An Empirical Study on Code Comprehension: Data Context Interaction Compared to Classical Object Oriented

Héctor Adrián Valdecantos*, Katy Tarrit†, Mehdi Mirakhorli†, James O. Coplien‡
*Universidad Nacional de Tucumán, FACET, Departamento de Ciencias de la Computación, Argentina
†Rochester Institute of Technology, USA
‡Gertrud & Cope, Denmark
hvaldecantos@herrera.unt.edu.ar, {ktics, mxmvse}@rit.edu, cope@gertrudandcope.com

***Abstract*—Source code comprehension affects software development — especially its maintenance — where code reading is one of the most time-consuming activities. A programming language, together with the programming paradigm it supports, is a strong factor that profoundly impacts how programmers comprehend code. We conducted a human-subject controlled experiment to evaluate comprehension of code written using the Data Context Interaction (DCI) paradigm relative to code written with commonly used Object-Oriented (OO) programming. We used a new research-level language called *Trygve* which implements DCI concepts, and *Java*, a pervasive OO language in the industry. DCI revisits lost roots of the OO paradigm to address problems that are inherent to Java and most other contemporary OO languages. We observed correctness, time consumption, and locality of reference during reading comprehension tasks. We present a method which relies on the *Eigenvector Centrality* metric from Social Network Analysis to study the locality of reference in programmers by inspecting their sequencing of reading language element declarations and their permanence time in the code. Results indicate that DCI code in Trygve supports more comprehensible code regarding correctness and improves the locality of reference, reducing context switching during the software discovery process. Regarding reading time consumption, we found no statistically significant differences between both approaches.**

***Keywords*-Program comprehension; Controlled experiment; Human subjects; Programming languages; Programming paradigms; Data Context Interaction; Object-Oriented.**

I. Introduction

Object-Oriented programming was created to organize computation around human mental models [1]. Kay explored how to use objects to represent childrens' operational models of the world around them, and how objects in a computer could extend a child's range of intellectual input into that realm. Kay's vision provided that children could interact with each other, their library, and other facilities through their personal computer (a "Dynabook") as an extension of their cognitive facilities. Objects could be used as the building blocks of problem solving, realizing operational models through a network of cooperating objects.

Smalltalk was an early language used to research these ideas at Xerox PARC, and it would eventually enjoy broad application in academia and industry for general-purpose computing, with strong connection to a visual user interface. Some general principles of object orientation were adopted by early programming languages such as Objective-C [2], C++ [3], and CLOS [4]. These languages enabled Kay's vision in varying degrees. Many of them shifted design focus from specific objects to object templates called classes. A class is like a module which can be instantiated multiple times. Each class defines the data template for its objects, as well as algorithm snippets called *methods* which would invoke each other to realize a use case.

In addition to instantiation, classes differ themselves from modules with a feature called inheritance. Classes could be used to represent (usually hierarchical) classification schemes. One class (called the base class) might represent some set of objects. The programmer could specify subsets of this larger set by writing subclasses, which define each subset in terms of its differences from the larger set. Last, an instance of one of these subclasses could be substituted anywhere the program expected an instance of the base class.

This substitutability proved to be a weak point for program comprehension. Consider o_a and o_b to be names of objects as declared in some object-oriented program. Assume o_b to be declared to be of class C_b. If a method of o_a invokes a method m of o_b, program execution will ensue in that method of o_b. But substitutability means that we cannot, from the program, anticipate what the class of o_b will be at run time. The method m can be in any subclass of C_b. (In a language without static type checking, m might be anywhere.) This makes it unreasonable to understand a program's execution sequencing from the program text alone. It is necessary to wait until the program is run, and only then, with the aid of a testing tool or program exploration tool, can we deduce the sequence of execution with certainty and reason about what the code might do next. So while object-oriented programs make it convenient to reason about access to a given set of encapsulated data, they stand in the way of reasoning about sequences of interactions between objects. In contemporary software, most of these sequences are designed to implement business or technical use cases rather than, say, effecting emergent execution.

One can solve this problem using contemporary object-oriented programming languages by localizing the use case sequencing in one method or class, but this tends to force

the participant objects to become loaded with all the methods that in any way involve them in a use case, or alternatively forces those object's classes to simultaneously inherit from multiple classes, each of which may carry the associated use case logic pertinent to that class. This leads to class hierarchies that are difficult to understand and, in general, to the need for multiple inheritance, whose problems have caused it to become an increasingly marginalized practice over the years. For a more thorough discussion of these issues, see [5].

The Data Context Interaction (DCI) paradigm [6] has been designed to address these challenges and provides a new way of thinking to improve programs' comprehensibility. DCI focuses on objects and their relationships to the roles of human mental models by which end users and programmers reason about them generally. DCI provides an architecture that extends contemporary object-oriented programming from its data-centric structure to focus on the business value of system-level operations. It provides a building block for the system use cases and instead of focusing on individual objects, the DCI paradigm focuses on communication between objects and makes it explicit. As a result, it improves the readability of the code, which helps programmers to reason about the behavior of the system and its use cases easily. This paper provides the first empirical evaluation of the impact of DCI using the Trygve language on program comprehension, compared to the OO paradigm using Java. More specifically, we answer the following research questions:

> ***RQ1:*** *Does the DCI-Trygve approach increase correctness of program comprehension compared to an OO-Java approach?*
> ***RQ2:*** *Does the DCI-Trygve approach decrease the time duration of program comprehension compared to an OO-Java approach?*
> ***RQ3:*** *Does the DCI-trygve approach improve the locality of reference during program comprehension compared to an OO-Java approach?*

The remainder of this paper is organized as follows. In Section II, we first introduce the Data Context Interaction paradigm and the Trygve language. While the research method is described in Section III, the obtained results are presented in Section IV and Section V discusses how we addressed the threats to validity. Finally, we look at related work in Section VI and conclude as for the impact of DCI and OO on code comprehension in Section VII.

II. Data Context Interaction and Trygve

In this section, we briefly introduce the DCI paradigm and trygve language with an excerpt of a runnable code example.

A. Data Context Interaction

DCI is a programming paradigm used to improve the readability of object-oriented programs and comprehension of their run-time behavior, by giving the system behavior first-class standing [6]. The DCI paradigm separates the domain model (**Data**) from use cases and run-time behavior (**Context**), and specifies roles that objects play to accomplish system operations through object collaboration (**Interaction**).

A DCI program structures the source code based on these concepts. For instance, for a *Bank Account System*, a *data* object could be instantiated from an *Account* class. The interface of this object is mainly designed to manipulate its own data, i.e. mutator or accessor methods like *increasing* and *decreasing* the balance or *inquiring* the current balance.

Each *Context* represents one use case and its deviations. An object participating in a use case has responsibilities which the object takes on as a result of playing a particular *Role* for a specific Context. Context code includes the *Role* declarations for a given use case, as well as the code to map these roles into objects at run time and the trigger method to initiate the use case. Classes and interfaces define and structure the state of the system (what-the-system-is), whereas Contexts and Roles define the behavior of the system (what-the-system-does).

Listing 1 Code Snippets for the Money Transfer Use Case in Trygve.

```
1  context TransferMoney {
2      private double amountToTransfer_
3      public TransferMoney(Account src, Account dst, double mnt){
4          SourceAccount = src;
5          DestinationAccount = dst;
6          Bank = this;
7          amountToTransfer_ = mnt.clone;
8      }
9      public double getAmountToTransfer() const {
10         return amountToTransfer_;
11     }
12     public void run() { Bank.transfer() }
13     stageprop Bank {
14         public void transfer() {
15             SourceAccount.withdraw();
16             DestinationAccount.deposit();
17         }
18         public double gets_amount_to_transfer() {
19             return getAmountToTransfer();
20         }
21     } requires {
22         double getAmountToTransfer() const;
23     }
24     role SourceAccount {
25         public void withdraw() {
26             assert(Bank.gets_amount_to_transfer() <=
27                    getBalance(), "Insufficient funds")
28             decreaseBalance(Bank.gets_amount_to_transfer());
29         }
30     } requires {
31         void decreaseBalance(double amt);
32         double getBalance();
33     }
34     role DestinationAccount {
35         public void deposit() {
36             increaseBalance(Bank.gets_amount_to_transfer());
37         }
38     } requires {
39         void increaseBalance(double amt)
40     }
41 }
```

Listing 1 presents a code snippet to illustrate an example of a *Context* for the Bank Account System. The Context declaration *TransferMoney* defines the system operation between two accounts, where data objects are of type *Account* and are used through *Roles* named *SourceAccount* and *DestinationAccount*. Contexts are language elements that represent a use case in the code. The interaction is implemented as roles which are played by objects at run time. The *requires* clause is used for the binding of objects to roles with a duck typing style of

interface matching[1]. The trigger method for this use case is the method *run()* that cues the *Bank* role-player object to initiate the transfer. We can see that the Context is also playing a role within its own Context declaration. A *stageprop* role specifies free-side effect behavior on objects. Objects playing this type of role only use accessor methods.

B. The **trygve** language

The **Trygve** language[2] was created as a research language to explore what a "pure" DCI language might look like. The semantics are richly inspired by *Smalltalk-80* but the syntax borrows heavily from *Java*. This latter choice was purely cosmetic, as we found that potential users would not give a second glance at earlier Smalltalk-based DCI languages: the syntactic culture shock was too high.

The *Trygve* language has classes and interfaces as we find in Java, but adds two new concepts. One is the *Role*. A Role is reminiscent of a Java interface in the sense that it is in part a collection of APIs for an object. However, unlike most Java interfaces, *Trygve* Role methods completely specify the algorithm that implements their respective semantics.

Role methods can invoke the methods of other Roles that are declared in the same Context. A Context instance is an object which provides a locus of coordination for the objects involved in a use case. We bind objects to Roles, thereby making them "role-players" within the Context. The binding is compile-time type safe using a duck typing style of interface matching. One Role method can invoke the method of another object within its Context by specifying that object's Role name and the desired method name and parameters. The Context provides a locus for understanding the flow of execution through a network of connected objects, making use cases full first-class citizens of the language.

III. Method

We conducted the first human-subject controlled experiment to compare the effects of DCI-Trygve and OO-Java on code comprehension. We analyzed the *correctness*, *time consumption*, and *locality of reference* during reading comprehension tasks. For each subject (i.e. students or professionals) reading code, we objectively evaluated their performance and reading behavior in order to state which paradigm helps produce more comprehensible source code.

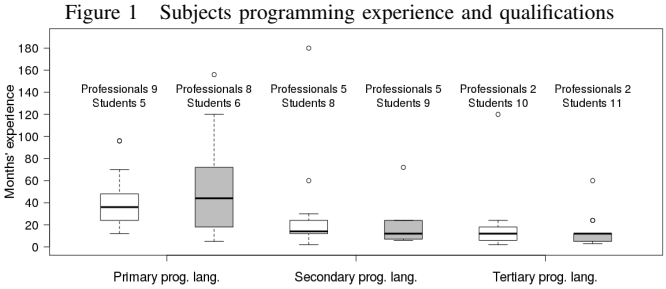
A. Subjects

Twenty-eight programmers with one or more years of programming experience in OO-Java took part in this study. The reason for which OO-Java experience was required is twofold. First, a group of participants were asked to read programs written in the Java language, and second, the other group were asked to read Trygve language that is built upon the Java VM and shares most of the Java syntax. We built two groups of 14 subjects each: the OO-Java and DCI-Trygve groups.

[1]Duck types are public interfaces that are not tied to any specific class. It replaces type to messages capabilities matching of objects. When duck typing, "it's not what an object is that matters, it's what it does" [7].

[2]Trygve language is open sourced in https://github.com/jcoplien/trygve

Figure 1 Subjects programming experience and qualifications



We collected programming experience information: number of months using a programming language and under what qualification (student or professional). Because programmers could know more than one language, we surveyed up to three and considered programmers language primariness (primary, secondary, and tertiary) to make a fair group balancing. Figure 1 shows the distribution of months of experience and the counting of subjects under the experience qualification for the three primariness level (white DCI-trygve, gray OO-Java).

B. Experimental Design

We conducted a one factor with two-treatment controlled experiment and followed a *parallel design* or between-group design where each subject receives only one treatment and we obtain independent measures. Another option was a *crossover design* or within-group design where each subject receives the two treatments and we get repeated measures [8]. Although the benefits of a crossover design are the elimination of the effects of confounding variables, such as experience, as each subject serves as his/her own matched control, and a higher statistical power with fewer subjects, we chose the parallel design to minimize the learning effects, a known challenge in controlled experiments on program comprehension [9]. We also wanted to avoid the effects of paradigm shifting as supported by Wiedenbeck et al. [10], [11], Corritore and Wiedenbeck [12], Abbes et al. [13] and Salvaneschi et al. [14].

Figure 2 Experimental Design Timeline

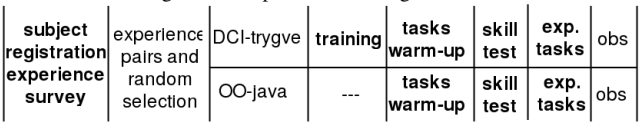


Figure 2 shows the timeline of the experimental design, in bold fonts are the steps where subjects have an active participation. Each step is described below:

1) Subjects register via an online survey to collect programming background experience.
2) Pairing of subjects based on their programming experience reported, followed by their random assignation into groups to block the experience factor.
3) Both groups formed from previous steps.
4) Delivery of the training material on DCI and Trygve language to subjects in the treatment group[3].
5) Warm-up tasks to get familiar with the web interface and the underlying mechanism to run the experiment.

[3]We delivered a 6-page tutorial document focused on the fundamentals of DCI and how to read Trygve language.

We used two small training systems in OO-Java and DCI-trygve were subjects could check their responses.

6) Skill test to assess the knowledge of subjects about the paradigm and language of the group they belong.
7) Experimental tasks on three systems to measure program comprehension: OO-Java systems for the control group and DCI-Trygve systems for the treatment group.
8) At the end of the experiments we collected data for the observed dependent variables described in Section III-F.

C. Description of systems used for this Study

The experiment focused on three systems:

- **Library Management System:** which tracks items borrowed from the library;
- **Menu system:** a command line menu where users can choose among several options;
- **Spell checker system:** which corrects automatically written texts by checking the spelling of each word.

We created strict guidelines to develop equivalent source code in DCI-Trygve and OO-Java versions for each system to ensure both versions were as similar as possible:

- **Functionality:** both versions of the system implement exactly the same use cases.
- **Source code formatting:** we kept a line space between classes in Java, and between classes, Contexts, and Roles in Trygve; we used lines of 80 characters long and the same naming conventions whenever possible.
- **Domain Terminology:** we used the same or similar name for identifiers. The code reflects a similar domain decomposition in terms of classes or interfaces according to the chosen paradigm.
- **Algorithms, Data Structure and I/O:** both versions have the same behavior for the end user; the same input and outputs; and small algorithms remain almost invariable.
- **Paradigm:** each system was written in a representative way of the paradigm it represents in the experiment.

The experiment includes three example systems written in both versions, i.e. OO-Java and DCI-Trygve. While the DCI version was developed by developers in the DCI community[4], we developed the OO-Java version. Table I shows quantitative data about the systems used in the experiment.

Table I Experimental units description

	SLOC		Files		Questions per tasks				
	Java	**Trygve**	**Java**	**Trygve[5]**	$\mathbf{t_1}$	$\mathbf{t_2}$	$\mathbf{t_3}$	$\mathbf{t_4}$	**total**
Library	195	189	8	7	6	5	3	6	20
Menu	228	244	5	5	6	5	3	5	19
Spell c.	237	223	10	5	6	5	3	6	20
totals	660	656	23	17	18	15	9	17	59

D. Tasks Design

Although previous studies of program comprehension relied on tasks such as fixing bugs as in [15], [16], or explaining a solution aloud as in [15], [17], our experiment only implied reading code and answering questions about the system's feature and execution behavior. For each of the three systems described in III-C, participants were asked to answer a set of questions related to four categories as defined below:

- **Check implemented features ($\mathbf{t_1}$)**: in this category, a subject is asked to select True for each implemented feature in the system. For example, in the Library system the subjects are given use cases such as "Users can finish the borrowing without a receipt" or "Users without an id card can use the system", while the first use case is implemented in the system and second one is not.
- **Describe objects interactions ($\mathbf{t_2}$)**: these tasks examine the program comprehension regarding the network of collaborating objects and how objects collaborate between each others and in which direction, reflecting the architecture of a (running) system. Subjects are given a series of object interactions such as, "An object A requests object B to perform function C" and they are asked to indicate if such interaction is valid and occurs in the execution of a system functionality.
- **Look for changed/unchanged objects ($\mathbf{t_3}$)**: these are questions about objects that change state when executing a particular scenario. For example, in the Library system is, given the "Borrow library items" use case, subjects are asked to select true/false/don't know indicating which objects (MockBookScanner, MockCardReader, PaperPrinter) are modified in this use case.
- **Sort execution flow ($\mathbf{t_4}$)**: we request subjects to sort a given list of steps to match the execution flow related to a system functionality considering a given input.

In each category, subjects are shown the source code of a system and asked true/false questions with a don't know (dk) option used to avoid guessing. Guessing introduces noise and produces unreliable results when looking for correctly ranked examinees, as noted in [18]–[20]. Based on these approaches, we adopted an intermediate approach to measure subject understanding. Our method consisted of attributing one point to correct answers and no point to incorrect or don't know answers. This differs from [19] in which they subtracted one point for incorrect answers and used a correct minus incorrect answers formula to attribute scores. While the goal of Mameren et al. [19] was to grade students more accurately, ours was to measure source code comprehension. Our main objective was thus to increase the reliability of our tests by including the *don't know* choice to avoid that subjects answer a question without knowing the answer.

E. Instrumentation and data collection

The instrumentation for the experiment was centralized in a web application[6] we built from scratch to perform the experiment. We fed the web application with all the code, questions, tasks, and guidelines required to run the experiment. The main goals of this application were to unobtrusively track

[4]In https://groups.google.com/forum/#!forum/object-composition the DCI community evolves ideas about the DCI paradigm and Trygve language.

[5]We consider a file as an element language declaration in the code. In Trygve language, a *Context* language element has *Role* declarations inside that cannot exist outside the Context. That's why there are usually less language elements in the DCI-Trygve versions.

[6]https://github.com/hvaldecantos/pctdatacollector

subjects' behavior (time and file switching) during program comprehension processes, to collect the answers, and, to present randomly the tasks and system code to subjects. During the experiment, the same interface and source code visualization style was used for all participants.

F. Analytical approach overview

Each unitary experimental run comprises *parameters* and *variables* to control and observe the results of the behavior of programmers under effects of the *factors* being investigated [8]. We have three dependent variables: correctness score, reading time, and the centrality degree of language elements. These variables help us to answer the RQs presented in Section I. Parameters are part of the experimental setting that are kept constant, as the instrumentation used, like the web application and its data collecting system. Besides, we also have undesirable factors that we block with balancing and randomization techniques to avoid confounding effects. Correctness score and time are the most common observable effects in program comprehension experiments, whereas centrality degree of language elements is introduced to study the locality of reference in program comprehension. Our quantitative dependent variables used are:

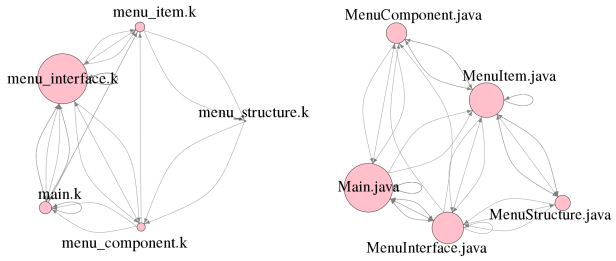
- **Correctness score:** measures the level of comprehension in programmers. It is used to answer RQ1 regarding which approach produces more comprehensible code.
- **Time Consumption:** measures the reading time consumption that programmers spent to finish comprehension tasks. We exclude time spent in reading, interpreting, and answering questions. This variable is used to answer RQ2 regarding which approach consumes less time to accomplish the comprehension tasks; it is also used to answer RQ3 by measuring time regarding the reading time programmers spent in each language element.
- **Centrality degree:** this variable is not directly measured, but computed from the sequencing of reading programmers choose to comprehending a system. This variable is used to answer RQ3 regarding which approach improves the locality of reference of the comprehension process.

G. Locality of reference analysis method

Peter Denning [21] defines *Locality* as a universal behavior of computational processes that tends to refer repeatedly to subset of their resources over extended time intervals. The *locality of reference* principle in a computer states that processes usually use the data and instructions with addresses near or equal to those they have used recently. Subsequently, the locality principle in program comprehension involves the paradigm thinking reflected in the structure of the code and the sequencing of reading that follows programmer decisions about which language element declaration is needed to understand a system operation from source code.

We monitor subject referral to language element declarations during code reading and the time subjects spend reading each language element. We observe *class* and *interface* declarations for Java; and *class*, *interface*, *Context*, and *enactment block* declarations for Trygve language. It is a common practice to store these language element declarations separately in files. As we capture how subjects switch among files, we can recreate this behavior in a directed multigraph[7] to represent how the source code was read. A node or vertex v_i represents a language element declaration and a link or edge e_i between two nodes depicts the file switching behavior. The graph $G = (V, E)$ acts like a cognitive network of language elements built by each programmer during the cognitive task of comprehending a system from reading its source code. For each reader and system example, we put this information into adjacency matrices from which we can build the graphs and compute the centrality metrics for each node.

Figure 3 Menu system graphs - DCI-Trygve (left) and OO-Java (right)



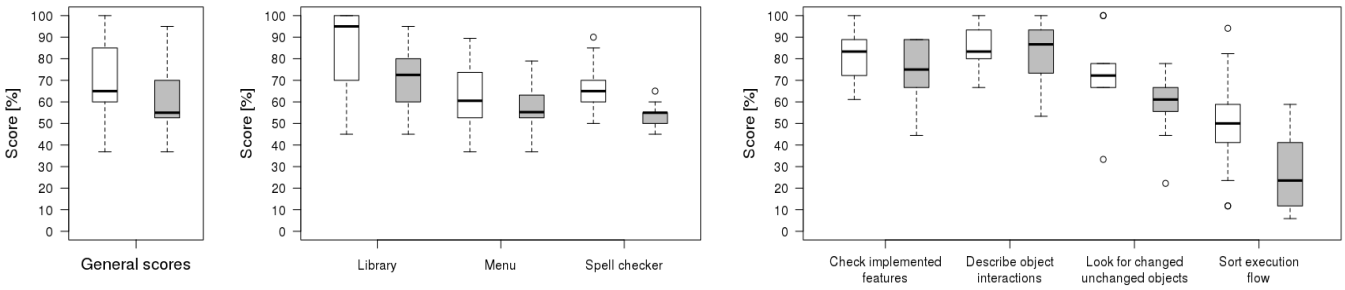
We compute the *Eigenvector Centrality* metric for each language element declared in a system example considering all type of comprehension tasks together. We avoid the analysis of individual tasks to highlight the effect of the paradigm and dissipate the effects of each comprehension task. The *Eigenvector Centrality* metric is used in Social Network Analysis to find the importance degree or influence of a node in a network. It is an extension of the *degree centrality*, which in its *in-degree* form only measures the inwards links to a node and its value shows prestige, whereas in its *out-degree* form measures the gregariousness of a node. "Eigenvector centrality tries to generalize degree centrality by incorporating the importance of the neighbors (or incoming neighbors in directed graphs)" [22]. The centrality of each vertex is proportional to the sum of the centralities of its neighbors. Also, connections to high scoring vertices contribute more to the score of a node. More formally, let A=(a_{ij}) be the adjacency matrix of a graph. The eigenvector centrality x_i of node i is given by:

$$x_i = 1/\lambda * \sum_k a_{k,i} * x_k \quad (1)$$

where $\lambda \neq 0$ is an eigen value. Figure 3 shows the graphs built by two individual subjects from different groups for the Menu system. Nodes are scaled to their importance according to the centrality metric. In the DCI-Trygve case there is a noticeable node that is the *Context* language element, whereas in the OO-java case high levels of importance are present in more than one node. We want to check statistically if these patterns are consistently observed in the sample.

As we also track the permanence time a programmer spent reading each language element, we are able to correlate the centrality degree a programmer gives to a language element

[7] A directed multigraph is a graph without restrictions on the number of links from one node to another node.



(a) Overall ranks **(b)** Ranks per system **(c)** Ranks per task

Figure 4 Box plots for correctness analysis (DCI-Trygve in white, OO-java in gray)

with the time the programmer dedicates to read it. That way, we open a window to find traces of locality of reference principle to study the process of program comprehension.

IV. Analysis and Results

In this section we present the results for A) correctness, B) reading time consumption, and C) locality of reference during reading comprehension tasks. We start with the null hypothesis stating that both approaches produce the same effects:

$H_0: \ \mu_{dci} = \mu_{oo}$
$H_a: \ \mu_{dci} <> \mu_{oo}$

We use the same form of the null hypothesis H_0 and alternative hypothesis H_a for the analysis regarding correctness and time consumption. In these two analyses we use the Mann-Whitney U test in its two-sided form to avoid the test gain greater power in the direction of the alternative hypothesis [23, p. 257] and to make no prior assumption regarding the direction of the results [24, p. 253]. We defined $\alpha = 0.05$ as our tolerance for making a Type I error. (The locality of reference analysis details are shown in Subsection IV-C.)

We recur to the use of median in all the statistical analyses since no assumption on the distribution of the data can be made[8]. Data sets are skewed, thus, the median better estimates the measure of central tendency. This is aligned with the non-parametric statistical methods chosen.

A. Correctness

In this subsection, we present the results concerning our RQ1, regarding which approach increases the correctness of program comprehension.

In the general case, we take into account all systems together. The indicator of correctness is the cumulative score of correct answers submitted for all tasks and normalized with the maximum possible score per system. As each programmer had to comprehend 3 system examples, we have 3 observations per subject, then, we ended with 42 observations to compute the test. Figure 4a shows the descriptive statistics for the DCI-Tryve group in white and for the OO-Java group in grey; and Table II shows the results of the Mann-Whitney U test ($\alpha = 0.05$, two-tailed). Based on these results, we are able to reject the null hypothesis and provide an answer for RQ1 (Section I), concluding that DCI-Trygve approach produces a more comprehensible source code.

[8]The Shapiro-Wilk test with α=0.05 resulted in p_{dci}=0.01932 and p_{oo}=0.04106 for correctness, and p_{dci}=0.00002 and p_{oo}=0.00754 for time consumption, which allows to reject the normality hypothesis.

Table II Mann-Whitney U test - Overall ranks

Group	N	Median	Rank sum	95% C.I.	p-value
DCI-Trygve	42	65	2096.5	[2.9, 15.0]	0.0052
OO-Java	42	55	1473.5		

We then follow the same approach as in the overall rank analysis for correctness to include break down results for each system and for each type of comprehension task separately.

In Figure 4b, we show the descriptive statistics for the different systems. In Table III, we show the results of the statistical test for each system example. We find that H_0 can be rejected for the "Library" and for the "Spellchecker" systems with high significance ($p < 0.05$, two-tailed). In all cases, the rank sum of DCI-Trygve is greater than OO-Java, letting the DCI-Trygve as the dominant group.

Table III Mann-Whitney U test - Correctness ranks per system

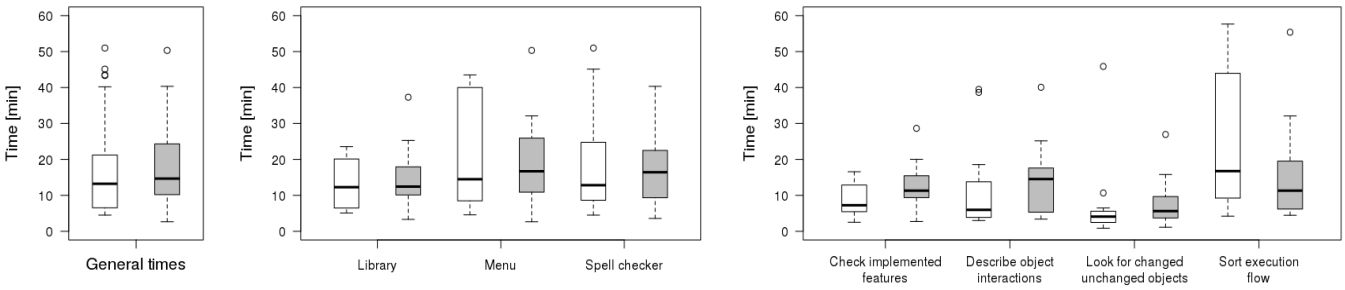
System example	Group	Obs.	Median	Rank sum	95% C.I.	p-value
Library	DCI-Trygve	14	95.0	245.5	[5e-05, 25.0]	0.0489
	OO-Java	14	72.5	160.5		
Menu	DCI-Trygve	14	60.53	221.5	[-5.3, 15.8]	0.3887
	OO-Java	14	55.26	184.5		
Spell checker	DCI-Trygve	14	65	276.5	[5.0, 20.0]	0.0005
	OO-Java	14	55	129.5		

Finally, Figure 4c shows the descriptive statistics for the correctness analysis for different type of comprehension tasks. For all tasks, the DCI-Trygve approach obtained greater rank sums than the OO-Java approach. The Mann-Whitney U test results show that we can reject the null hypothesis with high confidence ($p < 0.05$, two-tailed) only for "Look for changed/unchanged objects" and "Sort execution flow" tasks.

Table IV Mann-Whitney U test - Correctness ranks per task

Task type	Group	Obs.	Median	Rank sum	95% C.I.	p-value
Check impl. features	DCI-Trygve	14	83.33	223.0	[-5.6, 16.7]	0.3504
	OO-Java	14	75.00	183.0		
Describe obj. interactions	DCI-Trygve	14	83.33	204.0	[-6.7, 6.7]	0.9626
	OO-Java	14	86.67	202.0		
Look changed-unchanged o.	DCI-Trygve	14	72.22	261.5	[1.3e-05, 22.2]	0.0048
	OO-Java	14	61.11	144.5		
Sort execution flow	DCI-Trygve	14	50.00	252.5	[3.7e-05, 35.3]	0.0219
	OO-Java	14	23.53	153.5		

In summary, we have shown with high statistical significance that subjects in the DCI-Trygve group performed better in general, i.e. for accumulated scores across systems and



(a) Overall times **(b)** Times per system **(c)** Times per task

Figure 5 Box plots time consumption analysis (DCI-Trygve in white, OO-java in gray)

tasks. The break down analysis concluded that, DCI-Trygve participants performed statistically better in 2 out of 3 system examples, and in 2 out of 4 comprehension tasks. In other cases, we found no statistical difference between groups.

B. Time consumption

This subsection is dedicated to investigate the results concerning RQ2 related to time consumption, i.e. if there is an approach that helps programmers to understand code faster. We followed a similar approach as for the correctness analysis; we started with the overall time consumption, i.e. for all tasks and all systems, then we analyzed the time particularized for each system example, and finally for each type of task.

Figure 5a presents the descriptive statistics for the general case. The data distribution is subtly skewed in favor of DCI-Trygve approach and the rank sum also supports that subjects in this group consumed less time. Although, the Mann-Whitney U test ($\alpha = 0.05$, two-tailed) results in Table V shows no statistically significant difference between groups[9].

Table V Mann-Whitney U test for time consumption

Group	N	Median	Rank sum	95% C.I.	p-value
DCI-Trygve	42	13.22	1726.0	[-5.2, 2.9]	0.6007
OO-Java	42	14.67	1844.0		

Figure 5b and Table VI show the results considering each system example separately, and in Figure 5c and in Table VII, we present the results particularized for each type of task. For these cases, the Mann-Whitney U test ($\alpha = 0.05$, two-tailed) shows no statistically significant differences between groups.

Table VI Mann-Whitney U test for time consumption per system

System example	Group	Obs.	Median	Rank sum	95% C.I.	p-value
Library	DCI-Trygve	14	12.28	198.0	[-6.4, 5.6]	0.8362
	OO-Java	14	12.44	208.0		
Menu	DCI-Trygve	14	14.49	201.0	[-10.2, 11.4]	0.9451
	OO-Java	14	16.70	205.0		
Spell checker	DCI-Trygve	14	12.84	196.0	[-8.9, 7.1]	0.7652
	OO-Java	14	16.44	210.0		

Finally, the results particularized for type of task: Figure 5c presents the descriptive statistics and Table VII the test results.

[9]We have also restricted the time analysis in the general case to only consider times when the correctness level was greater or equal to 75%, to find if higher scores are the consequence of a higher time consumption. We have observed no statistically significant difference when running the Mann-Whitney U test ($\alpha = 0.05, two-tailed$) obtaining a $p-value = 0.1138$.

Table VII Mann-Whitney U test - Time consumption per task

Task type	Group	Obs.	Median	Rank sum	95% C.I.	p-value
Check impl. features	DCI-Trygve	14	7.25	168.0	[-8.0, 1.2]	0.1130
	OO-Java	14	11.32	238.0		
Describe obj. interactions	DCI-Trygve	14	5.97	175.0	[-11.1, 1.5]	0.2064
	OO-Java	14	14.54	231.0		
Look changed-unchanged o.	DCI-Trygve	14	4.09	183.0	[-5.1, 1.6]	0.3703
	OO-Java	14	5.62	223.0		
Sort execution flow	DCI-Trygve	14	16.74	227.0	[-3.5, 18.1]	0.2802
	OO-Java	14	11.30	179.0		

In summary, we have shown that we were not able to reject the null hypothesis with high statistical significance in any analyzed case. This indicates that there are no statistical differences regarding reading time consumption between DCI-Trygve approach and OO-Java approach when comprehending source code.

C. Locality of reference

In this subsection we present the results for the RQ3 about the locality of reference during the program comprehension process while understanding a system by reading its code.

We computed the centrality degrees and scaled them from 0 to 1 by dividing each degree by the max degree obtained in a node within its network. This normalization is important as the number of files in a system can vary between approaches. Also, we used the percentage of reading time spent on files considering the overall time spent on the entire system. Now, we show the analysis for one of the systems (Menu system) for both approaches, and then, for space reasons, we only present the results for the rest of the systems.

Figure 6 shows all the measures obtained for the Menu System for both groups. Each file in the system has its centrality median drawn in red (filled triangle), and in light-red (empty triangle) are all its centrality values. For the median of time consumption, we used blue (filled circle), and in light blue (empty circle) is each subject's reading time consumption. Note that all measures are obtained for each file in the system, and each file is sorted in the figure by its median degree of centrality, i.e. from an ascending degree of importance.

For each file, we computed the range of centrality degrees which are likely to include the population median. We used the one-sample Wilcoxon signed rank test ($\alpha = 0.01$, one-tailed) to compare the population median to a hypothesized median value to find a segment of centrality measures for each file, i.e. the possible values of centrality for the population

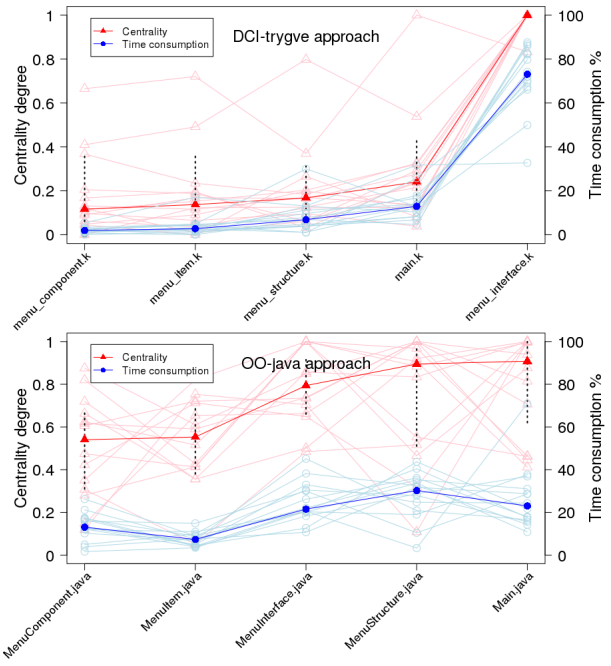


Figure 6 Centrality and Time - Menu system.

with a 99% of confidence. To get the lower end point of the centrality segment, we stated that the null hypothesis is H_o : $\mu \leq x$ and the alternative hypothesis $H_a : \mu > x$. Then, we hypothesized a median value starting from $x = 1.00$ (max centrality degree) and decreased its value iteratively by 0.01 until we set the lower end point of the segment in order to reject H_0, otherwise we set it to 0. To get the upper end point of the segment, we proceeded inversely. If we failed to reject the null hypothesis when computing the upper bound of the centrality segment, it means that the upper bound surpasses 1. As the centrality metric is scaled from 0 to 1, we take 1 as the upper bound. We wrote NA, i.e. not applicable, whenever all values of the sample had the same value, which is the case in the DCI-Trygve approach for the file with the declaration of the Context, when its centrality value obtained is 1 for all subjects of the group. We drew this segment as a vertical black dashed line in Figure 6. Table VIII and IX show the numerical values for these segment end points with its correspondent p-value for DCI-Trygve and OO-Java respectively.

Table VIII Centrality segments - Menu system DCI-Trygve

File	Centrality median	End points	p-value
menu_structure.k	0.116	0.06 0.36	0.0066 0.0078
menu_component.k	0.137	0.06 0.37	0.0055 0.0066
menu_item.k	0.167	0.09 0.32	0.0046 0.0091
main.k	0.241	0.13 0.44	0.0066 0.0093
menu_interface.k	1	0.99 1	0.0038 NA

We want to find language elements with high centrality degree that represent files that programmers consider important to comprehend a system. We consider class intervals of 0.2 units of centrality degree to study the frequency of language

Table IX Centrality segments - Menu system OO-Java

File	Centrality median	End points	p-value
MenuComponent.java	0.540	0.31 0.68	0.0093 0.0093
MenuItem.java	0.553	0.38 0.69	0.0066 0.0078
MenuStructure.java	0.794	0.66 0.93	0.0091 0.0091
MenuInterface.java	0.895	0.51 0.97	0.0091 0.0077
Main.java	0.908	0.62 1	0.0090 0.0038

elements regarding its segment of centrality degree. That way, we put in evidence if there is a node of a higher valued class that is unique, and if it is accompanied by a gap of occurrences in subsequent lower classes or not. To know if central nodes work as a locality set for program comprehension, we first correlate reading times with the centrality degree values.

Table X Centrality-Time correlation

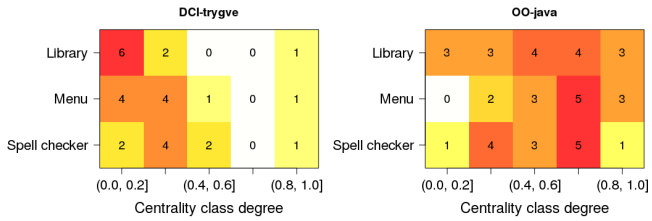
System example	Approach	Pearson (r^2)	Spearman (ρ)
Library	DCI-Trygve	0.921	0.776
	OO-java	0.814	0.903
Menu	DCI-Trygve	0.899	0.78
	OO-java	0.707	0.753
Spell checker	DCI-Trygve	0.867	0.771
	OO-java	0.817	0.787

From Table X, we can see that reading times are strongly correlated to the centrality degrees. As the time is measured after the programmer's decision to read a file, we can say that when a node is more central, more reading time is spent on that file. This is applicable for both paradigms, and it seems to reflect a mindful way of comprehending a system.

Finally, to answer RQ3, we proceed by counting the number of occurrences of centrality segments of files in the centrality class intervals. As systems may have different numbers of files, we considered an equal number of the most central ones for this count. Figure 7 shows the count of centrality segments mapping the class intervals of centrality degree. Each cell has the number of occurrences of files with the centrality class demarcated in the x axis regarding its centrality segment. Each row indicates which system example the count belongs to. A white color indicates the absence of files or language element declarations within a centrality class degree, and contiguous colors of red saturation show higher file occurrences. We can observe that in the highest centrality class, i.e. $(0.8, 1.0]$, the DCI-Trygve approach has a unique central file in all systems, and it is always followed by a gap in the second highest centrality class $(0.6, 0.8]$. In the OO-Java approach, the highest centrality class is shared from 1 up to 3 files and is followed by no gaps in its subsequent lower centrality class.

Regarding the identification of the most central element, we obtained that in the DCI-Trygve approach 97.1% of the time (41 out of 42) programmers selected the same file as the most central one, and this file was the *Context*. In OO-Java, programmers chose the same file 61.9% of the time (26 out of 42) as the most central one. We can conclude that DCI-Trygve has a most identifiable central language element in the cognitive network of language elements (graph) compared to OO-Java. As in both approaches the centrality

Figure 7 Raster plot - File count - Systems and centrality degree



degree is strongly correlated with the reading time, we can state that the *locality of reference* for the process of program comprehension is improved in DCI-Trygve avoiding context switching among language elements and letting the Context declaration work like a *locality set* where programmers choose to spend extended time intervals of reading to comprehend a system.

V. Threats to validity

Construct validity: Our approach to measure program understanding required careful selection of systems and formulation of questions. To mitigate this threat, we formulated questions focused on four categories related to understanding the runtime behavior of software and object interactions. Regarding the validity and reliability of measurements, we have correct or incorrect answers, reading time consumption, and measures related to the pattern of reading. Correctness classification is, at some level, based on human judgment, we addressed this threat by reviewing the source code, questionnaires, and correct answers with professionals and faculties. The other two measurements are more related to the mechanism designed to gather the data included in the web application used to run the experiment. We tested the application and ran two small pilots that were useful to correct the web application, improve the understanding of tasks, and the code presented to subjects. Additionally, the 'don't know' option and the preparation of subjects on its awareness helps to avoid the confounding factor of guessing in the construction of their scores.

The representativeness of the system examples is important to reflect the paradigms we are comparing. All DCI-Trygve system versions were written by professionals that participate in developing the Trygve language. The equivalent Java versions were reviewed by faculties and professionals to check and maintain its Object-Oriented representativeness. We use the same representative names for all identifiers following the conventions related to each paradigm. We did not perform renaming to make less obvious an answer, we aimed to show code with good programming practices.

Internal validity: We cannot disregard the maturity of Object-Oriented compared to Data Context Interaction paradigm. Training subjects in DCI-trygve approach was required to balance the skill levels between groups. The skill level assessed in the pre-test about the paradigm-language knowledge was similar for both groups (Table XI). Paradigm shifts take years and introducing subjects to a new paradigm takes time. As this threat was not fully removed, DCI-trygve approach was put in an unfavorable situation over OO-java.

Table XI Descriptive statistics - Groups skills

Group	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
DCI-trygve	5.00	8.00	9.00	8.86	10.00	11.00
OO-java	7.00	8.00	9.00	8.64	9.00	11.00

We performed a Spearman correlation on the skill of subjects and their scores obtained for each system. Results are similar in both groups ($p-value_{oo} = 0.002, \rho_{oo} = 0.471, p-value_{dci} = 0.003, \rho_{dci} = 0.441$). A positive estimate shows a healthy relationship, i.e. more knowledge, better scores.

The subjects in each group had similar programming experience, this was controlled during the pairing and random selection of the subjects. We created a simple heuristic to objectively compare the programming experience in subjects, then, we selected pairs with similar experience and randomly separated them into different groups. We end with the overall group experience slightly favorable to OO-java group.

The instrumentation threat is part of the internal validity. Both groups used the same instruments, and they got familiar with the interfaces of the instrument prior to the experiments in the warm up tasks. Therefore, this threat was removed. As our experiment was run over Internet, and subjects could finish the experiment at their own pace, this helped to reduce the Hawthorne effect by making participants feel like they are not in an experiment [25].

The experiment duration could influence the results, as subjects mature over the course of the experiment, they can get bored or tired, etc. The average time given to subjects in experiments related to code comprehension is 2.1 hs, and there is a large variance in the duration that seems to be independent of the type of task performed [26]. Our experiment had no time limit and subjects ran it at their own pace. It took on average 51.18 minutes, and all tasks were presented randomly to dissipate the learning effect between unitary runs.

External validity: Using a convenience sample makes it hard to generalize the results obtained, but as shown in [26], this is suitable for exploratory research. To reduce this threat related to inclusion of wrong participants, we performed a skill test right before the start of the experiment. We removed 3 subjects from the study which their skill levels were lower than expected showing that fundamental concepts of the paradigm was not understood. We also had to exclude a highly experienced subject with no pair in the other group. In the samples we had 61% of professionals and 39% of master students.

The representativeness of the code for the systems in each paradigm also plays a role in the external validity of the results. To mitigate this threat, the code for our system examples have been developed by external developers and peer-reviewed to represent the major characteristics of both approaches. Another inconvenience is that we are not using industrial scale systems. Although, our system examples surpass the average SLOC used in this type of program comprehension studies, we think that larger systems would have been beneficial to observe more clear the properties of DCI, as stated in [27].

Statistical conclusion validity: We could not make assumptions on the distributions of the data (normality) collected,

hence the chosen statistical non-parametric tests used. We also relied on medians instead of means, because in this situation, medians better represent the central tendency of the data.

We observed that reading time consumption is consistently lower in the DCI-Trygve approach, but yet not enough to consider it a statistically significant difference. There is only one case the rank sum obtained from the Mann-Whitney U test is lower in the OO-Java group. This tells us that the dominant approach in consuming less time while reading code for program comprehension is the DCI-Trygve approach. This could mean that the statistical results have traces of Type II error that could have been mitigated if sample size were larger. On the other hand, partitioning the analysis by systems or tasks produces a loss in statistical power that undermines the reliability of the results. This is reflected in the confident intervals and its closeness to zero.

VI. Related work

A. Experimentation on program comprehension

There are different kinds of controlled experiments on program comprehension that use human subjects. Their main motivations are the evaluation of new languages, design or programming techniques [14], [28], [29]; the study of paradigm shifts [28], [30]; the verification of suspected inconveniences found in the practice [13]; or the study of the effects of expertise [29], [31]. These motivations deeply behooves software maintainers, but there are also studies dedicated to find a mental model to understand the cognitive process of program comprehension [10], [32]–[34]. Usually investigations follow multiple motivations, as our, that is focused on the evaluation of two languages from different paradigms and on the verification of inherent problems of OO paradigm.

We studied the reading of source code at system level, not on small code snippets that fit in one page. We consider code reading a different skill than code writing, where design skills matter and a deeper assimilation of the paradigm is required. In that regard, we agree with [14]. Comprehension questions are the pragmatic way to get access to the results of the cognitive process of program comprehension. The counting or percentage of correct questions is used in the majority of research works to measure the level of comprehension. We consider only the reading time because it is tied to the organization of code governed by the paradigm in turn. One of our goals is to help in reducing the maintenance time, where the time spent reading old code surpasses 10 to 1 the time spent to write new code [35]. Measuring time is common in program comprehension studies, as in [13], [14], and it is also used with a strict limit as in [31], [34], [34]. The number of file switches is used in [12], [30], but we also capture the sequencing of reading in a multigraph to compute the importance of files according to each reader to study the locality principle in program comprehension.

B. Classical Object-Oriented inherent problems

The literature shows evidence of issues regarding program comprehension that repeatedly appears in OO systems. At the time OO was growing fast, Burkhardt et al. were trying to find a program comprehension mental model for this paradigm. Pennington's model (1987) [32] presented some limitations regarding its application in large sized and OO programs. Her model did not account for "representation of delocalized plans and the representation of text macrostructure" [10]. Letovsky & Soloway, 1986, had already observed that programs with delocalized plans were difficult to understand. A delocalized plan is one in which a plan's actions are widely separated in the program structure [36]. This definition of delocalized plan is what we know as a system functionality, that in DCI is located in one place within the *context* declaration where the programmer can read the denotations of other needed declarations and give objects semantic meaning for the system functionality in course. Object orientation relies on the use of classes as the main building blocks to build systems [37], where data and behavior for small units live together. This causes the delocalization of system operations and the definition of very small sized methods that define the behavior of systems. If using class hierarchies, "a trace has to be made through the object hierarchy, tracing messages until you reach the method where the work is done" [38]. Dynamic binding provides flexibility, but at the same time hinders the possibilities to precisely identify dependencies in the system through a static analysis of the code [39] [38]. Then, a dynamic analysis is required to understand the system, i.e. a necessity to add more tests. In our work we study the needs of contexts (implicit in OO and explicit in DCI) in the source code regarding system functionalities understandability.

VII. Conclusion

The correctness analysis shows that programmers in the DCI group performed better than programmers in the OO-java group. This is aligned with the theories behind DCI and it should encourage further and systematic investigations concerning the possibilities of this paradigm.

We were not able to find which approach helps to reduce reading time consumption, therefore timing remains an open issue that might find an answer if using greater samples.

We found that programmers in the DCI group tend to refer repeatedly and over extended time intervals to *Contexts* rather than to the remaining declarations in the code. This is aligned to the well understood *locality of reference principle*. We observed that Contexts acts like a locality as programmers develop understanding of system functionalities or use cases. On the other hand, the context-switching of the OO-java group was aggravated by the delocalized plans for a system functionality. Therefore, DCI-trygve approach improves the locality of reference in program comprehension process.

Acknowledgment

The authors would like to thank Trygve Reenskaug for his innovative efforts that were the foundation for DCI. This work was partially funded by the US National Science Foundation under grant numbers CCF-1543176.

REFERENCES

[1] "A personal computer for children of all ages," *ACM National Conference*, August 1972, history-computer.com/Library/Kay72.pdf.

[2] B. L. Cox, "The object oriented pre-compiler: programming smalltalk 80 methods in c language," *ACM Sigplan Notices*, vol. 18, no. 1, pp. 15–22, 1983.

[3] B. Stroustrup, *The C++ Programming Language (1st Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[4] S. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1988.

[5] T. Reenskaug and J. O. Coplien, *The DCI Paradigm: Taking Object Orientation into the Architecture World*, 2012, http://fulloo.info/Documents/CoplienReenskaugASA2012.pdf.

[6] ——, "The dci architecture: A new vision of object-oriented programming," *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

[7] S. Metz, *Practical Object-Oriented Design in Ruby: An Agile Primer*. Pearson Education, 2012.

[8] R. Conradi and A. I. Wang, *Empirical Methods and Studies in Software Engineering: Experiences from Esernet*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[9] J. Quante, "Do dynamic object process graphs support program understanding? - a controlled experiment." in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 73–82.

[10] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Mental representations constructed by experts and novices in object-oriented program comprehension," in *Human-Computer Interaction INTERACT'97*. Springer, 1997, pp. 339–346.

[11] C. L. Corritore and S. Wiedenbeck, "Mental representations of expert procedural and object-oriented programmers in a software maintenance task," *International Journal of Human-Computer Studies*, vol. 50, no. 1, pp. 61–83, 1999.

[12] ——, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *International Journal of Human-Computer Studies*, vol. 54, no. 1, pp. 1–23, 2001.

[13] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 181–190.

[14] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini, "An empirical study on program comprehension with reactive programming," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 564–575.

[15] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 361–370.

[16] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, "Exploring software measures to assess program comprehension," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Sept 2011, pp. 127–136.

[17] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon, "A study of student strategies for the corrective maintenance of concurrent software," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 759–768. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368195

[18] R. F. Burton and D. J. Miller, "Statistical modelling of multiple-choice and true/false tests: ways of considering, and of reducing, the uncertainties attributable to guessing," *Assessment & Evaluation in Higher Education*, vol. 24, no. 4, pp. 399–411, 1999.

[19] H. van Mameren, C. van der Vleuten *et al.*, "The effect of a 'don't know'option on test scores: number-right and formula scoring compared," *Medical Education*, vol. 33, no. 4, pp. 267–275, 1999.

[20] R. F. Burton, "Quantifying the effects of chance in multiple choice and true/false tests: question selection and guessing of answers," *Assessment & Evaluation in Higher Education*, vol. 26, no. 1, pp. 41–50, 2001.

[21] P. J. Denning, "The locality principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[22] R. Zafarani, M. A. Abbasi, and H. Liu, *Social media mining: an introduction*. Cambridge University Press, 2014.

[23] R. Schumacker and S. Tomek, *Understanding statistics using R*. Springer Science & Business Media, 2013.

[24] B. Boehm, H. D. Rombach, and M. V. Zelkowitz, *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer Science & Business Media, 2005.

[25] J. Siegmund, "Framework for measuring program comprehension," Ph.D. dissertation, Magdeburg, Universität, Diss., 2012, 2012.

[26] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 9, pp. 733–753, 2005.

[27] J. O. Coplien and G. Bjørnvig, *Lean architecture: for agile software development*. John Wiley & Sons, 2011.

[28] A. Lee and N. Pennington, "The effects of paradigm on cognitive activities in design," *International Journal of Human-Computer Studies*, vol. 40, no. 4, pp. 577–601, 1994.

[29] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. Corritore, "A comparison of the comprehension of object-oriented and procedural programs by novice programmers," *Interacting with Computers*, vol. 11, no. 3, pp. 255–282, 1999.

[30] R. J. Walker, E. L. Baniassad, and G. C. Murphy, "An initial assessment of aspect-oriented programming," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE, 1999, pp. 120–130.

[31] V. Ramalingam and S. Wiedenbeck, "An empirical study of novice program comprehension in the imperative and object-oriented styles," in *Papers presented at the seventh workshop on Empirical studies of programmers*. ACM, 1997, pp. 124–139.

[32] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive psychology*, vol. 19, no. 3, pp. 295–341, 1987.

[33] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[34] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[35] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[36] R. S. Rist, "Program structure and design," *Cognitive Science*, vol. 19, no. 4, pp. 507–562, 1995.

[37] J. O. Coplien, "Objects of the people, by the people, and for the people," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*. ACM, 2012, pp. 3–4.

[38] A. Dunsmore, "Comprehension and visualisation of object-oriented code for inspections," *Empirical Foundations of Computer Science (EFoCS), University of Strathclyde Livingstone Tower, Glasgow G1 1XH, UK*, 1998.

[39] N. Wilde and R. Huitt, "Maintenance support for object oriented programs," in *Software Maintenance, 1991., Proceedings. Conference on*. IEEE, 1991, pp. 162–170.