, we qualitatively analyzed the data to investigate: (1) where and how participants faced problems in completing their tasks through an IFT lens [51], and (2) how participants used IDEs to perform the comprehension steps identified by Sillito et al. [2].

A. *Using IFT to understand navigation and foraging behavior*

Navigating to the right place to get the right information was difficult for Eclipse participants. Before discussing participant behavior through an IFT lens, we first describe the IFT constructs in the context of our study. In Eclipse, we considered methods in the code as well as different Eclipse views (e.g. code editor window, the package explorer) as *patches*. The text in the *patches* (e.g., file names in the package explorer) were *information features*. In the case of Synectic, each card (containing code or any other type of artifact) was a *patch* and the text inside was the *information feature*.

Recall that participants were provided onboarding documents; in Eclipse this was displayed as a text document (within the IDE) and in Synectic this was presented via a set of annotations (Section IV). For the Eclipse group, each paragraph in the document was a *patch* as each was a separate logical unit, and the onboarding document was a collection of sequentially arranged *patches*. For the Synectic group, each annotation was a *patch*, and when these were arranged *patches* next to relevant code it created a relational topology.

Using these IFT constructs, we now analyze how participants navigated to the right information using the assigned IDE for each treatment group.

1) Foraging in the document: Participants in the Eclipse group had to forage for the right *prey* by navigating across sequentially arranged *patches* (paragraphs) in the onboarding document. Eclipse participants followed two strategies to avoid having to read the entire document; a visual *skimming* of the document and *searching* for keywords within the document. While sometimes these strategies proved successful, participants often faced unforeseen difficulties.

When *skimming*, Eclipse participants often overlooked the *patch* containing the intended *prey* and had to spend additional time and effort skimming back and forth. For example, E11 initially skimmed from bottom to top of the document looking for information related to code to control items in the system menu (for task T1). However, after failing to locate the relevant information, she spent 1.5 minutes scrolling up and down before finally locating the intended *patch*.

When *searching* using keywords, Eclipse participants were occasionally stymied from locating the desired *prey* within the document. This was typically caused by the use of incorrect keywords (or synonyms of relevant keywords), which would return either irrelevant *patches* or no *patches*. Furnas et al. [52] 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 of keywords causes newcomers to fail more often in achieving the actions or information they want, which is known as the *vocabulary problem*.

For example, during task T4, E2 used "search" as a keyword for locating information related to updating the search functionality in the `product` page. Since the documentation used the term "find" instead of "search", in this case, E2's searches returned only irrelevant *patches*. Other Eclipse participants, E4 and E5, had similar struggles with the *vocabulary problem*.

However, information foraging was less of a challenge for Synectic participants. This was likely because each card was a *patch* of information that linked directly to the related code. This proximity of related information reduced the foraging costs, as participants could quickly skim groups of information to identify if they were relevant. For example, even when the documentation used different words than what participants expected, because of the proximity of the annotations and cards they were able to get to the right information.

2) Foraging across code and document: Navigating to the correct *patch* in the onboarding document was only the first step. After finding the right information in the document *patch*, Eclipse participants had to refer to the appropriate code *patches* in order to comprehend the code. Navigating between these *patches* imposed additional costs on foraging, which were not observed among Synectic participants.

Foraging within the large code base had its associated costs. Code is hierarchical in nature (i.e. a method *patch* is contained inside a class *patch*, which is further contained in package or component *patches*). A relevant piece of code required for a task could be at any level of this hierarchy.

Eclipse participants relied on the Project Explorer view in order to navigate to potentially relevant code *patches*. The Project Explorer 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 made foraging strategies non-trivial and cost-prohibitive.

For example, E6 believed that the relevant piece of code (*prey*) for T1 was in a *patch* that discussed the user interface. However, the *prey* (`Mainview.java`) was actually six levels deeper than the file that E6 was 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.

Foraging costs get higher when *prey* is scattered across hierarchy (e.g. relevant classes were spread across different packages). Eclipse participants had to navigate across different levels of the hierarchical structure several times for each *prey*. This imposed high cognitive loads and reduced their focus.

Moreover, switching between different types of artifacts–code and document patches–that have different type and structure of content incurred additional cognitive load [53]. As the tasks' complexity increased, the increase in cognitive load was noticeable; participants had to consume complex information from a *patch* (document) and switch to another *patch* (code) to apply the information.

In Synectic, however, documentation *patches* were adjacent to the relevant code *patches* and connected through annotations and links. This combination created a unified source of information that required less attention splitting [53] and led to substantially reduced foraging costs and cognitive load.

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

Annotations helped participants perform better in the Synectic group. We use the four stages of code comprehension described by Sillito et al. [2] to understand how participants used annotations. The tasks in the study, because of their nature, spanned multiple code comprehension stages. Here we use the example of participant S5 in task T4 (see Table II for task description) to describe how participants completed each of the four code comprehension stages.

During T4, participants needed to understand the "search" functionality and modify the search based on the `product name` and `price` fields to that search functionality.

(1) Finding an initial focus point suggests that developers start comprehension by finding points/entities that are relevant to the task. In Synectic, annotations provided cues that helped participants narrow down the search space to a reduced set of *patches* potentially containing the initial focus point.

For example, S5 started this process by inspecting annotations that might lead to information about the task (i.e. *"product-related search"*). During this inspection, S5 found an annotation connected to a group of cards titled `product`, which explained *"how to add a [CRUD] page using product page as an example"*. He expanded the group to hunt for other annotations, which gave him a clue to product search and directed him to the `ProductRepository.java` card. This set of annotations and cards formed his initial focus point for further exploration.

(2) Building on those focus points is done by exploring other *patches* of information related to the focus point. Synectic, through the use of annotations, helps make the relationships between cards explicit.

Continuing the example of S5 in T4, after locating the `ProductRepository` class, S5 said, *"[card] contains a* `find` *method which seems to be interesting."* He therefore decided to explore other code locations that called the `find` method, which he did by following the annotation link from the `ProductRepository.java` to another card (`ProductService.java`). From there, he continued following the annotations to other related cards further expanding the focus points.

(3) Understanding the concepts between related entities can be gained when developers ask questions about concepts in the code that involve multiple entities. In Synectic, annotations linked sets of related cards (or groups of cards), which provided an overview of concepts and relationships between these entities.

Continuing with our example, S5 wanted to understand how the `ProductService` and `ProductRepository` classes interacted with each other to provide search functionality, which he easily did by reading the annotation connecting the respective cards. S5 was able to quickly determine that a method in the `ProductService` class calls a `find` method in the `ProductRepository` class. Thus, this annotation link allowed him to expand his conceptual understanding of entities related to the search functionality.

(4) Understanding concepts across multiple groups of related entities can be achieved when developers ask contextual questions about groups of *patches* located at all levels of the codebase, or even outside the project itself.

The information within Synectic's annotations, along with links between groups of cards, helps to relate concepts from individual groups in order to build a contextual understanding of the overall codebase. Considering our example, after S5 read and comprehended the annotations and cards linked to the `ProductService` and `ProductRepository` classes, he found additional documentation for using Spring Data's `JparRepository` library to make custom queries by method name. The web browser card containing this documentation (along with the rest of the context) provided S5 with enough understanding to be able to accomplish task T4, despite his previous unfamiliarity with the target Java project or the `JparRepository` library.

Typical IDEs (e.g. Eclipse) provide limited capabilities for expressing relationships at the conceptual levels (phases 3 and 4 from Sillito et al. [2]). Synectic, on the other hand 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 facilitated all four phases of code comprehension questions described in Sillito et al. [2].

VII. 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. Further, counterbalancing to remove learning effects was not appropriate as the tasks were ordered to allow newcomers to gradually familiarize with the project (which is similar to an onboarding process). 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 [54].

As with any empirical research involving participant observation, responses could have been affected by the Hawthorne effect [55]. 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.

VIII. Conclusion

In this paper, we evaluated the effects of annotations that support foraging for information and code comprehension through linking relevant information and code on a canvas-based IDE. To validate annotations in Synectic, we conducted a user study comparing newcomer task support for foraging and comprehension within a traditional IDE (Eclipse) and our canvas-based IDE.

The results of our user study show that providing the right information at the right place and time helps newcomers answer comprehension questions with significantly more accuracy, in less time. Tool support for both foraging and comprehension also significantly reduced the cognitive load on participants. Participants also found annotations in Synectic to be more usable than traditional IDEs like Eclipse. Annotations provided contextually-relevant information adjacent to the code that participants sought to comprehend, which allowed participants to quickly forage to the relevant information and build accurate mental models of the codebase.

Acknowledgment

We thank our participants for their time, and the reviewers for their feedback. This work was supported by the National Science Foundation under Grants No. 1560526 and 1815486.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, “Measuring program comprehension: A large-scale field study with professionals,” *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[2] J. Sillito, G. C. Murphy, and K. De Volder, “Asking and answering questions during a programming change task,” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[3] J. Lawrance, R. Bellamy, M. Burnett, and K. 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*, 2008, pp. 1323–1332.

[4] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, “An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 2, p. 14, 2013.

[5] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. 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*, 2016, pp. 97–108.

[6] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” *IEEE Transactions on software engineering*, no. 12, pp. 971–987, 2006.

[7] M. P. Robillard, W. Coelho, and G. C. Murphy, “How effective developers investigate source code: An exploratory study,” *IEEE Transactions on software engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[8] G. C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the elipse ide?” *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[9] D. Čubranić and G. C. Murphy, “Hipikat: Recommending pertinent software development artifacts,” in *Proceedings of the 25th international Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 408–418.

[10] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: a study of developer work habits,” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[11] V. M. González and G. Mark, “” constant, constant, multi-tasking craziness” managing multiple working spheres,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 113–120.

[12] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, “The work life of developers: Activities, switches and perceived productivity,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017.

[13] S. Chattopadhyay, N. Nelson, Y. R. Gonzalez, A. A. Leon, R. Pandita, and A. Sarma, “Latent patterns in activities: A field study of how developers manage context,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19. IEEE Press, 2019, p. 373–383. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00051

[14] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. 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*, 2010, pp. 2503–2512.

[15] R. DeLine and K. Rowan, “Code canvas: zooming towards better development environments,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 207–210.

[16] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger canvas: industrial experience with the code bubbles paradigm,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1064–1073.

[17] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz, “Practical domain-specific debuggers using the moldable debugger framework,” *Computer Languages, Systems & Structures*, vol. 44, pp. 89–113, 2015.

[18] A. Z. Henley and S. 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*, 2014, pp. 2511–2520.

[19] I. Steinmacher, I. S. Wiese, T. Conte, M. A. Gerosa, and D. Redmiles, “The hard life of open source software project newcomers,” in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 72–78. [Online]. Available: https://doi.org/10.1145/2593702.2593704

[20] R. Yates, N. Power, and J. Buckley, “Characterizing the transfer of program comprehension in onboarding: an information-push perspective,” *Empirical Software Engineering*, vol. 25, no. 1, pp. 940–995, 2020.

[21] P. Pirolli and S. Card, “Information foraging.” *Psychological review*, vol. 106, no. 4, p. 643, 1999.

[22] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, “How do professional developers comprehend software?” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 255–265.

[23] T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 185–194.

[24] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, “On the comprehension of program comprehension,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 31, 2014.

[25] J. Starke, C. Luce, and J. Sillito, “Searching and skimming: An exploratory study,” in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 157–166.

[26] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, “How programmers debug, revisited: An information foraging theory perspective,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2010.

[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)*, Sep. 2015, pp. 11–20.

[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)*, Oct 2017, pp. 9–17.

[29] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. 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*, 2012, pp. 1471–1480.

[30] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, “The whats and hows of programmers’ foraging diets,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 3063–3072.

[31] A. J. Ko, H. Aung, and B. 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*, 2005, pp. 126–135.

[32] M. J. Coblenz, A. J. Ko, and B. A. Myers, “Jasper: an eclipse plug-in to facilitate software maintenance tasks,” in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, 2006, pp. 65–69.

[33] M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.

[34] X. Chen and B. Plimmer, “Codeannotator: digital ink annotation within eclipse,” in *Proceedings of the 19th Australasian conference on Computer-Human Interaction: Entertaining User Interfaces*, 2007, pp. 211–214.

[35] N. Nelson, A. Sarma, and A. van der Hoek, “Towards an IDE to Support Programming as Problem-Solving,” in *Proceedings of the 2017 Psychology of Programming Interest Group (PPIG)*, 2017, p. 15.

[36] P. N. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983, no. 6.

[37] M.-A. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[38] L. A. Goodman, "Snowball sampling," *The Annals of Mathematical Statistics*, pp. 148–170, 1961.

[39] E. Tipton, L. Hedges, M. Vaden-Kiernan, G. Borman, K. Sullivan, and S. Caverly, "Sample selection in randomized experiments: A new method using propensity score stratified sampling," *Journal of Research on Educational Effectiveness*, vol. 7, no. 1, pp. 114–135, 2014.

[40] S. Balali, I. Steinmacher, U. Annamalai, A. Sarma, and M. A. Gerosa, "Newcomers' barriers... is that all? an analysis of mentors' and newcomers' barriers in oss projects," *Computer Supported Cooperative Work (CSCW)*, vol. 27, no. 3-6, pp. 679–714, 2018.

[41] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. Van Gerven, "Cognitive load measurement as a means to advance cognitive load theory," *Educational psychologist*, vol. 38, no. 1, pp. 63–71, 2003.

[42] J. Brooke *et al.*, "SUS-A quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[43] T. 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*, vol. 5, no. 4, pp. 1–34, 1948.

[44] E. Brunner and M. L. Puri, "Nonparametric methods in factorial designs," *Statistical papers*, vol. 42, no. 1, pp. 1–52, 2001.

[45] E. Brunner and F. Langer, "Nonparametric analysis of ordered categorical data in designs with longitudinal observations and small sample sizes," *Biometrical Journal*, vol. 42, pp. 663 – 675, 10 2000.

[46] E. Brunner and M. L. Puri, "19 nonparametric methods in design and analysis of experiments," *Handbook of statistics*, vol. 13, pp. 631–703, 1996.

[47] S. Domhof and F. Langer, *Nonparametric analysis of longitudinal data in factorial experiments*. Wiley-Interscience, 2002, vol. 406.

[48] E. Brunner and M. L. Puri, "A class of rank-score tests in factorial designs," *Journal of Statistical Planning and Inference*, vol. 103, no. 1, pp. 331 – 360, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378375801002300

[49] J. Feys, "New nonparametric rank tests for interactions in factorial designs with repeated measures," *Journal of Modern Applied Statistical Methods*, vol. 15, no. 1, p. 6, 2016.

[50] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[51] P. L. T. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information*, 1st ed. New York, NY, USA: Oxford University Press, Inc., 2007.

[52] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[53] J. Sweller, P. Chandler, P. Tierney, and M. Cooper, "Cognitive load as a factor in the structuring of technical material." *Journal of experimental psychology: general*, vol. 119, no. 2, p. 176, 1990.

[54] S. E. Sim and R. 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*. IEEE, 1998, pp. 361–370.

[55] R. McCarney, J. Warner, S. Iliffe, R. Van Haselen, M. Griffin, and P. Fisher, "The hawthorne effect: a randomised, controlled trial," *BMC medical research methodology*, vol. 7, no. 1, p. 30, 2007.