An empirical study on code comprehension: DCI compared to OO

Hector A. Valdecantos
hav2082@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
An empirical study on code comprehension: DCI compared to OO

Author: Héctor A. Valdecantos
Supervisor: Dr. Mehdi Mirakhori

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering in the

B. Thomas Golisano College of Computing and Information Sciences
Department of Software Engineering
Rochester, New York

August 3, 2016
The thesis “An empirical study on code comprehension: DCI compared to OO” by Héctor A. VALDECHANTOS, has been examined and approved by the following Examination Committee:

Dr. Mehdi Mirakhorli  
Thesis Committee Chair  
Assistant Professor

Dr. Meiyappan Nagappan  
Assistant Professor

Dr. Scott Hawker  
SE Graduate Program Director  
Associate Professor
ROCHESTER INSTITUTE OF TECHNOLOGY

Abstract

B. Thomas Golisano College of Computing and Information Sciences
Department of Software Engineering
Rochester, New York

Master of Science in Software Engineering

An empirical study on code comprehension: DCI compared to OO

by Héctor A. Valdecanos

Comprehension of source code affects software development, especially its maintenance where reading code is the most time consuming performed activity. A programming paradigm imposes a style of arranging the source code that is aligned with a way of thinking toward a computable solution. Then, a programming paradigm with a programming language represents an important factor for source code comprehension. Object-Oriented (OO) is the dominant paradigm today. Although, it was criticized from its beginning and recently an alternative has been proposed. In an OO source code, system functions cannot escape outside the definition of classes and their descriptions live inside multiple class declarations. This results in an obfuscated code, a lost sense the run-time, and in a lack of global knowledge that weaken the understandability of the source code at system level. A new paradigm is emerging to address these and other OO issues, this is the Data Context Interaction (DCI) paradigm. We conducted the first human subject related controlled experiment to evaluate the effects of DCI on code comprehension compared to OO. We looked for correctness, time consumption, and focus of attention during comprehension tasks. We also present a novel approach using metrics from Social Network Analysis to analyze what we call the Cognitive Network of Language Elements (CNLE) that is built by programmers while comprehending a system. We consider this approach useful to understand source code properties uncovered from code reading cognitive tasks. The results obtained are preliminary in nature but indicate that DCI-trygve approach produces more comprehensible source code and promotes a stronger focus the attention in important files when programmers are reading code during program comprehension. Regarding reading time spent on files, we were not able to indicate with statistical significance which approach allows programmers to consume less time.
Acknowledgements

Thanks to the object-composition group, the community behind DCI, that helped me in the understanding of the DCI paradigm and trygve language, specially thanks to Matthew Browne, Andreas Söderlund, Trygve Mikkjel Heyerdahl Reenskaug, and Rune Funch Søltoft. Thanks to James O. Coplien for his collaboration in the first part of this investigation and for playing an important role that favored the direction of my line of study to something that I like. Thanks to Mehdi Mirakhorli to accept being my advisor and support the realization of this thesis. Thanks to Mehdi’s research group, specially Palak Sharma, Raghuram Gopalakrishnan, and Danielle Gonzales. Thanks to Ibrahim Mujhid for helping with the java versions, and to my friend and professional colleague Alexis Sarghel to review the questions and to set the server where our web application was deployed to run the experiment. Thanks to José Vargas, a friend and former colleague and IT professional, to Hernán Wilkinson, co-founder of 10Pines, and Darío Korec from Asof Argentina to help in the recruitment for the experiment.

I am grateful for the funding received through the Argentine Presidential Fellowship in Science and Technology and the Argentina Fulbright Commission that made possible the completion of my master degree. I thank the Software Engineering department’s Graduate Scholarship Committee at RIT for selecting me to receive an RIT Graduate Scholarship to support my graduate studies.
Contents

Abstract ii
Acknowledgements iii

1 Introduction 1
  1.1 Context ................................................. 1
  1.2 Thesis organization ................................. 3

2 Motivation 5
  2.1 State of the problem .................................. 5
    2.1.1 Classical Object-oriented inherent problems .... 7
    2.1.2 Maintenance ...................................... 9
  2.2 Research objectives .................................. 11
  2.3 Research questions .................................. 12

3 Data, contexts, and interactions 14
  3.1 DCI paradigm ......................................... 14
  3.2 What the system is .................................... 15
  3.3 What the system does ................................ 16
  3.4 The computational metaphor of theater .......... 17
  3.5 End user mental model ............................... 18
  3.6 The trygve language ................................ 19
    3.6.1 Trygve program structure ...................... 21
    3.6.2 Classes and interfaces ......................... 22
    3.6.3 Contexts and roles ............................. 22
    3.6.4 Code example explained ....................... 23

4 Related work 28
  4.1 Program comprehension experimentation ........... 28
  4.2 A mental model for program comprehension ....... 32

5 Research approach 36
  5.1 Scope ................................................. 36
  5.2 Experimental design .................................. 37
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Experiment time-line design</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Experimental parameters and variables</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Participants</td>
<td>41</td>
</tr>
<tr>
<td>5.6</td>
<td>Instrumentation</td>
<td>42</td>
</tr>
<tr>
<td>5.7</td>
<td>Experimental units</td>
<td>43</td>
</tr>
<tr>
<td>5.7.1</td>
<td>An experimental unit centric model perspective</td>
<td>45</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Code equivalence</td>
<td>46</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Descriptive metrics of system examples</td>
<td>47</td>
</tr>
<tr>
<td>5.8</td>
<td>Experimental tasks</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Analysis and results</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>Correctness analysis</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Timing analysis</td>
<td>54</td>
</tr>
<tr>
<td>6.4</td>
<td>Centrality analysis</td>
<td>57</td>
</tr>
<tr>
<td>7</td>
<td>Discussion and Conclusion</td>
<td>63</td>
</tr>
<tr>
<td>7.1</td>
<td>Threats to validity</td>
<td>63</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Conclusion validity</td>
<td>63</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Internal validity</td>
<td>64</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Construct validity</td>
<td>65</td>
</tr>
<tr>
<td>7.1.4</td>
<td>External validity</td>
<td>65</td>
</tr>
<tr>
<td>7.2</td>
<td>Conclusion</td>
<td>66</td>
</tr>
<tr>
<td>7.3</td>
<td>Future work</td>
<td>69</td>
</tr>
<tr>
<td>A</td>
<td>Supporting data</td>
<td>70</td>
</tr>
<tr>
<td>A.1</td>
<td>Code metrics</td>
<td>70</td>
</tr>
<tr>
<td>A.2</td>
<td>Experimental tasks metrics</td>
<td>70</td>
</tr>
<tr>
<td>A.3</td>
<td>Centrality and time</td>
<td>71</td>
</tr>
<tr>
<td>Bibliography</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

2.1 A execution of system functionality as shown in (Dunsmore, 1998, p. 21). 8
2.2 Activities to find code rationale as shown in (LaToza, Venolia, and De-Line, 2006). 10

3.1 Software system entities interactions diagram. 19

5.1 Experiment time-line design 38
5.2 Web application display code 43
5.3 Web application displaying the question 44
5.4 Generic unit centric model perspective 45
5.5 Unit centric model perspective, example 46
5.6 Unit centric model perspective view 46

6.1 Box plot of correctness ranks 51
6.2 Box plot of correctness ranks per system example 52
6.3 Box plot of correctness ranks per task type 53
6.4 Box plot of timing scores 54
6.5 Box plot of time consumption per system example 55
6.6 Box plot of time consumption per task type 56
6.7 CNLE for Menu system - DCI-trygve approach 58
6.8 Centrality and Time - Library system 59
6.9 Histogram of centrality classes - Library system 61
6.10 Raster plot - File count over systems and centrality degree classes 62

A.1 Centrality-Time analysis Bank system 71
A.2 Centrality-Time analysis Library system 72
A.3 Centrality-Time analysis Menu system 73
A.4 Centrality-Time analysis Spell Checker system 74
A.5 Centrality-Time analysis Tower of H. system 75
List of Tables

3.1 DCI main separation. .................................................. 16
4.1 Consulted controlled experiments with human subjects on program comprehension ........................................... 32
5.1 Experimental variables description. ................................. 40
5.2 Java code ............................................................... 47
5.3 trygve code ............................................................. 48
6.1 Mann-Whitney U test for scores. .................................... 51
6.2 Mann-Whitney U test for scores per system. ....................... 52
6.3 Mann-Whitney U test for scores per task type. ..................... 53
6.4 Mann-Whitney U test for time consumption. ......................... 54
6.5 Mann-Whitney U test for time consumption per system. .......... 55
6.6 Mann-Whitney U test for scores per task type. ....................... 56
6.7 Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve). .................................................. 60
6.8 Wilcoxon rank signed test for centrality segments - Library system (OO-java). ...................................................... 60
6.9 Centrality-Time correlation. .......................................... 61
6.10 Centrality class count for most central files. ....................... 62
A.1 Simple code metrics per system example. ......................... 70
A.2 Experimental tasks per system example. ............................ 70
A.3 Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve). .................................................. 71
A.4 Wilcoxon rank signed test for centrality segments - Bank system (OO-java). ...................................................... 71
A.5 Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve). .................................................. 72
A.6 Wilcoxon rank signed test for centrality segments - Library system (OO-java). ...................................................... 72
A.7 Wilcoxon rank signed test for centrality segments - Menu system (DCI-trygve). .................................................. 73
A.8  Wilcoxon rank signed test for centrality segments - Menu system (OO-java). .......................................................... 73
A.9  Wilcoxon rank signed test for centrality segments - Spell checker system (DCI-trygve). .................................................... 74
A.10 Wilcoxon rank signed test for centrality segments - Spell checker system (OO-java). .......................................................... 74
A.11 Wilcoxon rank signed test for centrality segments - Spell checker system (DCI-trygve). .................................................... 75
A.12 Wilcoxon rank signed test for centrality segments - Tower of H. system (OO-java). .......................................................... 75
List of Abbreviations

CNLE  Cognitive Network of Language Elements
DCI   Data Context Interaction
OO    Object-Oriented
SLOC  Source Line Of Code
Chapter 1

Introduction

1.1 Context

Comprehension and readability of source code affect software development, especially its maintenance where reading code is a central activity. (Raymond, 1991). Furthermore, the phase that a software system stays longer throughout its evolution is the maintenance phase that consumes over 70% of the total life-cycle cost of successful software developments (Boehm and Basili, 2005). “Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code” (Martin, 2009). A computer system expressed through programming languages references all the technologies required to execute it, these are libraries, databases, frameworks, patterns, etc, but it also comprises the programming paradigms used that arranges, gives form, and organizes the source code as text to be read. Therefore, a programming paradigm imposes a style of arranging the source code that is aligned with a way of thinking toward a computable solution. In our research work we measure how Data Context Interaction (DCI) paradigm affects comprehension of source code in programmers by comparing it with classical Object-Oriented (OO) paradigm.

Software engineering concerns are more related to a process than to a product. The idea of software as a product is losing its strengths as years pass by. After all, the time scale in the series of mutations a software system suffers during its evolution are usually small enough to stop considering it a product anymore. If we see it as a product, then it is a product that changes on demands of clients, like a service. Even if a software system does not change its source code, it is still a process running in a computing machine serving solutions to end users. A software system is a process that directly or indirectly involves human interactions that symbiotically makes the system mutates. A big portion of these mutations are made by changes applied to the source code that models the running system, therefore, the source code should also be interpreted by the humans in charge of modifying the code to maintain the system, and not only by computers. Hence code comprehension is an essential activity for the welfare of the software system as a whole.
Measuring the comprehension of source code is similar to measure the comprehension of natural language texts\textsuperscript{1}, but with the addition that the source code text written by programmers is unequivocally interpreted by a computer that tyrannically shows the results of its execution as a reality presented to other humans. That way, the source code is an element of communication between the members of the development team that serves as a blueprint for a reality that has to be agreed between end users and programmers for a software system to be considered successful. With that idea in mind, the architecture of a software system only exists in the run-time and becomes an expression of a social intention that seems to keep a strong similarity to the dynamics of language in the pragmatic sense. Consequently, the source code today is not only to be executed by a computer to produce an output to end users, it is also used for communication purposes between programmers within a team that jointly describes the reality to be shown to end users, therefore the comprehension of source code is a key aspect to understand these phenomena\textsuperscript{2}. “Experimentation provides a systematic, disciplined, quantifiable and controlled way of evaluating human-based activities. This is one of the main reasons why empirical research is common in social and behavioral sciences” (Wohlin et al., 2012). Throughout the development of our research we have discovered that comprehending source code is maybe one of the activities with a high load of social and psychological aspects in software engineering, hence, comparing comprehension of source code is a very hard task to do and it is subject to unreliable results due to multiples human confounding factors.

The main topic of our research is directly related to human interactions, is about language and thinking, meaning and realities, time and space, about the constant dumping of knowledge in a computing machinery that resembles the human mind. Connecting all this high level philosophy of computing to specific metrics related to source code, comprehension questions, spent reading time and centrality degree of language elements may seen fictional but it is an starting point to understand the human part of computing. The key to link both parts is in what we understand for programming paradigms. During the experiment we give subjects tasks with code comprehension questions to count their correct answers, to measure their time of reading, and to capture how the system code is read. The data gathered from these tasks give us an insight about program comprehension, that analyzed from a software engineering point of view allows us to take distance from all psychological and social factors to study the paradigms as a set of rules applied by programmers.

Traditionally, source code comprehension prescribes measures regarding time and

\textsuperscript{1}The cognition model of how programmers comprehend source code is based on models made first for natural languages. We write more about it in Section 4.

\textsuperscript{2}In the Immanuel Kant sense, \textit{phenomenon} is an observable manifestation, in contrast to \textit{noumenon} that is not directly accessible as an observable experience.
correctness, and more recently eye-tracking techniques have been used to start observing the behavior of programmers in front of the source code. Because the characteristics of our research, we don’t need the level of detail that eye-tracking techniques give, we are looking at bigger conceptual chunks of code as language elements like classes, interfaces, or contexts that conform the blueprint of running systems. By considering the reading time a programmer spends on a file we get valuable information at system level to evaluate properties of the source code regarding the paradigm used. These measures and observations represent a method to investigate the properties of source code regarding comprehension at system level.

In this thesis we review some issues that repeatedly appears in Object-Oriented systems regarding code comprehension and we present the novel Data Context Interaction paradigm as an alternative. In our research work, we explain the design of the code comprehension experiment we have run and we report the results together with all the decisions we took and the implications and consequences they meant for the investigation.

1.2 Thesis organization

In the following item we summarize the content of each chapter:

**Chapter 2**: we present the evidence found in the literature that made us reconsider classical Object-oriented thinking. Since the introduction of OO paradigm there were detractors from other language communities. We visualize how maintenance might be affected because of the problems found in OO. Finally we show our objectives and our research questions that have motivated the investigation.

**Chapter 3**: this chapter is dedicated to the Data Context Interaction paradigm. We explain the main ideas behind DCI and the metaphor of theater for computer programs. We briefly present *trygve* language, a new DCI-centric language, showing and explaining its main language elements through an small example.

**Chapter 4**: we first present the related works regarding the experimentation that researchers have been done on program comprehension. Then, we present those related to the creation of a mental model for program comprehension based on text comprehension.

**Chapter 5**: in this chapter we explain in detail all steps taken to build our controlled experiment. We present the scope, variables, factors, experimental design, subjects, instrumentation, experimental units, etc. We also present an experimental unit centric model perspective we have created to explain how intricate might be the relationship among elements of a program comprehension experiment.
Chapter 6: we show the statistical analysis we followed to answer our research questions regarding correctness, timing, and centrality. We observe programmer’s focus of attention by analyzing the centrality or importance of files and the reading time spent during comprehension tasks.

Chapter 7: we briefly present the limitations of our investigation by explaining all the threats to validity and the boundaries of the results found. We draw our conclusions based on the results taking into account the limitations and exposing some discussions. Finally, we present the work left for future investigations and the possible ramifications that this thesis might open.
Chapter 2

Motivation

2.1 State of the problem

Since its beginning, professional programmers have been reluctant to accept some of the ideas behind Object-Oriented (OO) paradigm. It tooks a long journey to become what is now the most widely used paradigm, but, still today there is some kind of opposition or resistance to it, like a non-ending search to fully understand what Object-Oriented is about. Even though, there is a general common consensus on what this paradigm involves, but there are also so many particular variations of interpretations since its origin that speaking of OO would definitely bring some disagreement between communities of programmers. For our case, we think it is better to tie the paradigm to a language to make a more specific reference of what flavor of OO paradigm we are talking about, after all, a programming language is the language a community of programmers use as a mean to express a computer system in a specific paradigm. It might be the case that what we call OO is not what the mainstream calls OO. To avoid confusion, as we said, we combine the paradigm with the language as one term. We use OO-Java to make a reference to the paradigm thinking that can be identified as Object-Oriented as usually seen in java code. For example, Object-Oriented thinking may differ substantially from java to ruby or javascript communities. This divergent thinking might also be an evidence of the misunderstanding of what OO is, because a paradigm should transcend languages but we see that different characteristics of the language produce a different understanding of the paradigm, maybe as a Whorfian effect. Some may consider the so-called pillars of Object-Oriented to explain what this paradigm is about, by stating that it is about abstraction, encapsulation, inheritance, and polymorphism. A less technical view may consider objects passing messages withing a network of collaborating objects to explain what OO paradigm is about.

What has been around us for years as Object-Oriented programming we actually see it as class-oriented, where code describing system behavior cannot exist outside a class, blurring in this way the relationships between objects as dynamic entities because of their behavioral declaration being part of static program building blocks called
classes. In the user’s manual for the trygve language\footnote{The \textit{trygve} programming language is the first language that implements the ideas of Data Context Interaction paradigm and it is driven by a community. Trygve lenguage is open source and it is hosted in \url{https://github.com/jcoplien/trygve}. This language is implemented mostly by James O. Coplien. There are other DCI languages but seems to be individual efforts.} it is stated that “Java tried to be a pure OO language by outlawing global functions, but that is a simplistic hope at best. It ended being only a class-oriented programming language. Like most languages of its kind it has many features to finesse class relationships. These features encourage class-oriented thinking, ... So you will find neither \textit{friends}, or \textit{static} objects, or the concept of \textit{super}, nor the \textit{protected} access property in trygve” (Coplien, 2016).

A class-oriented approach has multiple undesired effects in the code, the running system, and in the software development business. Class-oriented programming overemphasized classes as the main building block for a computer system, this raises problems primarily in its design and maintenance. For example, having the state and behavior together encapsulated in one entity results in building blocks with elements that change at different rates; a system conceived only through class decomposition will end with the system intelligence or behavior scattered among classes; a class has no business value for itself inside a business perspective, it is not a deliverable as a use case is, but there is no building block that represents in the code the context were a use case or system functionality happens.

“Today’s programming languages, most of which relate to object orientation, tend to focus on data as the primary organizing structure. Classes featured heavily as the main building blocks of these programs” (Coplien, 2012)

The act of classifying is mostly static and declarative as it has been used for years in biology, botany, and zoology to explain through hierarchies of commonalities the taxonomy of species. Classes do not express time, at least not at today digital computing time scale. We can use classes to explain changes at species time scale and figure out the evolution and relationships between living or extinct organisms. It is not easy nor possible to express the run-time of a computer system using only classes in this sense. We can not pretend that by describing a class we will be able to instantiate an object to make it magically do its part within a system functionality. If this is true, there is two possibilities here: we are instantiating living things as a living organism, or we are mixing inside a class the behavioral run-time description of objects that does not belong to a class. As much as we would like, we are definitely not creating living organisms with the shape of objects. “An object-oriented program’s run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program’s run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the
other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa” (Vlissides et al., 1995, p. 22)

### 2.1.1 Classical Object-oriented inherent problems

The kind of problems that practitioners tend to encounter in an OO system have been mitigated through years of experience, but these problems seem to be inherent to some parts of the paradigm. Proponents of Object-oriented paradigm compiled valuable knowledge as in design pattern and principles, but generally this mitigation comes together with an accidental complexity. This is the case of some design patterns from which we can gain flexibility and reduce coupling between classes at the cost of losing comprehensibility of the written code and impede a manageable maintenance. An empirical study of the impact of design patterns on the reusability, understandability and expandability of source code shows that design patterns not always improve these quality attributes (Khomh and Guéhéneuc, 2008). From the software craftsmanship movement there are proponents that have organized the knowledge collected through continuous practices, for example, (Martin, 2009) and the clean code and TDD practice that result in Object-oriented code that is test-dependent as a way to counteract its lack of understandability, and it is maintenance time consuming as a result of maintaining a parallel testing system. The lesson learned through years of experience is valuable and still useful to understand how to develop software systems, but unseen root problems in the paradigm will always delude our intentions.

In OO, “Most methods are very small in size, many only a line or two, making it difficult to be able to define the behavior of a program. Because of so many small methods, to be able to understand how one line of code works in some cases, a trace has to be made through the object hierarchy, tracing messages until you reach the method where the work is done” (Dunsmore, 1998). The overuse of inheritance and polymorphism when using dynamic binding may turn a large system into a nightmare when trying to trace the execution of a system function. Dynamic binding provides flexibility, but at the same time it complicates the understanding of dependencies and hinders the possibilities to precisely identify dependencies in the system through a static analysis of the code (Wilde and Huitt, 1991) (Dunsmore, 1998). Then, a dynamic analysis is needed to know the dependencies, this can be understood as a need to add more tests. Generally, a reliable and cost effective object-oriented system will have even more than twice the amount of test code than production code. The reason why of this production:test code ratio is because “test cases may not detect all the behavior that the program is capable of exhibiting, and thus incorrect conclusions may be drawn” (Wilde and Huitt, 1991). “Test code should describe what the production code does. That means that it tends to be concrete about the values it uses as examples of what results to expect, but abstract about how the code works” (Freeman and Pryce, 2009,
p. 247). The complicated trace of execution due to inheritance and polymorphism reveals a lack of a place to find system functionalities in the code. “In object-oriented design, control is dispersed and sequencing occurs only as a result of message passed between objects and resultant behavior of objects. Thus, there is only local knowledge of objects and a lack of global knowledge of control” (Lee and Pennington, 1994, p. 5).

Abbes et al. have studied the effects on program comprehension when anti-patterns\textsuperscript{2} are lurking around the source code (Abbes et al., 2011). They have worked with Spaghetti\textsuperscript{3} and Blob\textsuperscript{4} code and they could not find a statistically significant difference when those considered negative patterns appears in the code in isolation to hinder program comprehension. “Collected data showed that the occurrence of one anti-pattern in the source code of a system does not significantly make its comprehension harder for subjects when compared to a source code without any anti-pattern.” (Abbes et al., 2011). “Surprisingly, subjects appear to perform better ... when there is an occurrence of the Spaghetti Code.” (Abbes et al., 2011) On the other hand, they did find that combination of the studied anti-patterns impacted negatively and significantly in program comprehension. These results may speaks about some properties of these called “anti-patterns” patterns that favor code comprehension, maybe these both cases studied by Abbes et al. are demanding for an identifiable space in the code to describe a system behavior.

\textsuperscript{2}The term anti-pattern seems not right if we are aware of what a pattern is. If a pattern results in something that is judged negatively we might have not seen or considered all the forces that guide that pattern to resolve in such a manner.

\textsuperscript{3}This pattern can be perceived in classes with a poor separation of concern, with long methods that uses global variables and method’s names that suggest procedural programming.

\textsuperscript{4}This pattern is observable in large and complex classes where it is centralized part of a system behavior and uses other classes as to access data. It is sometimes called God class.
Chapter 2. Motivation

After reviewing the related works we detected specific items that matched the claims stated in works done by Coplien and Reenskaugh about the problems related to OO:

- **Obfuscated code**: in object-oriented systems, when more flexible and modifiable the code is, we usually get more complex and spread implementation of system functionalities. This is observed in related works, as in (Ramalingam and Wiedenbeck, 1997) that concludes that “the OO style programs appears to have obscured operations and control flow”.

- **Data-centric design**: the design is based on the decomposition of the data, where behavior is subsumed to the form of the data. Classes not only define behavior on objects that modifies its own state, but define behavior that modifies other objects states, this degrade cohesiveness.

- **Poor traceability**: the discovery cost of knowing where to find a particular system function in code is often high in OO systems. A system functionality is often spread over multiple classes.

- **Lack of global knowledge**: we can observe a lack of locality in the code to understand a system running an operation, we have to read small pieces of code in multiple classes. Object-oriented systems bear from a lack of locality of intentionality (Coplien and Reenskaug, 2012, p. 26) because system operations are spread over object interfaces making the source code hard to understand.

- **Weak run-time understanding**: reading the execution flow in OO systems is only achievable by jumping around through different language elements that defines the system. “With such disparity between a program’s run-time and compile-time structures, it’s clear that code won’t reveal everything about how a system will work. The system’s run-time structure must be imposed more by the designer than the language. The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is” (Vlissides et al., 1995, p. 23). Objects are the main run-time building blocks, nevertheless in classical OO systems we program behavioral interaction between objects within classes.

2.1.2 Maintenance

In the evolution of a successful software project, maintenance take the big portion of time dedicated to a system. Maintenance refers to activities that take place at any time

---

5For a deep explanation about where OO fails we recommend reading (Coplien and Reenskaug, 2012) and (Coplien and Bjørnvig, 2011)
after the newly development project is implemented and deployed, this includes: fix bugs, code enhancements, changes to perform old features under new conditions, and new feature additions. This activities can be included in the classical categorization of corrective, adaptive, and perfective that were used and expanded in (Kemerer and Slaughter, 1999). In a IEEE feature article by A. Mayrhauser and A. Marie Vans, authors use a similar categorization of activities associated with software maintenance: adaptive, perfective, and corrective, plus reuse and code leverage. They show that these categories require different aspects of code understanding, and some of the activities within a maintenance task, such as “understand the system”, are common to several tasks (Von Mayrhauser and Vans, 1995). It is estimated that more than 50% of all professional programmer’s time is spent on program maintenance tasks that involve modifications and updates of previously written programs. Because the programs are most often written by other programmers, comprehension plays a central role in this endeavor (Pennington, 1987b).

Developers also look other sources to understand the code, but the source code is indeed the first resource to find the rationale behind it. This tells us that not all information is adequately expressed in the code. In a survey report is stated that “when investigating a piece of code, developers turn first to the code itself: on average respondents spent 42% (± 29%) of their understanding time examining the source code, 20% (± 17%) using the debugger, 16% (± 19%) examining check-in comments or version diffs, 9% (± 10%) examining the results, 8% (± 12%) using debug or trace statements, and 3% (± 14%) using other means. In other words, the code itself is the best source of information about the code. However it is not flawless. Developers commonly become disoriented in unfamiliar source code, and discerning the relationship between observed program behavior and the source code is often difficult” (LaToza, Venolia, and DeLine, 2006).

![Figure 2.2: Activities to find code rationale as shown in (LaToza, Venolia, and DeLine, 2006).]
A trend in OO language communities that impedes maintenance regarding state and behavior unification is noted by Coplien and Reenskaug: “Class-oriented programming puts both data evolution and method evolution in the same shear layer: the class. Data tends to remain fairly stable over time while methods change regularly to support new services and system operations. The tension in these rates of change stresses the design” (Coplien and Reenskaug, 2012).

2.2 Research objectives

We want to find measurable results related to source code comprehension and give an empirical answer to the claims made by DCI proponents. Our research will help software developers in selecting an alternative way to construct software in order to reduce costs in maintenance. By measuring the comprehension level of DCI-tryge and OO-java source code and detecting where the focus of attention resides when programmers read code, we will be also testing some of the claimed benefits of DCI approach to counteract some of the above mentioned disadvantages of traditional OO approach.

A lot of thinking have been around the history of computer and programming languages. One persistent tendency is that we, programmers as humans, have been dumping knowledge into computer programs to delegate part of our mental operations. We went first from well defined mathematical operations stored in mechanical gears, then, to programs run by people switching plugs between modules of computable semantic to achieve results from repeatable operations, up to storing the whole program instruction into the machine itself. Nowadays software systems are complex and developed by a group of people that configure a culture within the team, and more and more we are dumping human related information in our computer programs. What we have started to do is introducing the end user thinking into the program itself. A long time has passed since end users are not computer engineers or mathematicians writing their own programs, today computers are used mostly by people unaware of how to program or create software. Due to the complexity of software systems required today, we need the end user thinking reflected in a more direct way in the code as a part of the computation to be performed if we want to achieve expectable results.

When ideas are made public and they are perceived as new ideas and are considered disruptive from the way most people usually formulate thoughts or opinions, and at the same time it is possible to foresee some benefit but it is not clear how to take advantage of them, we can say that we are in front of a new paradigm. In our small research work we want to spread the ideas behind Data Context Interactions paradigm. This is a small task compared to the long way yet to go. Because we also
have to cross the boundaries of established way of thinking and try to foresee the con-
sequences of changing direction. This is a way to hurry up changes that have been
naturally happening in the history of programming languages. With this work we are
accelerating the process of adoption or rejection of these new ideas by presenting an
empirical study to compare a well established way of programming, like OO using
java language, with a the new DCI paradigm using trygve language.

Code comprehension is usually not taken part of the curricula of computer science
of software engineering in undergraduate degrees. Reading large system is not like
reading an algorithm that usually keeps in an entire page. Furthermore, reading and
comprehending a software system that is spread out in multiple files is programming
paradigm dependent. We want to create consciousness of the importance of knowing
how to read large systems. The centrality metric taken from social network analysis
applied to the network of cognitive language elements that programmers built in order
to comprehend a system seems to reflect the paradigm thinking. Paying attention on
how programmers understand source code at system level will help to improve the
code itself and the system it describes.

The results we can get from our research might seem distant from the general ob-
jectives we have presented in this section, but we believe they are a small step that can
help in considering computation not something that happens only in the computer but
also in the mind of people and in the society that depends more and more on software.
We are discovering the needs to raise the level of communication of source code to
understand what we are doing when writing, reading, and comprehending a software
system.

2.3 Research questions

We want to investigate the effects of using DCI paradigm regarding the comprehension
of source code. We are looking for empirical evidence to support the claimed benefits
that comes when using DCI paradigm compared to using OO paradigm.

\[ \text{RQ1: Does DCI-trygve source code increase correctness of program comprehension}
\]
\[ \text{compared to OO-java source code?} \]

Our first research question involves correctness of comprehension, i.e. what de-
velopers understand when reading source code should match the intention of the de-
signers of the code that have develop the system. The score of correct answers were
used in (Corritore and Wiedenbeck, 2001), (Ramalingam and Wiedenbeck, 1997), and
(Salvaneschi et al., 2014) to name a few. Correctness have been used in the majority
of research works regarding program comprehension when comparing two or more
paradigm-languages as shown in Table 4.1. Program comprehension questions are the
easy way to get access to the results of the cognitive tasks programmers perform to
understand a system. For this purpose we count the correct answers that scores the level of comprehension in each subject. A correct answer means that the source code was understood and a the answer match our expectation of correctness.

**RQ2: Does the comprehension of system functionalities using DCI-trygve source code take less or more time than using OO-java source code?**

In our second research question we introduce the role of time consumed when understanding a source code. This is important because one of our objectives is to help in reducing the maintenance time. The measure of time spent on comprehension tasks has been used in most of the research works we have consulted when researchers were trying to compare the benefits of two or more different paradigms or languages. Specially close to our research are (LaToza et al., 2007) and (Salvanesehi et al., 2014). This research question is behind the idea to find a technique, paradigm, or way of thinking that helps to reduce the wasted time when comprehending a system through its source code.

**RQ3: Is programmer’s attention more focused using DCI approach than using OO approach during source code comprehension?**

Our third research question is related to the focus of attention when a programmer is reading source code to comprehend a system. This is about the importance and the permanence time spent in specific locations in the code. It is similar to the location of fixation usually studied in eye-tracking research works, but we call focus of attention to remark the cognitive aspects involved during program comprehension at system level. We want to know first if there is a noticeable central element in the systems in terms of network theory structural analysis, and second, if the centrality degree of languages elements are correlated to the time spent on reading. Our method to observe the properties of the paradigms involved in the experiment regards how subject approach the information contained in files. It differs from other researches that have looked how the information regarding a software system is read, as in (Corritore and Wiedenbeck, 2001) that observed the access to documentation and source code regarding the directions and breath of comprehension. In our work we observe the programmer’s attention on files that contain the source code that describe the system. This will shed light on understanding the cognitive role of context elements in trygve language in terms of centrality and reading time. We want to know if that element is actually functioning as central building block during the cognitive task of system comprehension. Conversely, we do not expect to find a most noticeable central element in the OO-java approach. As claims state that system functionality in OO are spread among classes, we presume that centrality is shared with more than one language element in the OO-java approach.
Chapter 3

Data, contexts, and interactions

3.1 DCI paradigm

DCI stands for Data, Context, and Interaction. Even though it is not a new idea, this paradigm is not well spread and it remains unknown within software engineers nowadays. We can find traces of DCI concepts in MVC (Model View Controller) pattern\(^1\) where “The top level goal was to support the user’s mental model of the relevant information space and to enable the user to inspect and edit this information” (Reenskaug, 2003). DCI concepts give a new meaning to OOP (Object Oriented Paradigm) or a new start from its roots (Reenskaug and Coplien, 2009b). The main ideas behind the DCI are design methodologies like Responsibility-Driven Approach (Wirfs-Brock and Wilkerson, 1989) and Object-Oriented Role Analysis and Modeling – OOram (Reenskaug, Wold, Lehne, et al., 1996). DCI was conceived by Trygve Reenskaugh and further developed jointly with Coplien et al. (Coplien and Bjørnvig, 2011).

With DCI paradigm we consider that the programmer’s mental model and the end user’s mental model should be aligned to reduce errors and surprises in running computer systems. That means we should strive to implement in code the end user’s concepts about the system. This is strong related with the direct manipulation metaphor in MVC pattern to give “the sense that end users are actually manipulating objects in memory that reflect the images in their head” (Reenskaug and Coplien, 2009b). The traditional work products in a software development process that actually capture the end user’s ideas of how a system works are use cases as described by Jacobson in (Jacobson, 1992) and (Jacobson, Spence, and Bittner, 2011). System behavior is specified in use cases and they determine the network of interacting objects that follows established set of steps toward achieving a goal for specific users or actors. From analyzing use cases we can extract the roles that objects play when interacting with other objects. These roles will define a system behavior within a use case or context independently from the type of objects that are capable of playing those roles. In DCI terminology we call the description inside contexts as the what-the-system-does part; whereas domain

---

\(^1\)MVC pattern was conceived in 1978 by Trygve Reenskaug when dedicated to Alan Kay’s vision of the Dynabook.
objects with the data they hold will represent the system state or \textit{what-the-system-is} part.

The algorithm that specifies a use case is implemented directly in the code within the body of a context and its roles as first-class language constructs. Contexts orchestrate system execution by specifying object interactions through roles assignments and by declaring methods to trigger use cases. All of this makes traceability easy between requirements and implementation, and it also makes the run-time behavior more visible in the code than traditional Object-Oriented approaches. With DCI we reach an strong separation of what changes at different rates, thus DCI thinking helps in addressing modifiability too. In DCI, only when executing a use case at run-time, i.e. a system functionality, we can have complete objects playing its assigned roles ready to interact with the right behavior injected at the right time toward achieving a specific goal. In that way, DCI allows developers to pay more attention to the architecture of the running system. The idea of an end user’s mental model written in the code makes DCI appropriate to address functional requirements. The end users’ point of view is about what the system can do for them. A DCI architecture allows programmers to focus in system behavior, not in smaller individual units of isolated behavior. DCI stands for Data, Context, and Interaction to denote three essential perspectives and to make an strong reference to the run-time of a system.

- **Data**: are uniquely identifiable objects within a single address space. Represent the data interface an all possible intrinsic behavior each object has. Data define the structure of the space in a running system.

- **Context**: is the responsible to initiate inter-object communication. A context provides the structure of the run-time, the algorithm that objects undergo to implement a use case or system functionality.

- **Interaction**: is the actual inter-object communication where the high-level sequencing of execution is governed by role methods that drives the communication between objects.

### 3.2 What the system is

The \textit{what-the-system-is} is the part with slow frequency of changes over time in the evolution of a software system. This part is the one that has a strong relation to the data that the system works with. This part encompasses all the elements in a software system that are directly responsible of the software state. This is the data that is usually represented in classes that mimic the structure of tables if using a relational database. It is also the behavior described in those classes that become the possible behavior of
Chapter 3. Data, contexts, and interactions

objects that has not side effect on other objects nor other data than in the data encapsulated in the object itself. We call this behavior described inside classes as the intrinsic operations within objects.

When we think about a system we start defining conceptually its parts in terms of the domain. We explain the system from the domain expert’s point of view. The concepts of classes of objects needed to build the system appear and we start using the domain vocabulary to begin naming these classes. Abstract classes factors out commonalities and guide the programming activities, they shape the form of the system regarding the data. In a DCI architecture, classes only define dumb objects, we want the classes being stable as long as possible. We don’t define the interaction between objects within class definitions as in traditional object-oriented programming. We want to separate what the system is from what the system is capable to do. Furthermore, why would we want to have an object with all possible behavior available when a system functionality may only need part of its behavior at specific time? An object should be only smart enough to provide the required functionality at the right time and in the right context.

We can see in Table 3.1 the main separation DCI paradigm proposes. This division is just a general reference to know what can be included in these partitions, this is not a blunt division or a strict rule to follow. This division is based on an strong idea about parts that change at different rates. It represents the separation in the highest level of abstraction in a DCI architecture.

<table>
<thead>
<tr>
<th>What-the-system-is</th>
<th>what-the-system-does</th>
</tr>
</thead>
<tbody>
<tr>
<td>System state</td>
<td>System behavior</td>
</tr>
<tr>
<td>Inherent, essential</td>
<td>Acquired, complex</td>
</tr>
<tr>
<td>Data objects</td>
<td>Roles and contexts</td>
</tr>
<tr>
<td>Data access</td>
<td>Objects interaction</td>
</tr>
<tr>
<td>Classes and interfaces</td>
<td>Contexts and roles</td>
</tr>
<tr>
<td>Intrinsic operations on objects</td>
<td>Extrinsic operations on objects</td>
</tr>
<tr>
<td>Class defining data structure</td>
<td>Algorithms defining run-time structure</td>
</tr>
<tr>
<td>Slow changing part</td>
<td>Fast changing part</td>
</tr>
<tr>
<td>UML static views</td>
<td>UML dynamic views</td>
</tr>
</tbody>
</table>

Table 3.1: DCI main separation.

### 3.3 What the system does

The services that the system provides to the end user and all the interactions that happen is what the system does in order to accomplish its purpose. We are not referencing here the intrinsic capabilities of data objects, with what-the-system-does we are focused

---

2Coplien usually references as dumb objects to objects that are only responsible for its own data to contrast the smart objects idea grown in traditional object-oriented programming.
Chapter 3. Data, contexts, and interactions

in the relation between objects and theirs interactions, this is the collaborating network of objects toward achieving a system goal. What the system does is described in the roles, this behavior does not belong to any object until the moment of executing a use case or context. The services the system provides are implemented in the roles envisioned by the end users to name the domain objects. Roles are used to give a name and meaning to domain objects when interacting with other objects. With roles the end user abstracts the unneeded behavior for specific services to emphasize only the important aspects for the function being performed. The end user names domain objects with the role they need to play in order to deliver a service. These system services change at human scale according to business needs. What the system does is related to “the tasks the system carries out for users and the way those tasks are structured” (Coplien and Bjørnvig, 2011, p. 32). The structure is declared or designed in the source code within the roles implementation, and these role are orchestrated in contexts to accomplish some system functionality. The end user’s view of services are key to design what the system does part of a system.

The main elements involved in what-the-system-does part are contexts and roles. To define roles we need the end user’s view of the services that the software is going to provide, these are the use cases in requirement analysis. A use case is a sequence of tasks toward a goal. Use cases define the scope of system functionalities and in a DCI system they are implemented as contexts. The real architecture of a system exists only in run-time and DCI dedicates specific parts of the source code to describe the execution of a system, therefore we are closer to model more accurately the architecture in the code.

3.4 The computational metaphor of theater

Objects are the actors of a series of acts and scenes of a play that represent the use cases and its deviations that finally shape what end users can do with a computer system. The programmer is like the stage manager of the play that writes the lines that cue actors or objects to carry out their parts in specific moments in order to interact with other actors or objects toward the completion of a scene or a system functionality within the entire system. The same play, this is the set of scenes or contexts, can be reinterpreted by different actors or objects over and over again.

There is a strong use of the theater metaphor in trygve language, we can deduce that from the vocabulary used in the DCI paradigm related works, specially in the trygve language manual (Coplien, 2016). In DCI thinking, methods are called scripts in a very theatrical sense, actors are the objects which play roles, and cues are the invocation of methods that allows a scene or use case to develop properly as stated in the scripts. The metaphor of theater in computer field has been around us since 1993,
specifically in Human-Computer Interaction field. DCI thinking shares the vision embodied in the search of a better metaphor for human-computer experiences. End users incrementally build a model of the system’s internal processes based on their own experimentation. This model brings end users expectations into line with the capabilities of the software system (Laurel, 1991, p. 30). Laurel explores in her book named *Computer as theater* a deeper vision of the meaning of an interface in general and in the general sense. DCI thinking follows this vision by incorporating the end user’s mental model in the code.

In the history of computer programming, programming languages have departed more and more from the language that machines understand. New languages usually incorporate not only high level constructs that compress lower level chunks of code related to the problem to be solved, but also adds mechanisms useful to communicate the intention of programmers to human beings. The computational metaphor of theater pursues, in part, this goal. For example, in a theatrical play cues are not part of the play, they are part of the script used to produce and reproduce the play multiple times. In a similar manner, we can think cues as not being part of a trygve language program for the computer, they are actually for humans. Humans need these cues to enhance program understanding by expressing better the intention of programmers.

### 3.5 End user mental model

During the development of a software system there are three generic entities that deserve a thorough observation to understand what the source code should include. These entities are humans, computers, and programs. They might seem too general entities, but they will help us to realize the reasons why the end user’s mental model, this is his or her vocabulary and thinking, should also be written in the source code text besides just the programmers mental model of the system.

Who deals with the program or source code of a system? Mostly programmers do, but also architects, and less frequently testers, and rarely business people. Who deals with the running system? We cannot deny that most frequently end users do, then testers and also programmers, and often business people too. Then, the question that raises is where the mental model of end users is in the system. We generally find end users mental model in the analysis documents, like in use cases or requirements. If the end user interact with the system most frequently than any other stakeholders, shouldn’t the end user mental model be expressed in the source code too?

The interaction between computers is not as complex as human interactions. We can think of multiple interacting computers as one big computer, but interactions among human beings are complex and subjected of interpretations and full of confounding factors. In Figure 3.1 this is explained by the looped arrow attached in the
human entity indicating interactions between humans that is missing in the computer entity figure.

From a general point of view, DCI is about aligning the end user’s mental model with the programmer’s mental model. “For a smooth interaction between man and machine, the computer’s ”mental” model (also the programmer’s mental model) and the end user’s mental model must align with each other in kind of mind-meld. In the end, any work that users do on their side of the interface manipulates the objects in the code. If the program provides accurate real-time feedback about how user manipulations affect program state, it reduces user errors and surprises” (Reenskaug and Coplien, 2009b). It means that, as end users deals, or will deal, with the actual system (i.e. they only interact with the system on execution or running system) the description of the run-time should be clearly expressed from their perspective to avoid surprises later. End users describe use cases with their own vocabulary where nouns are roles that give different meanings to domain objects in different contexts, therefore use cases should be written in the source code.

### 3.6 The trygve language

From a naive view of trygve we may consider this language a slightly variation of java language, but a deeper exploration of its features by programming and experiencing the language will reveal a paradigm thinking and a lot of frustration for the newcomers. This is a DCI-centric language that has reached research level to investigate and
examine the DCI paradigm and it is open-sourced in github\(^3\). “The trygve language is designed to reduce the transition from run-time to scripting and back again” (Coplien, 2016). We have the structure of the behavior or run-time written inside contexts in the roles that objects will play, and the structure of the state or data in classes and interfaces. Trygve language has the following general characteristics:

- Trygve is built upon java VM and shares most of the java syntax.
- No friend, static, super, protected keywords exists to avoid class-oriented programming.
- Only one value: null.
- Semicolons are optional;
- Method declarations can be const to create read-only methods.

Regarding the type checking system, the new language trygve is a strongly typed language. It is strongly type-checked at run-time, and roles types are duck-typed. In trygve there is no exception handling mechanism, no RTTI (Run-Time Type Information), and no concurrency. The trygve language strive for stateless computation within roles scripts and has a strong focus in avoiding class programming. In trygve we will find only declarations and expressions, the DCI community that is designing the language choose to leave no room for statements. An expression is evaluated as the value of the last expression executed. For example, an if-else is usually an statement in many languages, but as we can see in Listing 3.1\(^4\) we can return the last executed expression in an if-else structure, in that way we can get rid of intermediate identifiers necessary otherwise to hold those values. Classes and contexts are declarations that can contain other declarations as nested declarations, and method and roles are also declarations but they cannot contain inner declarations.

```java
int fact(int n) {
    return if (n <= 1) 1 else n * fact(n-1)
}
```

LISTING 3.1: If-else as an expression.

\(^3\)Rich information about trygve language can be found in https://github.com/jcoplien/trygve.

\(^4\)Example taken from ACCU 2016 conference from the presentation ‘Aglimpse of trygve: From class-oriented to real OO’ by J. Coplien. ACCU is an non-profit organization for anyone interested in developing and improving programming skills, originally named after Association of C and C++ Users.
3.6.1 Trygve program structure

In Listing 3.2 we show a general program structure that a trygve program usually follows. From line 1 to line 6 we have an interface and a class declarations. Interfaces and classes remain similar to java but with paradigmatic differences, for example, in classes will not appear complex code describing the interaction between objects.

```
1 interface Interface {
2     // interface definition
3 }
4 class Class {
5     // class definition
6 }
7 context Context {
8     role Role {
9         public void starts() {
10            // execution description
11         }
12         // more role methods
13     } requires {
14         // list of role-player object’s method signatures
15     }
16     public Context(Role o) {
17         Role = o
18     }
19     public void trigger() {
20         Role.starts
21     }
22 }
23 }
24 {
25     Class object = new Class()
26     new Context(object).trigger()
27 }
```

Listing 3.2: Trygve general program structure.

From line 7 to line 23 we have a context declaration. Contexts are new language elements that represent a use case in code. Inside a context can exist roles, from line 8 to line 15 there is a role named Role where we describe the execution for the objects which will play the role at run-time. A role in trygve comes with a requires clause, from line 13 to 15 is declared the requires clause which specifies the methods that an object requires to play a role. Any object that implements the method’s signatures in the requires clause can play the role (duck typing). These are the methods that will access the data the objects maintain, and these methods can be used in the role body. A context can have one or more constructors, from line 17 to 19 is declared the constructor
where dumb objects are assigned a role to become smart role-player objects. A context usually has one or more trigger methods to start the use case as shown in line 20 to 22.

A trygve program is not complete without an enactment block, from line 22 to 25 is declared the enactment bloc where objects are instantiated from classes, and context are created to be triggered. After the execution of a context, the objects that has played a role in it become to its original behavioral form as before being assigned a role.

### 3.6.2 Classes and interfaces

Today, most object systems express and organize concepts closer to the program data model than to its process or behavioral model (Coplien and Reenskaug, 2012). Trygve programmers have similar intentions in mind when using a class (i.e. to represent a domain concept) or an interface (i.e. to approach different domain objects) than in java, but, in trygve, classes won’t contain complex computation between objects, and interfaces will be used to organize the structure of aggregated objects in classes. Thus, classes describe simple or dumb objects, and classes and interfaces together define the form of the data.

### 3.6.3 Contexts and roles

A context is a use case in code, and roles are the behavior to be attached to objects during the execution of the use case. Contexts are responsible to assign or map roles to objects and to define the triggers that run a use case. Within roles is specified the network of collaborating object that pushes toward a goal or use case completion. In other words, in a context we are describing the structure of the run-time for the execution of a system functionality.

Passing different objects that are capable of playing the specified roles declared in a context allows programmers to maintain the structure of a system execution and change the way data is accessed. This is where polymorphism resides in DCI, where different types of objects can play a role within a context in different executions. This does not disrupt the main use case execution flow as dynamic polymorphism does in classical OO, the structure of the execution remains the same, it only changes how data is accessed. As long as objects that play roles can satisfy the requires clause, we are accessing different data with the same run-time structure defined in contexts.

Classes are part of the domain model, they represent the way the data is structured. Classes classify object by how they are built, whereas roles classify objects by how they act. These are the two tops for decomposing a system using DCI approach, these are the what-the-system-is and the what-the-system-does.
3.6.4 Code example explained

The following code example is presented from Listing 3.3 to Listing 3.8 and it is about a clock timer and two type of clocks (digital and analog) to show the hour based on the timer. The system functionality of moving time (context) creates a network of interacting objects to keep the timer moving and the clocks printing the hours. The digital and analog clock types are described by simple classes that implements the Clock interface. This interface helps to organize the types of clocks as clocks that their hour can be updated. The ClockTimer class declares a timer for objects that maintain their own states, these are the data for hours, minutes, and seconds. The only system functionality of this small example is declared in the context MoveTime that moves times and makes the clocks show the hours. The interaction between objects is described within roles where the timer interacts with the clocks, the clocks get updated, and the timer keeps moving. To start this system functionality we need to instantiate the context as a regular object and run the use case, this happens in the enactment block shown in Listing 3.8.

```
1 interface Clock {
2   public void update(Clocktimer timer) const;
3 }
```

Listing 3.3: Clock Interface.

The interface shown in Listing 3.3 is required to structure the data used within a generic list of different clocks as shown in line 5 in Listing 3.8.

```
1 class DigitalClock implements Clock {
2   public void update(Clocktimer timer) const {
3     System.out.println("[" + timer.getHour().toString() + "":" + timer.getMinute().toString() + 
4                     ";" + timer.getSecond().toString() + 
5                     ");
6   }
7 }
```

Listing 3.4: Digital clock class.

```
1 class AnalogClock implements Clock {
2   public void update(Clocktimer timer) const {
3     System.out.println("{" + timer.getHour().toString() + 
4                     "|" + timer.getMinute().toString() + 
5                     "|" + timer.getSecond().toString() + "}");
6   }
7 }
```

Listing 3.5: Analog clock class.

---

5This is an executable example that has be run using trygve version 2.28
Listings 3.4 and 3.5 are straightforward classes that define objects to represent the analog and digital clocks. It has only one method and it does not modifies the state of the instance. The method update is post-fixed by a const to prevent this method from changing the state of the object.

```cpp
1 class Clocktimer {
2   private int hour;
3   private int minute;
4   private int second;

5   public Clocktimer() {
6       hour = 0;
7       minute = 0;
8       second = 0;
9   }

10   public int getHour() const { return hour; }
11   public int getMinute() const { return minute; }
12   public int getSecond() const { return second; }
13   public Clocktimer getTimer() const { return this; }
14   public void tick() {
15       incrementSecond();
16   }

17   private void incrementSecond() {
18       second++;
19       if(second == 60) {
20           second = 0;
21           incrementMinute();
22       }
23   }
24
25   private void incrementMinute() {
26       minute++;
27       if(minute == 60) {
28           minute = 0;
29           incrementHour();
30       }
31   }
32
33   private void incrementHour() {
34       hour++;
35       if(hour == 24) {
36           hour = 0;
37       }
38   }
```

**LISTING 3.6: Clock timer class.**

Listing 3.6 declares a class that describes a clock timer domain object. Its data or state is defined from line 2 to line 4 as private data to provide an hour, minute, and
second. The state of a new brand object of type `ClockTimer` is set in the constructor defined in lines 6 to 10 that sets the hour, minute, and second to zero. Simple accessors are declared from lines 11 to 14, where the `const` keyword specifies them as read-only methods, meaning that when any of these method are invoked the object that receives the message won’t change its state. From line 15 to 37 are declared mutator methods, one public method that unchains the remaining private methods. All method in the `ClockTimer` class are instance methods, remember that there is no class method or static keyword in trygve. Furthermore, this methods are intended to only modifies the data on the same instance. There is no complex object interactions within a class declaration in a DCI system, class declares simple data objects with intrinsic behavior capable to access only to its own data.

```java
1  context MoveTime {
2       public MoveTime(Timer timer, List<Clock> clocks) {
3           Timer = timer;
4           Clocks = clocks;
5       }
6       public void run() {
7           Timer.moves_time();
8       }
9       role Timer {
10          public void moves_time() {
11              tick();
12              Clocks[0].updates_state();
13              Thread.sleep(1000);
14              Timer.moves_time;
15          }
16          public Clocktimer gets_timer() {
17              return getTimer();
18          }
19          } requires {
20              public Clocktimer getTimer() const;
21              public void tick();
22          }
23          stageprop [] Clocks {
24              public void updates_state() {
25                  Clocks[index].update(Timer.gets_timer());
26                  if (index < lastIndex) Clocks[index + 1].updates_state();
27              }
28          } requires {
29              public void update(Clocktimer theChangedSubject) const;
30          }
31    }
```

**Listing 3.7:** Move time context.
Listing 3.7 shows the unique context in this simple trygve example. In the constructor (lines 2 to 5) the objects passed as parameters get assigned a role, this is where objects become role-player. In this case a ClockTimer type object get assigned a Timer role and a List<Clock> type object get assigned the role vector Clocks. We can use a role type instead of a class type as a parameter in a method signature, as in line 2 with first argument. In this case we use the Timer role type instead of a class type, this is known as duck typing. It means that the object timer should meet the contract established in the requires clause of Timer role (lines 19 to 22). A duck type relies on what the object can do instead of what type the object should be. In our example, the object timer should implement getTimer() and tick() methods. If we see where the instantiation of the MoveTime context happened (Listing 3.8 line 7), we see that the object passed as first parameter is of type ClockTimer that implements both methods stated in the role’s requires clause. We can also see the assignment of the List<Clock> object to the role Clocks as a duck type but with the object argument depending on a class type.

There are two types of roles declared inside MoveTime context: Timer and Clocks. Roles cannot exist outside contexts, they only make sense inside contexts. First, you should note that a stageprop is a role that its requires clause only contains const methods, meaning that its role-player object won’t change state during the context execution. On the other hand, a role declared using the keyword role contains in its requires clause at least one method that mutates the state of the role-player object.

Another thing to note in Listing 3.7 regarding role declarations is that the stageprop role is a role vector. A role vector is used to specify the same role for a collection of objects, in our case the DigitalClock and AnalogClock objects. A role vector declares the index and lastIndex keywords to be used in its body as we use in lines 25 and 26.

Roles describe the interactions between objects, in our example we have a network of three objects interacting in the system functionality that moves time. The interaction between two role-player objects can be easily detected when we see a role identifiers inside the body of a role script. We have a Clocks role-player object that cues a Timer
role-player object to update their states (line 25). As this is a role vector we can deduce that each clock object in turn interacts with the Timer role-player object.

Finally, in Listing 3.8 is the enactment block for the system example where the context get instantiated in line 7 and then get triggered in 8. Enactment block are simple piece of code that deploy the complexity of the system.

It’s worth noting that DCI extracts the structure of the dynamics of run-time from classes and put it in contexts and roles. In OO systems functionalities are scattered across multiple classes, whereas in DCI the main functionalities of the system are encapsulated within the contexts declarations. This makes program comprehension easier as there is one location for each system operation to be found, the context. This also helps developers to separate the state and behavior of the system, the what-the-system-is and the what-the-system-does parts. These parts usually change at different rates during software development and keeping both separated facilitates the maintenance and the engineering of software systems.
Chapter 4

Related work

4.1 Program comprehension experimentation

Research works like (Sjøberg et al., 2005) and (Falcao et al., 2015) are useful to have an overview of the existent researches related to controlled experiments in software engineering in general and program comprehension in particular. Both research works implement a systematic literature review and report data of experimental research works done in the span of time that goes from 1993 to 2013. In (Sjøberg et al., 2005) authors selected 103 articles that report controlled experiments with human subjects from years 1993 to 2002, this represents 1.9% of all published articles in 12 leading journals authors looked at. They used IEEE keyword taxonomy and found that 2.9% of researches were related to experiments with programming paradigm and also 2.9% for software psychology\(^1\). Students were used as participant in 81% of the cases. Authors remark that using students as subjects may reduce experimental realism, but it may still be useful for testing novel approaches and initial hypotheses. In (Falcao et al., 2015), a short systematic review report that used data from (Borges et al., 2015), authors looked also at controlled experiment with human subjects and found 135 articles from years 2003 to 2013, that represents a 15% of total articles reviewed, but they don’t detail whether the experiment was related to program comprehension or not. Their results show that subjects involved in experiments were recruited mostly by convenience, 78.2% were students. Participation were mandatory in 20% of the experiments, 31% voluntary, 53% unclear stated. In our case participation is voluntary and we use a convenience sample of students from our Universities and professionals from former colleagues connections. Running controlled experiments using human subject in the field of program comprehension is not the most common type of experiment, but we realized that experimenting with human subject seems to be a reasonable approach given that Software Engineering encompasses a set of activities with a strong load on social aspects (Juristo and Moreno, 2013, p. 26).

\(^{1}\)Software psychology is related to the understanding of code regarding human factors.
With the arrival and the stay of object-oriented paradigm and the appearance of object-oriented languages, new experiments emerged to measure their benefits or to be compared with existent procedural or functional languages. This is the case of (Lee and Pennington, 1994) where authors investigate how procedural and object-oriented paradigms differed in practice. Their research is motivated by the inconvenience found when switching from procedural to object-oriented design, as well as the difficulties to learn the OO paradigm for expert procedural designers. Through an experiment involving a design exercise with 10 designers (3 procedural experts, 4 OO experts, and 3 OO novices that were procedural experts), authors analyze the differences through nine categories of design activities. They used *think out loud* verbal protocol with a dual purpose, to track the progression of design activities by annotating each action the designers took, and to find design activity categories. Authors have measured the amount of time devoted to each category to conclude about which paradigm may favor which category. They have also reviewed the resulting design solutions, and they have run three trials to measure the effect of learning on subjects. Even though they have a low statistical power because of the small amount of subjects involved in the experiment (3 procedural experts, 4 OO experts, and 3 OO novices), their research resulted interesting for us regarding the theories or assumptions they were testing. Considering expertise, they have found that procedural subjects spent more time analyzing the problem than solving it, OO subject spent more time describing data abstractions (classes-objects) and also evaluating their design. Regarding design results, procedural designers got more variation on their design solutions, whereas OO design solutions matched across designers in terms of class definition and its relationships. In our experiment, subjects are not asked to produce a design solutions or modify code, we are interested in measure the comprehension of source code only by reading. When producing code by writing or modifying it, design skills are involved, and develop these skills for a novel paradigm, like DCI, would take more time than just learn to read and comprehend code.

At the time that OO was becoming more popular, the need to find a mental model representation for OO was growing and research works like (Burkhardt, Détienne, and Wiedenbeck, 1997) were carried out to know about program understanding in this new paradigm. There was already existent program comprehension models, like Pennington’s model based on van Dijk and Kintsch’s model of text understanding that distinguishes the *program model* and the *domain model*. Authors found some limitations on Pennington’s model on program comprehension regarding its application in OO and in large sized programs. Specially interesting to us is that authors remark that previous models didn’t account for “representation of delocalized plans and the representation of text macrostructure”. Authors define a plan as the main goals of the problem that “correspond to functions of the program viewed at a high level of granularity. They do
not correspond to single program units. The complex plan which realizes one goal is usually a delocalized plan in an OO program” (Burkhardt, Détienne, and Wiedenbeck, 1997). This definition of delocalized plan is what we know as a system functionality, that DCI proponents claims to represent a problem in OO systems regarding program comprehension. In DCI, a system function directly corresponds to a specific location in the source code where the context is defined. Burkhardt et al. evaluated the validity of the cognitive distinction of program model and situation model finding that this distinction is still valid in OO programmers. They found that the “situation model (or domain model) is more fully developed using the OO paradigm even on early phase of comprehension”. This contrasts Pennington’s findings in procedural paradigm for smaller programs where she stated that first it was develop a program model and just after that the domain model. Our research is not as ambitious as theirs, we have a smaller scope, we are not creating a general comprehension model neither evaluating one for DCI. We are evaluating the paradigm-language factor being OO or DCI and how it affects comprehension in terms of time, correctness, and focus of attention. Even though, the literature reviewed on this topic is useful for our research work in terms of what was understood by object orientation and how the experiments approached this and other paradigms.

An empirical study about source code comprehension done by (Salvaneschi et al., 2014) to compare Reactive Programming and Object-Oriented Programming is closer to our purposes. We are going to do a similar work but focused in system functionalities instead of the reactive part of applications. The research approach used by the authors is useful regarding the experiment design and methodology. They recruited 38 subjects and made 2 groups, the OO group with 20 programmers and the RP with 18 programmers. The tasks proposed for their experiment only involve code inspection, they did not included code-writing tasks because they consider a different factor to be measured. A first approach for our experiment would have included tasks where subject should modify and change the source code, but we were not able to give an appropriate training to reach the necessary level of paradigm adoption. Besides, we agree with Salvaneschi et al. in considering that code intervention requires different skills than code comprehension.

From the related literature we have reviewed, controlled experiments with human subjects regarding program comprehension pursue different objectives and test different types of hypotheses. Some of them are postulated to investigate or develop a particular mental model for a specific paradigms (Burkhardt, Détienne, and Wiedenbeck, 1997), or to observe the effect of specific tasks (Burkhardt, Détienne, and Wiedenbeck, 2002) or strategy (Karahasanović, Levine, and Thomas, 2007). It is common to find related works that measure the effect of expertise in subjects and the effect of learning in different phases of the experiment (Wiedenbeck et al., 1999). Other experiments are run
to compare paradigms or languages exclusively, and others to test specific techniques. Our research is comparative in nature and simple in terms of the experimental design, but also most effective in terms of the resulted data. The complexity of these kind of empirical researches resides in the interpretation of the results and in the management of all possible threats to validity. In Table 4.1 we show a summary of all experiments with human subjects we have reviewed. The objective column explains which goals were primarily addressed. With mental model we refer those experiments in which the objectives were to evaluate or develop a mental model of programmers using different kind of paradigms or languages, we have classified as expertise those works were researchers compare how previous knowledge affect comprehension, we use the word phase to indicate experiments that are longitudinal and involve more that one trial to measure the learning effect over time, with task we refer to studies that investigate the effects of different tasks in the comprehension process, and we use paradigm-lang to identify research works that tried to find a better suited paradigm or language for program comprehension in general or for specific factors. Ours is on the last category and is similar to (Salvaneschi et al., 2014), (Walker, Baniassad, and Murphy, 1999), and (Wiedenbeck et al., 1999). Other researches are still handy regarding the experience of carrying out an experiment. What differentiates our work is that we are also measuring the focus of attention regarding the source code, the type of task, and the paradigm, i.e. a result of the programmer’s behavior during the process of program comprehension.

A common measure in program comprehension in all research works is time, even thought it is used in different ways, e.g. with upper limit or unlimited time. It seems an implicit agreement and common sense that less time to comprehend means easy understandability. But, of course time is used along together some measure of correctness. We use time and correctness to express comprehensibility of source code and we also measure the dwelling time in different parts of the system under experimentation to find the focus of attention. In a research work to study the most convenient layout and color of UML classes, authors define fixation as the stabilization of eyes on an object of interest for a period of time, where processing of visual information occurs (Yusuf, Kagdi, and Maletic, 2007). Eye-tracking research over source code is not as common as it is in graphic user interfaces, but this does not mean we cannot use some of their concept in our research. Roman and Marku study the program comprehension strategies, using an eye-tracker researchers observed the behavioral aspects of programming, specifically the location of fixation, fixation duration, and attention switching between areas of interest (Bednarik and Tukiainen, 2006). We are looking for focus of attention in the source code at a higher level of abstraction than programs of 15 to 30 SLOC as done in Roman and Marcu study. We observe directly the main language elements that designers use to write code as interfaces and class, in java, and interface, class, context and enactment block in tryge language.
Chapter 4. Related work

<table>
<thead>
<tr>
<th>Title</th>
<th>Author/s</th>
<th>objective</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus structures and mental representations in expert</td>
<td>(Pennington, 1987b)</td>
<td>mental model, phase, expertise</td>
<td>COBOL, Fortran</td>
</tr>
<tr>
<td>The effects of paradigm on cognitive activities in design</td>
<td>(Lee and Pennington, 1994)</td>
<td>phase, expertise, paradigm-lang</td>
<td>Design</td>
</tr>
<tr>
<td>Mental representations constructed by experts and novices in object-oriented program comprehension</td>
<td>(Burkhardt, Détienne, and Wiedenbeck, 1997)</td>
<td>mental model, phase, expertise</td>
<td>C++</td>
</tr>
<tr>
<td>An empirical study of novice program comprehension in the imperative and object-oriented styles</td>
<td>(Ramalingam and Wiedenbeck, 1997)</td>
<td>paradigm-lang, expertise</td>
<td>C++</td>
</tr>
<tr>
<td>A comparison of the comprehension of object-oriented and procedural programs by novice programmers</td>
<td>(Wiedenbeck et al., 1999)</td>
<td>paradigm-lang, expertise</td>
<td>Pascal C++</td>
</tr>
<tr>
<td>An initial assessment of AOP</td>
<td>(Walker, Baniassad, and Murphy, 1999)</td>
<td>paradigm-lang</td>
<td>AspectJ, Java</td>
</tr>
<tr>
<td>Mental representations of expert procedural and object-oriented programmers in a software maintenance task</td>
<td>(Corritore and Wiedenbeck, 1999)</td>
<td>paradigm-lang</td>
<td>C / C++</td>
</tr>
<tr>
<td>An exploratory study of program comprehension strategies of procedural and object-oriented programmers</td>
<td>(Corritore and Wiedenbeck, 2001)</td>
<td>paradigm-lang</td>
<td>C OO C++</td>
</tr>
<tr>
<td>Object-oriented program comprehension: Effect of expertise, task and phase</td>
<td>(Burkhardt, Détienne, and Wiedenbeck, 2002)</td>
<td>mental model, expertise, task(plan) phase</td>
<td>C++</td>
</tr>
<tr>
<td>Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study</td>
<td>(Karahananović, Levine, and Thomas, 2007)</td>
<td>mental model strategy</td>
<td>Java, UML</td>
</tr>
<tr>
<td>An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension</td>
<td>(Abbes et al., 2011)</td>
<td>code-comparison</td>
<td>Java</td>
</tr>
<tr>
<td>An Empirical Study on Program Comprehension with Reactive Programming</td>
<td>(Salvaneschi et al., 2014)</td>
<td>paradigm-lang</td>
<td>Scala, RP, Scala OO</td>
</tr>
</tbody>
</table>

Table 4.1: Consulted controlled experiments with human subjects on program comprehension

4.2 A mental model for program comprehension

Text comprehension or discourse comprehension of natural language started its development around 1970 and met psychology in 1980 with Teun A. van Dijk, Walter Kintsch, David Rumelhart and others. Before that, related studies focus at the sentence boundary, henceforth discourse comprehension moved from the structure of isolated context of independent sentences to the analysis of semantics of entire discourses with social context. The study of discourse emerged as an independent and interdisciplinary field were linguistics, grammatical methods, sociology, and psychology fields play an important role. In “Strategies of discourse comprehension” (Van Dijk and Kintsch, 1983), authors presented what was going to be the foundational model for some of the first works related to program comprehension in software engineering field. Then, the advent of artificial intelligence and the development of automatic processing of texts accelerated the search of a comprehension model to fit greater scopes
than isolated sentences in texts under study. Early models were categorized as structural, whereas subsequent proposed models were more dynamic and authors called strategical models.

Through an enumeration of basic assumptions grouped in cognitive and contextual assumptions, van Dijk and Kintsch present in their work an idealistic model by eliminating some underlying complexity of discourse comprehension. They also show the limitation of their model regarding different aspects related to linguistic parsing, knowledge representation and use, and contextual information. These limitations were purposely introduced to convey a practical approach. Van Dijk and Kintsch’s model pictures discourse processing at word level to construct overall theme or macrostructures, and macrostructures are needed to give meaning to words. It is a complexity oriented model where continual feedback between less complex and more complex units operate within a strategy. This is not an algorithm model where a generative grammar with syntactic parsing rules formulate a structural description. The strategies that the reader applies are “like effective working hypotheses about the correct structure and meaning of text fragment, and these may be disconfirmed by further processing” (Van Dijk and Kintsch, 1983, p. 11). It is strategic because readers use their knowledge in a strategical way to understand the text regarding what their interest is. Their model is broadly represented by the textbase and the situation model. The general strategy of text comprehension is the construction of the textbase information, i.e. the representation of the input discourse in the episodic memory\(^2\). The textbase is build from the propositions and relations among propositions that give meaning to text in all its detail. The situation model is the construction of what the text is about that incorporates previous experiences and previous textbases of similar knowledge content. “Understanding is restricted to an evaluation of the textbase not only with respect to local and global coherence, but also with respect to its corresponding situation model” (Van Dijk and Kintsch, 1983, p. 12).

Nancy Pennington in “Stimulus structures and mental representations in expert comprehension of computer programs” (Pennington, 1987b) took the fundamental concepts of text comprehension from van Dijk and Kintsch to carry out an empirical study on program comprehension. She measured mental representation in expert programmers using procedural paradigm with source code written in Cobol, Fortran, and Assembler. Pennington’s model is based on van Dijk and Kintsch’s model of text understanding. She distinguishes between two kind of representation the reader makes while understanding a text:

---

\(^2\)In psychology, episodic memory is related to events that get stored as memories and together with the semantic memory conform a declarative memory. The semantic memory is more factual and is related to something that is known and can be inferred.
• Text structure knowledge: is based on structured programming about the recognition of fundamental control flow structures like sequence, iteration, and conditional. These structures play a role in comprehension of procedural programming in organizing the memory representation of the overall text or macrostructure. Structured programming proponents claimed that a strict structured program was easy to comprehend because it would correspond programmer’s mental organization.

• Plan knowledge: is the understanding of multiple program instructions that executed together accomplish a certain system functions. “A plan is a structure with roles for data objects, operations, tests, or other plans, and with constraints on what can fill the roles in a given instantiation as well as specifications as to data flow and control flow connecting segments within plans” (Pennington, 1987b).

Pennington’s view of computer programs slightly includes the idea that a program is also written for other programmers. What she clearly states is that programmers should be skilled in comprehending written programs, but it is not explicit the idea of writing programs intended for human comprehension instead of computer execution. “Because programs are instructions to a computer, the closest analogs among natural language texts are instructions about how to perform a particular task, often referred to as procedural instructions” (Pennington, 1987b). This might suggest that programs are also written for programmers, but what she does is actually show a structural comparison of different type of languages and a resemblance between a person and a computer regarding their ability to follow written instructions. The parts of a program intended for humans are usually declarative, we can find this in a program where natural language is used to name identifiers that cues programmers to understand the system, i.e. the ‘what’ that declares the intention of a program construct instead of the ‘how’ that is found in the narratives of source code in its procedural parts.

In her following work, Nancy Pennington adapted van Dijk and Kintsch’s model making a translation of their model of text comprehension to program comprehension (Pennington, 1987a). She took the textbase that is the verbatim representation of the surface of a text in the memory, this is the first representation built when comprehending text, and she named program model to expresses the what and how it is said in a program text. It is isomorphic with the text structure and reflects what is contained in the text at a propositional level, micro and macro structure, and it is built by mean of automatic processes from the verbatim representation. The situation model, the situation which is referred by the text that is isomorphic with the situation described in the text, it is built by inferences using the domain knowledge or other situation models already processed. She corresponds the situational model to a domain model in program comprehension.
Much of the subsequent works on program comprehension take into account Pennington’s model, either for investigating comprehension models for other programming languages and paradigms like in (Von Mayrhauser and Vans, 1995). Even though we are not going to test a mental model of program comprehension, research works regarding mental models resulted interested to us because we can understand how a paradigm thinking helps to model the meaning of computer system in the minds of programmers. There is a close relationship on how programmers approach the source code with the language-paradigm they need to understand.
Chapter 5

Research approach

5.1 Scope

We analyze general characteristics of DCI and OO paradigms to evaluate their impact on the correctness, timing and focus of attention of the underlying cognitive process of program comprehension. In the context of students or professionals reading source code we will interpret the results in order to state which paradigm helps to produce more comprehensible source code. Although comprehension happens in programmers’ mind, we don’t have to think that we are testing or grading people, we are comparing a pair of paradigm-language combinations to test our hypotheses.

Paradigms are like school of thinking that imposes a way of reasoning and expressing a computational solution where adopters perceive it as a convenient way to proceed. The design of a software system is also made through a comprehension process shaped by a paradigm thinking to produce mental models of a situation or problem that is then iteratively written in a file as a computer script or program. Therefore, the code takes a form and tells how it is going to be read in order for readers to catch the intent of the writer. In our work we are investigating only the reading part which requires different skills from writing as noted by (Salvaneschi et al., 2014). Through comprehension questions we measure the results of understanding source code by counting correct answers and by measuring the time to answer these questions. We also track how programmers navigate the code through main language elements to capture the focus of attention of the process performed to comprehend a program given a particular question, need or incentive to read.

Here we define the scope in terms of the objective and goal of the experiment. We follow a Goal Question Metric (GQM) suggested in (Wohlin et al., 2012, p. 85). This is a high level overview of the experiment:

- **Object of study**: The main objects of study are DCI and OO paradigms. A paradigm is possible only within human beings, it represents a way of thinking, interpreting, or modeling the world. Because of that, the source code reflects the
paradigm through how its code is arranged. For DCI source code we use trygve, a novel DCI-centric language, and we select java for OO source code.

- **Purpose:** The intention of the experiment is to evaluate the comprehension and focus of attention of source code in programmers when reading DCI or OO code. We want to know how much these paradigms influence the comprehension of source code, specifically, we want to quantify programmers comprehension of DCI-trygve source code compared to OO-java source code.

- **Quality focus:** The primary effects under study are the correctness and timing of source code comprehension. Therefore, the quality focus is the effectiveness and efficiency of source code comprehension when reading code using DCI-trygve or OO-java approaches, that ultimately is perceived as a maintainable code.

- **Perspective:** We, researchers, carried out this experiment, hence the perspective is from researcher’s point of view. Although, the selected quality focus concerns software managers too.

- **Context:** The experiment was run using master students and professional programmers with one or more years of experience in java OO programming. We presented five small-medium representative systems examples in two versions, DCI and OO for source code comprehension.

## 5.2 Experimental design

This is a one factor with two treatment controlled experiment to compare DCI-trygve with OO-java regarding program comprehension. We use randomization in all steps and a combination of balancing and blocking techniques. We use balancing to simplifies and strengthens the statistical analysis and we use blocking to eliminate the undesirable effects of confounding variables that can distort the observation of the desired effects as explained in (Wohlin et al., 2012). We balanced both experimental groups with an equal number of subjects and we blocked all variables as possible, e.g. the experience of subjects and the equivalence of the experimental units.

Depending on the characteristics of the comparison, this kind of experiments can be conducted with a crossover design, where each subject receives the two treatments, or with a parallel design where each subject receives only one treatment. A crossover design is also called within-subject design or repeated measurements design, and the benefits of applying this design are primarily that confounding variables effects, like experience, can be eliminated because each subject serves as his/her own matched control. Other benefits is that with less subjects we can reach a higher statistical power. One of the drawbacks presented in this kind of design is the wash-over period. On
the other hand, a parallel design, also known as between-group design, uses separate samples of subjects to avoid learning effects. Some experiments cannot be designed with a crossover design because of the incompatibility of the treatments under study. In our case, we considered that shifting paradigms would cause an undesirable irreversible effect in the comprehension of programs. We know that changing paradigms is not easy and sometime takes years, and we do not want those effects in the results. We can find the same decision in (Wiedenbeck et al., 1999) (Corritore and Wiedenbeck, 1999) (Corritore and Wiedenbeck, 2001) (Abbes et al., 2011) (Salvaneschi et al., 2014) regarding the paradigm shifting factor. As we still have to deal with the learning effect because subjects will gain knowledge in the first system presented, we introduce systems and tasks randomly during the experiment. The downside with a parallel design is that we have less data points per subjects.

As our experiment requires the participation of people, we have gone through the process established by the Institutional Review Board (IRB) at our University to protect human subjects in order to get the approval and run the experiment. We, as researchers, have completed the Human Subjects Research (HSR) course from the CITI Program\(^1\). Our experiment was categorized as Human Subjects Research with No Greater than Minimal Risk.

### 5.3 Experiment time-line design

We present the time line of the experiment in a format taken from (Leedy and Ormrod, 2010) to show each step, from the registration of subjects to the final observations made. In Figure 5.1 we show in green those steps where subjects have an active participation.

![Figure 5.1: Experiment time-line design.](image)

- **Subject registration & experience survey**: this is the first step done by the people who want to become participants. At the time of registration, prospective subjects complete a small programming experience survey to help us draw the sample of a population.

---

\(^1\)This is the Collaborative Institutional Training Initiative, it is a leading provider of research education content that certifies researchers involved in human subject related researches.
Chapter 5. Research approach

- **Matched pairs**: based on the programming experience survey, we can form pairs in terms of the experience reported. This will help us to have two groups with a balanced experience level. To avoid confusion, our study is not a pair comparison design\(^2\), we are just using the technique of pairing to randomly select paired subjects to different groups and maintain the experience factor balanced.

- **Random selection within pairs**: from each pair of subjects we randomly select to which group each subject goes.

- **DCI-trygve & OO-java**: these are the two groups formed from the previous step, they are balanced in size and in the experience level reported.

- **DCI-trygve training**: one group should receive training in DCI-trygve approach to compare the correctness and timing of source code comprehension against the OO-java group. The group that receive the training can be viewed as the group that receive a treatment, and the other group as the control group.

- **Training tasks**: before running the experiment, subjects are given two system examples for training purposes to get familiar with the web interface and the mechanism to run the experiment.

- **Skill assessment**: this is a pre-experiment test that measures the skill level of subjects regarding the paradigm and language of the group they belong.

- **Tasks\(^*\)**: these are the experimental tests or trials applied to the source code where subjects have to answer comprehension questions by reading and navigating the system in order to measure the correctness, timing, and focus of attention.

- **Obs**: these are the observations made, the data gathered required to test our research hypotheses.

The step where tasks\(^*\) are given to subjects represents the actual experiment run, all former steps are preparation steps that help us to detect and know the population and blocking confounding factors.

### 5.4 Experimental parameters and variables

These are variables and parameters that exist in each unitary experimental run performed by subjects. We describe them in their most simple form through the values they can hold and the possible interpretation of their values. We have three dependent variables, one for the correctness level, other for the time taken to perform the

\(^2\)In a pair comparison design experiment the mean of differences between paired subjects are computed to test the null hypothesis, for instance using a pared t-test. A matched-pair design or pair comparison design experiment should rely on objective measurements to reliable match a pair, otherwise pairs might be biased by different confounding factors.
tasks, and the last to measure the centrality degree of language elements. We compute the centrality by tracking the behavior of subjects during the cognitive process of source code comprehension. Timing and correctness are the most common observable variables in program comprehension experiments. The behavioral aspects of a comprehension process is usually used to detect patterns or strategies made by programmers. In our case, as we are interested in the paradigm and language used to write the source code, we use it to determine which part of the program was the focus of attention regarding the comprehension of source code. To detect the focus of attention, we have split the source code of the systems in its main language constructs, these are classes and interfaces for java, and classes, interfaces, contexts, and enactment blocks for trygve. This separation is usually made in java by distributing the code for a system in different files. We use the same metaphor of files during the experiment, then we can track how subjects switch among files and the permanence time in each file. In the end, we have an adjacency matrix from which we can determine the central language element using centrality metric from graph theory.

Table 5.1: Experimental variables description.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm-language</td>
<td>Qualitative</td>
<td>Two possible alternatives: OO-java, DCI-trygve.</td>
<td>This is the treatment or factor under study.</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>Parameter</td>
<td>Fixed, we use the same instrumentation in both groups when applying both treatments.</td>
<td>This is the web site that we specially prepared to run the experiment. All subjects have the same interface and the same source code visualization.</td>
</tr>
<tr>
<td>Programming experience</td>
<td>Blocking variable.</td>
<td>Experience measured in months and qualification (student/professional/other). We recruited programmers with more than 1 year of experience.</td>
<td>This variable includes the experience in different languages, it is used for balancing skills levels between groups and block the effects of experience by let it constant.</td>
</tr>
<tr>
<td>Reading time</td>
<td>Quantitative</td>
<td>Measured in deciseconds.</td>
<td>Time taken to perform a comprehension task. A task contains multiple questions. We measure permanence time in files.</td>
</tr>
<tr>
<td>Correctness score</td>
<td>Quantitative</td>
<td>Integer: from 0 to 93.</td>
<td>There are true-false questions and sorting questions with a don’t know option to answer a comprehension questions. We accumulate the correct answers submitted as the score obtained.</td>
</tr>
<tr>
<td>Language element centrality</td>
<td>Quantitative</td>
<td>A value between 0 to 1, where 1 is the higher degree for centrality.</td>
<td>This metric is not directly taken, but computed from the reading behavior, i.e. the switching between files.</td>
</tr>
</tbody>
</table>

The dependent variables are the observable effects or response variables. We also have parameters, that are characteristics of the experiment that are maintained constant during each unitary run, as it is the instrumentation used in our experiment (see
Section 5.6). The blocking variables are those that usually have an effect on the independent variables but we don’t want those effects in the measurements.

5.5 Participants

Recruitment is always a topic of concern for any human subject related experiment and is also the most frequently criticized matter regarding the external validity of the results. Hence, the recruitment and the external validity have to be considered together toward a balanced resolution. We start with the subject characterization, this is the desirable characteristics of subjects regarding the generalization of the results and the feasibility of recruiting. One thing about characterizing subjects is that when we want to be more specific and precise in the characterization before recruitment, then the recruitment may become more problematic. We looked for programmers, specifically, java programmer with one or more years of experience programming in OO java. This was the required characterization because participants read programs written in java language, and truyge language shares some of the java syntax. It was convenient because it typifies the majority of people who write code in the world today. “Unlike other disciplines, the experimental subject has a very important effect on the results of the experiment in Software Engineering and, therefore, this variable has to be carefully considered during experiment design” (Juristo and Moreno, 2013, p. 58). We took care during the recruitment phase to be able to characterized the selected subjects to assess the external validity of the experiment.

The population sample we use was not a fully randomized sample, as we are doing an exploratory research to compare a novel paradigm-language approach we use a convenience sample taken from students at our University and professionals from close former colleagues connections. As stated in (Sjøberg et al., 2005), convenience samples are still appropriate for the exploratory, the illustrative, and the clinical situations because there is not yet a relevant population using this brand new approach nowadays.

At the time of registration participants filled a small form with information about their programming experience, that allowed us to draw the population in detail. The form was composed by questions regarding one primary and two secondaries programming languages together with the total amount of month’s experience for each one. We have also asked about the qualification of the experience given, as professional, student, or other, and a self assessment using a Likert scale about the proficiency regarding Object-Oriented programming knowledge.

Although we have not use a random sample, we did use randomization between matching pairs regarding experience level reported. We wanted to make two balanced
groups in size and programming experience. For that reason we elaborated an heuristic based on the data gathered from the registration form to build an index that helped us to match subjects pairs. Subjects with the most similar index where separate and then randomly taken to one of both experimental groups.

\[
experience\_index = \frac{\sum_{i=1}^{3} sim_i \cdot pri_i \cdot exp_i \cdot qua_i}{5/oo}
\] (5.1)

For each subject we generated an experience index as shown in equation 5.1. We used the data from the registration form for the three programming languages reported by participants. The \( sim \) variable is the java similarity, given a language it takes different values, 2 if the language is java, 1.5 if it is similar to java, and 1 if it is not similar to java. The \( pri \) variable is the language primariness factor, it weights 2 for the primary language reported and 1.5 or 1 otherwise, if there is no input given this value is 0. The \( exp \) variable is the months' experience number. The \( qua \) variable is the qualification, it is valued with 2 for professionals and 1.5 for students. Finally, \( oo \) variable corresponds to the self assessment knowledge of Object-oriented that subjects report using the five levels Likert scale.

We refused to gather auxiliary data as any demographic data such as population, race, income, and education, as well as personal data as age and sex. We wanted to draw our results exclusively from a software engineering point of view.

## 5.6 Instrumentation

The instrumentation for the experiment is centralized in a web application we built from scratch to run the experiment.\(^3\) We have all the source code (experimental units) for the system examples managed through the our web application. We use the web application to present the code, to show the guide lines for participants, and to gather the data. All these three instruments: the objects, the guidelines, and the measurement instruments, are presented in by Wohlin et al. in (Wohlin et al., 2012) as the instrumentation.

The objects, i.e. the source code, is displayed using a javascript library called \textit{prism}\(^4\) that allows us to present the source code as usually shown in professional websites for programmers. In Figure 5.2 we show a screen-shot of the GUI for subjects when running the experiment. In this picture we show the class Account for the trygve version of the Bank system example. At the left part of the screen there is a menu

\(^3\)The web application can be found in github and used to run similar software comprehension experiments [https://github.com/hvaldecantos/pctdatacollector](https://github.com/hvaldecantos/pctdatacollector)

\(^4\)A syntax highlighter and a visualization theme. prism [http://prismjs.com/](http://prismjs.com/)
with a list of files for the whole system example under experimentation. We shuffle the
list of files for each system presented to lower possible confounding effects of subject’s
permanence times in files due to the same order of the files between experimental runs.

![Figure 5.2: Web application display code.](image)

We tried to keep a very simple GUI design to avoid confounding effects of the
usage of the web application. To avoid learning effects regarding the usage of the
system, subjects start in a training mode where they can run a training experiment
with two system for training purposes. After one run in training mode, subjects have
the choice to start the real experiment. In Figure 5.3 we can see an experimental task
question, in this case the task is check implemented features. It is easy and intuitive for
subjects to switch from the task to code by clicking on the name of the file and in the
question answer tab. We only track time when the source code is displayed, thus we
consider only the reading time.

We wanted to create similar conditions as any professional setting, for example,
participants were not limited in the way they complete the experiment nor in the time
they needed to comprehend the system. They can complete the experiment in one
session or in multiple sessions. Having a web application to run the experiment helped
in some way to avoid the Hawthorne effect, subjects get tracked in their switching and
permanence among files unobtrusively.

### 5.7 Experimental units

When defining our experimental units we have maintained the focus on the engineering
part of software development. It may seem counter intuitive to think that the objects
in which the experiment is run is the source code instead of the programmers, but
if we test programmers our experiment would have become more similar to those in
psychology. To avoid falling in a psychological experiment, we assume programmers follow the rules that dictates a programming paradigm. “Depending on the goal of the experiment, the experimental unit in a SE experiment can then be the software project as a whole or any of the intermediate products output during this process” (Juristo and Moreno, 2013, Section 4.2). Any engineering relies on science to define methods, techniques, and procedures to achieve similar results when engineering a product or service. A computer programming paradigm is based on a set of rules, patterns, and principles that defines a style or way of programming and thinking that ultimately draws the form or arrangement of the text that represents the source code. At the same time we are far from discarding the social and psychological content involved in using a language to elaborate a computational solution within a software development team. But, for our experiment we take a software engineer position to evaluate program comprehension.

We have created five systems examples for the experiment, and two more for training purposes. Each system is written in two versions, one in OO-java and the other in DCI-trygve. As we were in touch with the community that is developing the trygve language in [https://groups.google.com/forum/#!forum/object-composition](https://groups.google.com/forum/#!forum/object-composition)
language, we get some help to understand the paradigm in order to create the experimental units. We also take some examples made within the community\textsuperscript{6} and introduced small changes to adapt to our purposes. We have written all OO-java versions. Both versions are aimed to be similar, see Section 5.7.2 about system equivalence.

5.7.1 An experimental unit centric model perspective

The experimental unit of an experiment is the object in which the experiment is run. In this Section, we take an experimental unit centric model approach to show the relationship between the treatments, experimental units, and results for the reader to understand about how dynamic and symbiotic this relationship can be in our experiment.

![Figure 5.4: Generic unit centric model perspective.](image)

The generic form of this model is shown in Figure 5.4 where we can see a treatment applied to an experimental unit and the observable results where each $ij$ application of the treatment is an unitary experiment, thus the observable $results_{ij}$. In an example from Juristo and Moreno’s book, researchers wanted to determine the size of the code when implementing an algorithm using two different programming languages. To give the example more external validity, let’s say we have 5 different algorithms, then we will have 10 observations ($i = 1, 2; j = 1, ..., 5$) or unitary experiments\textsuperscript{7} in one run, as shown in Figure 5.5. In our experimental unit centric model we don’t show experimental subjects. In this example, subjects are the people who write the algorithm (apply the treatment) in one or two languages, depending on the experimental design used. Subjects may have some effects on the results but researchers can block these effects by considering subjects with similar experience in the language and in translating algorithms into programs, researchers can also grow the number of subjects in the sample of the population used to get more realistic results.

Our experiment has some similarities with the above mentioned example, but we want to show some differences between the experimental units in the example and in our experiment. For our experiment we have five different computer systems, each written in two different paradigms, OO and DCI, using two different languages, java

\textsuperscript{6}The Library system example was originally written by Andreas Söderlund, the Bank system appears first in “Lean Architecture” book and it was intervened by Matt Brown, the Spell checker system example is currently part of the trygve manual that Coplien is writing, and the Tower of Hanoi system was originally created within Object-Composition group.

\textsuperscript{7}An unitary experiment is an application of the treatment on an experimental unit.
and trygve. We can think of a system as one experimental unit with two possibilities, being written in trygve or java, as shown in Figure 5.6. Therefore, the source code files with the code for the system are our experimental units, the trygve version is only seen by subjects from the DCI-trygve group and the java version by the OO-java group.

As we said, the experimental subject is who apply the method or treatment to the experimental units. Subjects comprehend code using a paradigm-language and we, researchers, measure the level of correctness and timing of this cognitive process. In our experiment the treatment and the experimental units are affected by the same paradigm for each group, both of them are subject of human psychological effects because the source code is written by a human and the treatment is apply by a human.

### 5.7.2 Code equivalence

Another point that we have paid careful attention is the equivalence of source code in describing the same system in different paradigm-languages. The running system from an end user’s point of view resulted from executing the source code in different paradigms and languages should be identical. For the blueprint of the systems we defined some points we followed to preserve equivalence of source code:

- **Code readability**: the code was normalized, for example, we keep a line space between classes in java, and between classes, contexts, and roles in trygve; we use lines of 80 characters long and same naming conventions when possible.

- **Code comprehensibility**: in the code we use the same or similar name for identifiers. The code reflects almost the same or similar domain decomposition in terms of classes or interfaces but following the paradigm approach in turn.
• **Run-time**: we have the same program behavior, not only the system manage the same information and the output are the same, but we maintain small algorithms almost invariable in both systems. The end user’s view of different versions of a system are exactly the same keeping the same I/O metaphor. This was possible because our system example were small.

• **Paradigm**: each system should be written in a representative way for the paradigm it represents in the experiment. This is one of the main threat to construct validity explain further in Section 7.1.3.

This represents an objective way to declare equivalency between both program versions and maintain the differences between paradigms. From what we experienced by programming the 5 system examples in both versions is that in general a java implementation will have more number of classes and interfaces than a trygve version, this is because in trygve we have the role language element that exists inside a context and describes what a class or subclass may be intended for in a java program. Furthermore, a context in tryve has small similarities to what in java sometime appears as a class that unchains the execution to accomplish a use case followed by jumps among method invocations that live in different classes in the code. In OO design this might be accomplish through a facade or maybe a mediator design pattern, but sometimes ends like a god class or doer class with all the drawbacks about coupling of different system responsibilities.

Another difference between both versions is related to the java interfaces and what is called a role contract in trygve. A role contract allows programmers to build a duck type that can avoid the usage of interfaces when interfaces are not used to organize the domain objects but to arrange the the behavior of a system operation. As we can expect, there are irreconcilable differences between both paradigm-languages expressed in a system implementation. With our experiment we measure how this differences affect program comprehension in programmers.

### 5.7.3 Descriptive metrics of system examples

We have reviewed and study our ten programs from an objective point of view by counting and comparing their language elements and physical line of code. This is a way to observe the differences of the shape of the code that may help in interpreting the results. In Table 5.2 and 5.3 we show the basic descriptive statistic measures regarding line of code for our five system in both versions.

<table>
<thead>
<tr>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>72.0</td>
<td>202.0</td>
<td>225.0</td>
<td>193.6</td>
<td>232.0</td>
<td>237.0</td>
</tr>
</tbody>
</table>

**Table 5.2**: Java code
The means of SLOC for each system example presented to subjects in our experiment were 193 SLOC for java (see Table 5.2) and 202 SLOC for trygve (see Table 5.3). Similar experiment have been performed with smaller programs. A research work with similar research questions than ours (Salvaneschi et al., 2014) used programs from 9 to 82 SLOC with a mean of 30 and 44 SLOC for the different versions. Different researches related to program comprehension used smaller programs, like (Wiedenbeck et al., 1999) that used program from 15 to 25 SLOC, and (Pennington, 1987b) that ran two experiments, one with short programs used programs with 15 SLOC and another with 200 SLOC. We consider that programs with more than 200 SLOC are useful to observe differences at system level regarding the paradigm under study.

5.8 Experimental tasks

For each of the five systems we formulate four main questions of tasks related to program comprehension. On average there are 18 sub-questions per system (see Table A.2), with a minimum of 15 on the smaller system and a maximum of 20 sub-questions on larger systems. The whole experiment includes 93 sub-questions for the four categories of tasks.

The experimental tasks consists in approximately 18 independent questions with three option responses, true, false, and don’t know (dk) questions for each system. As said before, we are not looking to grade subjects, we are interested in measuring how amenable results the combination of a paradigm-language for a programmer to comprehend the source code. When grading people, examinees will try to find guessing strategies to favor them in order to obtain better grades when they lack of the required knowledge to answer correctly. It is known that guessing introduces noise and produces unreliable results when looking for correctly ranked examinees, as noted in (Burton and Miller, 1999), (Mameren, Vleuten, et al., 1999), and (Burton, 2001). Based on these researches we have opted to adopt an intermediate approach related on how to measure subjects’ answers in our experiment. This is, we give only one point to correct answers given, and zero otherwise (incorrect or don’t know answers). This differs from (Mameren, Vleuten, et al., 1999) in which they subtract one point on incorrect answers and used correct minus incorrect answers formula scoring. The goal of Mameren et al. was to grade students more accurately, ours is to measure the comprehensibility of source code. Henceforth, our main objective is to raise the reliability of our tests by
avoiding subjects answering a question when they could not find the answer, that is why our tests include the don’t know (dk) option.

We first looked at the experimental measurements objectives regarding our research questions. We wanted to know accurately as possible what to measure. Here we define the experimental tasks in order to get and idea about what objective we are addressing in each one:

- **Check implemented features**: this task is about delimiting a system functionality from the source code. Does the code express the boundaries of a system functionality or use case? A specific system functionality does not encompass the entire system. The limits of a system functionality is something that might be blurred in OO. By measuring this question we try to answer if the code reveals the intention at the system level. It is related to the traceability of use case in code. This task requests subjects to select the features or operations implemented in the system related only to the execution of a system functionality.

- **Look for changed and unchanged objects**: after a system functionality has been executed we should be able to know by reading the code and given the input which objects have changed state and which ones have not. It is about the results of an execution, what has happened after triggering the system functionality is reflected in the states of the objects involved. We request subjects to select true when an object of a given type changes state during the execution of a system functionality, and false otherwise.

- **Sort execution flow**: this is about the algorithm of a system functionality that is somewhat hidden behind class interfaces in OO. It is about describing what the system does in terms of small steps toward a goal to discover the use case steps in the source code. We request subjects to sort a given list of steps to match the execution flow related to a system functionality considering a given input.

- **Describe object interactions**: this is about the network of collaborating objects. How objects collaborates between each others and in which direction, this is the real architecture of a (running) system. In terms of objects collaboration, this task is about how the system does what it does. We request subjects to select true when a claim like “An object A requests object B to perform function C.” states something that actually happens in the execution of a system functionality considering all possible execution paths.
Chapter 6

Analysis and results

6.1 Overview

In this chapter we analyze the data to give an answer to our research questions presented in Section 2.3. For all tasks in all systems examples we observe the answers submitted by the experimental subjects, the reading time spent, and the reading behavior regarding how subjects switched among files to read the source code. There are 5 system examples in both versions, DCI-trygve and OO-java, each one has 4 tasks and each task has several questions, from 2 to 6. In total, a unitary experiment run comprises 93 questions (see Table A.2). We collected the data from 12 participants that have completed the experiment (6 subjects from the DCI group and 6 subjects from the OO group), in average it took 49 minutes of reading time to finish all tasks.

6.2 Correctness analysis

We are testing the claims which assert that DCI improves source code comprehension. We showed in Section 2 and Section 4 that there are some evidence in the literature that supports those claims. Even though, we can not assume the given claims true upfront. We start with the null hypothesis stating that the DCI-trygve approach produces the same effects regarding correctness of source code than the already known OO-java approach:

\[ H_0 : \mu_{dc} = \mu_{oo} \]
\[ H_a : \mu_{dc} <> \mu_{oo} \]

Even though we are interested in knowing if DCI-trygve approach increases comprehension, we use a two-tailed test because, otherwise, the test would gain greater power in the direction of the alternative hypothesis. Furthermore, we want to be able to detect if the DCI-trygve approach is statistically significant worse in some cases. We care both sides, hence the chosen null and alternative hypotheses\(^1\).

Correctness analysis is about counting correct answers, each count means the source code was understood and a correct answers was submitted. The indicator of correctness is the cumulative score of correct answers submitted for all tasks and for all system

\(^1\)We use the same form of the null hypothesis \( H_0 \) and alternative hypothesis \( H_a \) for the remaining analysis regarding correctness and time consumption.
examples. This score can go from 0, meaning that no correct answer was submitted, to the max score of 93 where all submitted answers were correct. We use the percentage of the max score for the correctness analysis, because the amount of questions differs in some cases between systems and between tasks (see Appendix A.2). We show an overview of the results using descriptive statistics in Figure 6.1 where we can compare visually both distributions of general scores and the means that are drawn as red plots.

The mean for DCI-trygve group is $\mu_{dci} = 74.5$, whereas the mean for OO-java group is $\mu_{oo} = 67.17$. We can state, by making a single value comparison of means and by visually inspecting the distribution of the scores, that in our sample the DCI approach produced better results regarding correctness of code comprehension. To estimate the probability that this difference of means for the two samples would remain true for both populations we apply an appropriate statistical test in regard to the type of data collected to verify if the difference between means is statistically significant. Because our data are ordinal in nature, i.e. a rank that scores correct answers obtained by each experimental subject, and we cannot assume any underlying distribution of our samples, we use the Mann-Whitney U-test. We stick to non-parametric methods because there is no reliable method to verify the normal distribution assumption in our small samples. For all tests we define $\alpha = 0.05$ as our tolerance for making a Type I error.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Rank avg</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCI-trygve</td>
<td>6</td>
<td>8.67</td>
<td>52</td>
<td>0.037</td>
</tr>
<tr>
<td>OO-java</td>
<td>6</td>
<td>4.33</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Mann-Whitney U test for scores.

From Table 6.1 with the results of the statistical test we can reject the null hypothesis with an 95% level of significance. With the rank sum we indicate which approach is
dominant. This result provides an answer to RQ1 (Section 2.3) allowing us to conclude that DCI-trygve approach produces a more comprehensible source code.

In the correctness analysis we also include particular results for each system and for each type of comprehension question. We follow the same approach as in the general analysis already made. We continue with the analysis of the correctness, first particularized by system examples and then by task types.

In Figure 6.2 we show the descriptive statistics for the different systems presented to subjects. We can note that the DCI-trygve approach produces better results, except for the Bank system example. In Table 6.2 we show the results of the statistical test for each system example. We find that $H_0$ can be rejected only for the Menu system with high significance ($p < 0.05$). In four cases presented we can observe that the DCI-trygve approach rank sums are greater than OO-java approach, except in Bank system example.

<table>
<thead>
<tr>
<th>System example</th>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>Rank avg.</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>DCI-trygve</td>
<td>6</td>
<td>86.67</td>
<td>6.00</td>
<td>36.0</td>
<td>0.621</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>92.22</td>
<td>7.00</td>
<td>42.0</td>
<td></td>
</tr>
<tr>
<td>Library</td>
<td>DCI-trygve</td>
<td>6</td>
<td>77.50</td>
<td>7.75</td>
<td>46.5</td>
<td>0.226</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>63.33</td>
<td>5.25</td>
<td>31.5</td>
<td></td>
</tr>
<tr>
<td>Menu</td>
<td>DCI-trygve</td>
<td>6</td>
<td>66.67</td>
<td>8.83</td>
<td>53.0</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>52.63</td>
<td>4.17</td>
<td>25.0</td>
<td></td>
</tr>
<tr>
<td>Spell checker</td>
<td>DCI-trygve</td>
<td>6</td>
<td>59.17</td>
<td>8.08</td>
<td>48.5</td>
<td>0.121</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>52.50</td>
<td>4.92</td>
<td>29.5</td>
<td></td>
</tr>
<tr>
<td>Tower of H.</td>
<td>DCI-trygve</td>
<td>6</td>
<td>85.96</td>
<td>7.08</td>
<td>42.5</td>
<td>0.560</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>80.70</td>
<td>5.92</td>
<td>35.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Mann-Whitney U test for scores per system.
Finally, we continue with the correctness analysis particularized for different type of comprehension task. Remember that the experiment comprises four type of tasks and each task includes multiple comprehension questions as explained in Section 5.8. In Figure 6.3 we show the descriptive statistics for the different tasks. We can estimate visually that the DCI-trygve approach produces better results for all type of tasks, but the Mann-Whitney U test results show that we can reject the null hypothesis with high confidence (\( p < 0.05 \)) only for “Look for changed-unchanged objects” tasks. What we can observe is that for all tasks the DCI-trygve approach rank average and rank sums are greater than in the OO-java approach without any exception.

![Box plot of correctness ranks per task type.](image)

**Figure 6.3:** Box plot of correctness ranks per task type.

<table>
<thead>
<tr>
<th>Task type</th>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>Rank avg.</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check implemented features</td>
<td>DCI-trygve</td>
<td>6</td>
<td>80.00</td>
<td>7.25</td>
<td>43.5</td>
<td>0.468</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>76.11</td>
<td>5.75</td>
<td>34.5</td>
<td></td>
</tr>
<tr>
<td>Describe object interactions</td>
<td>DCI-trygve</td>
<td>6</td>
<td>87.68</td>
<td>7.25</td>
<td>43.5</td>
<td>0.462</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>81.16</td>
<td>5.75</td>
<td>34.5</td>
<td></td>
</tr>
<tr>
<td>Look for changed-unchanged objects</td>
<td>DCI-trygve</td>
<td>6</td>
<td>80.95</td>
<td>8.50</td>
<td>51.0</td>
<td>0.048</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>66.67</td>
<td>4.50</td>
<td>27.0</td>
<td></td>
</tr>
<tr>
<td>Sort execution flow</td>
<td>DCI-trygve</td>
<td>6</td>
<td>53.21</td>
<td>8.00</td>
<td>48.0</td>
<td>0.145</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>44.23</td>
<td>5.00</td>
<td>30.0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.3:** Mann-Whitney U test for scores per task type.

In summary, we have shown that with high statistical significance DCI-trygve approach performs better in general. We found that there is no difference between approaches when particularized by system examples, and DCI-trygve approach improved comprehend on subjects when asked to identify changed and unchanged objects after an execution of a system functionality.
6.3 Timing analysis

This Section is dedicated to investigate the results concerning our RQ2 (Section 2.3) related to time consumption, this is if reading and comprehending source code written in DCI-trygve requires less or more time than using code written in OO-java. Timing analysis is about the time required to understand the system examples. The time is measured when the source code is shown to subjects, thus, for this analysis we only consider the source code reading time and we exclude the time spent in reading and interpreting the tasks and questions. We follow a similar approach than in previous section, we start analyzing the time in general, for all tasks in all system, then we analyze the time particularized for each system example, and finally for each type of task.

We start by presenting the descriptive statistics for the general case in Figure 6.4. We can observe that both distributions are mostly overlapped and the means (red plots, $\bar{x}_{\text{dci}} = 39.66$ and $\bar{x}_{\text{oo}} = 56.27$) difference favors DCI-trygve approach for our specific sample. By testing the $H_0$ for time consumption we fail to reject it with a level of significance $\alpha = 0.05$ as shown in Table 6.4. On the other hand, the rank average and the rank sum shows that the dominant approach with lower time consumption is DCI-trygve.

![Figure 6.4: Box plot of timing scores.](image)

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Rank avg.</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCI-trygve</td>
<td>6</td>
<td>5.50</td>
<td>33.0</td>
<td>0.379</td>
</tr>
<tr>
<td>OO-java</td>
<td>6</td>
<td>7.50</td>
<td>45.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Mann-Whitney U test for time consumption.
We continue with the analysis particularizing the reading time for each system example. In Figure 6.5 we show the descriptive statistics for the different systems examples. We can note that the DCI-trygve approach requires spend less time for all systems, except for the Bank system example. In Table 6.5 we show the results of the statistical test for each one. Again, we fail to reject the null hypothesis for all cases using a high significance level ($p < 0.05$).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{boxplot_time_consumption.png}
\caption{Box plot of time consumption per system example.}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|l|l|l|}
\hline
System example & Group & N & Mean & Rank avg. & Rank sum & p-value \\
\hline
Bank & DCI-trygve & 6 & 5.05 & 6.67 & 40.0 & 0.936 \\
& OO-java & 6 & 4.31 & 6.33 & 38.0 & \\
Library & DCI-trygve & 6 & 7.94 & 5.17 & 31.0 & 0.230 \\
& OO-java & 6 & 12.23 & 7.83 & 47.0 & \\
Menu & DCI-trygve & 6 & 7.58 & 5.67 & 34.0 & 0.471 \\
& OO-java & 6 & 14.55 & 7.33 & 44.0 & \\
Spell checker & DCI-trygve & 6 & 9.13 & 5.33 & 32.0 & 0.298 \\
& OO-java & 6 & 13.56 & 7.67 & 46.0 & \\
Tower of H. & DCI-trygve & 6 & 9.96 & 5.67 & 34.0 & 0.471 \\
& OO-java & 6 & 11.63 & 7.33 & 44.0 & \\
\hline
\end{tabular}
\caption{Mann-Whitney U test for time consumption per system.}
\end{table}

Finally we show the results particularized for task type. In Figure 6.6 we present the descriptive statistics for the different tasks. We can estimate visually that the DCI-trygve approach produces better results for all type of tasks, but the Mann-Whitney U test statistic results show that we cannot reject any null hypothesis with high confidence. What we can observe is that in all tasks the DCI-trygve approach rank sums
are lower than OO-java approach without exception denoting that the DCI-trygve approach requires less time for code comprehension for all type of tasks.

![Box plot of time consumption per task type.](image)

**Figure 6.6:** Box plot of time consumption per task type.

<table>
<thead>
<tr>
<th>Task type</th>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>Rank avg</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check implemented features</td>
<td>DCI-trygve</td>
<td>6</td>
<td>13.04</td>
<td>5.67</td>
<td>34.0</td>
<td>0.471</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>16.19</td>
<td>7.33</td>
<td>44.0</td>
<td></td>
</tr>
<tr>
<td>Describe object interactions</td>
<td>DCI-trygve</td>
<td>6</td>
<td>9.30</td>
<td>5.17</td>
<td>31.0</td>
<td>0.230</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>15.32</td>
<td>7.83</td>
<td>47.0</td>
<td></td>
</tr>
<tr>
<td>Look for changed-unchanged objects</td>
<td>DCI-trygve</td>
<td>6</td>
<td>5.01</td>
<td>5.00</td>
<td>30.0</td>
<td>0.173</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>11.00</td>
<td>8.00</td>
<td>48.0</td>
<td></td>
</tr>
<tr>
<td>Sort execution flow</td>
<td>DCI-trygve</td>
<td>6</td>
<td>12.31</td>
<td>6.33</td>
<td>38.0</td>
<td>0.873</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>6</td>
<td>13.77</td>
<td>6.67</td>
<td>40.0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.6:** Mann-Whitney U test for scores per task type.

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 is no differences regarding reading time consumption between DCI-trygve approach and OO-java approach when comprehending source code. We have also restricted the timing analysis to only consider times when the correct level for each task was greater than 75% and we have not observed any statistical significant neither. Therefore timing remains an open issue that might find an answer if using greater samples.
6.4 Centrality analysis

In this Section we investigate the results concerning our RQ3 (Section 2.3) regarding which approach has a most noticeable central language element, this is related to the focus of attention when a programmer is reading source code to comprehend a system. With the centrality analysis we want to know, first, if there is a noticeable central element in the systems in term of network structural analysis, and second, if the time spent on reading is related to the the centrality metric computed for each node.

For each subject we create five graphs that represent the reading behavior while understanding each system example. As we capture how subjects switch among files, we are able to recreate this behavior in a directed multigraph\(^2\) to illustrate how the source code was read. We put this information in adjacency matrices from which we can build the graphs and compute the centrality metrics for each node. We call this graph the Cognitive Network of Language Elements (CNLE), this is the network built by each programmer during the cognitive task of comprehending a system from reading its source code. It is a common practice to store in separated files different types of language elements with high level of abstraction. We use the file metaphor in the same way as it is usually used to avoid unfamiliarity to subjects and to be able to capture the switching between files and permanence time in each file. Therefore, we can say that this network is formed by nodes that represents language elements with high level of abstraction. In our case, a language element can be a class or interface for java language; and a class, interface, contexts, or enactment block for trygve language.

\[ G = (V, E) \] (6.1)

In graph theory, a graph is represented as a pair of sets, as shown in Equation 6.1, where \( V \) are the nodes, that in our case are the language elements (or files), and \( E \) are the links between nodes that subjects creates when switching between language elements (or files) when comprehending a system. In Figure 6.7 we show two remarkable cases for both approaches, these are the CNLE of two individual subjects for the Menu system example. The size of the nodes are scaled to their importance according to the centrality metric measured from the structure of the CNLE built by each subject. In the DCI-trygve case we can observe a noticeable central element that happens to be the context language element in trygve language, whereas in the OO-java case high levels of importance is shared among more than one node.

We are going to use eigenvector centrality metric which is used in social network analysis to find the importance degree of a node in a graph. 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

\(^2\) A directed multigraph is a graph without restrictions on the number of links from one node to another node.
Figure 6.7: CNLE for Menu system example - DCI-trygve approach (left) and OO-java approach (right).

a node. “Eigenvector centrality tries to generalize degree centrality by incorporating the importance of the neighbors (or incoming neighbors in directed graphs). It is defined for both directed and undirected graphs” (Zafarani, Abbasi, and Liu, 2014).

Part of source code comprehension relies on how programmers approach the information contained in language element declarations inside files (nodes). For example, the importance of a file increases if it is consulted repeatedly, meaning that the file contains important information to be understood. In addition, if a file gets reached after being reading an important file, the reached files increases more its importance than getting reached from a non-important file. This is analogous as indicating which parts of a system require more attention in order to be understood. Therefore we define importance as the information contained in files that is central and essential to understand a system. It must be noted that not only the main reasons why a programmer is reading a system will constitute the forces that structure the CNLE but also the paradigm thinking that arranges the source code and guide the reading. We compute the eigenvector centrality metric for each system example separately 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 centrality index for entire graphs cannot reveal information beyond the whole structure of each CNLE, it cannot tell us more than which graph has a more centralized structure. This is known as centralization, a global metric that refers to the overall cohesion or integration of a graph, it tells little about local properties, it does not refers to the relative importance of local points in the graph. The overall centrality metric for the entire graph can be misleading if used to detect the existence of a node with higher centrality degree than the rest of the nodes within a network. We left the analysis of the centralization degree for future works. Next, we follow an intuitive analysis based on the comparison of individual node centrality degrees.

We compute the centrality for each file to correlate file degree centrality values with
subject’s reading times spent on files. The centrality degree is scaled from 0 to 1 by dividing each degree by the max degree obtained in a node within the network. This normalization is important because the number of files in a system can vary between approaches. We use the percentage of time reading spent on files considering the overall time spent on the system. We show the analysis for one of the systems that reflects better the characteristics of both paradigm-language approaches, and then, for space reasons, we only present the results for the rest of the systems.

In Figure 6.8 we show all the measures obtained for the Library system for both groups. In red (filled triangle) we draw the means of centrality and in light-red (triangle) all centrality metrics regarding each subject’s reading behavior, whereas in blue (filled circle) we draw the means of time consumption and in light blue (circle) each subject’s time consumption on file. Note that all measures are obtained for each file in the system and each node or file is sorted in the figure by its mean degree of centrality, i.e. from less important to more important files.

![Figure 6.8: Centrality and Time - Library system.](image-url)
We want to show what values of centrality for each file are more probable for the population in both groups to be obtained. We use the one-sample Wilcoxon signed rank test with a level of significance $\alpha = 0.05$ to compare the population mean to a hypothesized mean value to find a segment of centrality measures for each file, i.e. the possible values of centrality with a 95% of confidence. To get the lower end point of the segment we state that the null hypothesis is $H_0 : \mu \leq \bar{x}$ and the alternative hypothesis $H_a : \mu > \bar{x}$. Then, we hypothesize a mean value starting from $x = 1.00$ and decreasing its value iteratively by 0.01 until we set the lower end point of the segment when we are able to reject $H_0$, otherwise we set it to 0. To get the upper end point of the segment we proceed inversely. If we fail to reject the null hypothesis when computing the upper bound of the centrality segment, it means that the upper bound is greater than 1. As the centrality metric is scaled from 0 to 1, we take 1 as the upper bound. In this case the p-value if grater than 0.05. We write NA, not applicable, when all values of the sample has the same value, this is generally the case in DCI-trygve approach for the file with the declaration of the context, when its centrality value obtained is 1 for all subjects in the group. We draw this segment as a vertical black dashed line in Figure 6.8. Table A.5 and A.6 shows the numerical values for these segments with its respective p-value for DCI-trygve and OO-java respectively. The rest of the data is in the Appendix A.3.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>book.k</td>
<td>0.054</td>
<td>0.08</td>
<td>0.022</td>
</tr>
<tr>
<td>screen_printer.k</td>
<td>0.055</td>
<td>0.14</td>
<td>0.037</td>
</tr>
<tr>
<td>item_record.k</td>
<td>0.087</td>
<td>0.18</td>
<td>0.037</td>
</tr>
<tr>
<td>main.k</td>
<td>0.109</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>mock_card_reader.k</td>
<td>0.121</td>
<td>0.24</td>
<td>0.023</td>
</tr>
<tr>
<td>paper_printer.k</td>
<td>0.156</td>
<td>0.37</td>
<td>0.037</td>
</tr>
<tr>
<td>mock_book_scanner.k</td>
<td>0.227</td>
<td>0.52</td>
<td>0.023</td>
</tr>
<tr>
<td>borrow_library_items.k</td>
<td>1.000</td>
<td>0.99</td>
<td>0.079</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypad.java</td>
<td>0.073</td>
<td>0.15</td>
<td>0.037</td>
</tr>
<tr>
<td>ScreenPrinter.java</td>
<td>0.105</td>
<td>0.21</td>
<td>0.037</td>
</tr>
<tr>
<td>Book.java</td>
<td>0.129</td>
<td>0.20</td>
<td>0.023</td>
</tr>
<tr>
<td>ItemRecord java</td>
<td>0.167</td>
<td>0.32</td>
<td>0.037</td>
</tr>
<tr>
<td>PaperPrinter.java</td>
<td>0.280</td>
<td>0.36</td>
<td>0.023</td>
</tr>
<tr>
<td>MockCardReader.java</td>
<td>0.427</td>
<td>0.59</td>
<td>0.037</td>
</tr>
<tr>
<td>Screen.java</td>
<td>0.572</td>
<td>0.91</td>
<td>0.037</td>
</tr>
<tr>
<td>MockBookScanner.java</td>
<td>0.668</td>
<td>1.0</td>
<td>0.022</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.949</td>
<td>1.0</td>
<td>0.159</td>
</tr>
</tbody>
</table>

Table 6.7: Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve).

Table 6.8: Wilcoxon rank signed test for centrality segments - Library system (OO-java).

We want to show the uniqueness of the most central element in each CNLE, for that reason we consider class intervals of 0.2 units of centrality to draw a histogram for the mean value of centrality. In that way will put in evidence if a node of the higher valued class for centrality is unique and if it is accompanied by a gap of occurrences in subsequent lower classes or not. In Figure 6.9 we can observe the most central element in the DCI-trygve approach is unique and it is followed by a gap of two class intervals ((0.4, 0.6] and (0.6, 0.8]), whereas in the OO-java approach we observe no gap and a
unique node occurrence in the higher class. These histograms are representative for both paradigm according to the data we collected. See Appendix A.3 for the other systems.

![Figure 6.9: Histogram of centrality classes - Library system.](image)

Regarding time consumption, we proceed to correlate the means of centrality values obtained in each file to the means of reading time dedicated to each file. Table 6.9 shows the results obtained. We can see that reading time spent on files are strongly correlated to the centrality degree of files. As the time is measured after the programmer’s decision to read a file, we can say that, in general, when more central a node is, more reading time is spent on it. This is applicable for both paradigms, and it seems to reflect a mindful way of comprehending a system, but we left more analysis on this topic for future works.

<table>
<thead>
<tr>
<th>System example</th>
<th>Approach</th>
<th>Pearson ($r^2$)</th>
<th>Spearman ($\rho$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>DCI-trygve</td>
<td>0.9867</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>0.8470</td>
<td>0.7</td>
</tr>
<tr>
<td>Library</td>
<td>DCI-trygve</td>
<td>0.9926</td>
<td>0.8333</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>0.9699</td>
<td>0.8833</td>
</tr>
<tr>
<td>Menu</td>
<td>DCI-trygve</td>
<td>0.9908</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>0.9284</td>
<td>0.9</td>
</tr>
<tr>
<td>Spell checker</td>
<td>DCI-trygve</td>
<td>0.9954</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>0.9474</td>
<td>0.8182</td>
</tr>
<tr>
<td>Tower of H.</td>
<td>DCI-trygve</td>
<td>0.9937</td>
<td>0.8571</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>0.9524</td>
<td>0.9030</td>
</tr>
</tbody>
</table>

**Table 6.9: Centrality-Time correlation.**

Finally, to answer RQ3, we proceed to count the number of occurrences in the centrality class regarding the centrality segment obtained with 95% of confidence. As systems has different amount of files, we consider an equal number of the most central ones. We are interested in knowing if higher centrality degree is shared. Table 6.10 shows the count of centrality segments mapping the classes of centrality degree from 0 to 1 with a class interval of 0.2 of centrality unit. We can observe that in the highest centrality class, i.e. $(0.8, 1.0]$, DCI-trygve approach has a unique central file, and that 3 out of 5 times is 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 2 up to 4 files and it is not followed by a gap in the subsequent lower centrality class.

<table>
<thead>
<tr>
<th>System example</th>
<th>Approach</th>
<th>Files considered</th>
<th>(0.0, 0.2)</th>
<th>(0.2, 0.4]</th>
<th>(0.4, 0.6]</th>
<th>(0.6, 0.8]</th>
<th>(0.8, 1.0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>DCI-trygve</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Library</td>
<td>DCI-trygve</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>8 most central</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Menu</td>
<td>DCI-trygve</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Spell checker</td>
<td>DCI-trygve</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>5 most central</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Tower of H.</td>
<td>DCI-trygve</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OO-java</td>
<td>7 most central</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

TABLE 6.10: Centrality class count for most central files.

A better visualization of this mapping is shown in the raster plot in Figure 6.10. Each cell has the number of occurrences of files with the centrality degree class demarcated in the x axis regarding its centrality segment. Each row indicates to which system example the count of files belong. A white color indicates absence of files within a centrality degree class, and contiguous colors of red saturation show higher files occurrences in a centrality class degree for a specific system example.

Figure 6.10: Raster plot - File count over systems and centrality degree.

Regarding the identification of the most central element, we obtained that in DCI-trygve approach 93.3% of the time (28 out of 30) programmers select the same file as the most central one and this file is the context. In OO-java, programmers choose the same file 76.7% of the time (23 out of 30) as the most central one. Finally, we can conclude that DCI-trygve has a most identifiable central language element in the CNLE compared to OO-java. As in both approaches the reading time is correlated with the degree of centrality of files we can state that the programmer’s attention in the DCI-trygve approach is focused mostly in only one element, the context, whereas in OO-java is spread in more than one file.
Chapter 7

Discussion and Conclusion

We complete our work by exposing some discussions regarding the threats to validity and the conclusions. We list major threats to validity, how we mitigate them, and which ones are considered in the closing section. Finally, we include some discussion together with the conclusions derived from the results.

7.1 Threats to validity

A threat can be addressed in the experiment design or can be accepted and included in the interpretation of the results. Even though a threat can be overlooked, to make a valid experiment a suspected threat cannot be disregarded. We use Cook and Campbell extended list of threats to validity in the following subsections.

7.1.1 Conclusion validity

This kind of threat is also referred as statistical conclusion validity. Conclusion validity issues “affect the ability to draw the correct conclusion about the relations between the treatment and the outcome of an experiment” (Wohlin et al., 2012, p. 103)

- Regarding the reliability of measurements, we have measures related to the classification of the answers given, i.e. if an answer is correct or not, measures related to time, and measures related to pattern of reading or how the subject switched among files to read a system. Classifying the correctness of an answer is at some level based on human judgment, we reduced this threat by reviewing the source code, questionnaires, and correct answers with professionals and faculties at the University. The other two measurements are more related to the mechanism designed to gather the data, this is the web application used to run the experiment. We tested the application repeatedly and ran two small pilot experiments that were useful to correct the system and improve the understanding of tasks.

- As we run the experiment with a small sample we get low statistical power in the results. On the other side, we have a large number of questions, that is one
of the reasons why we were able to reject the null hypothesis when we compare both approaches in general than particularized for systems or tasks regarding correctness analysis. Therefore, we think this lower statistical power represents a threat to be making a type II error.

### 7.1.2 Internal validity

When a relationship between the treatment and the outcome is observed, we must make sure that it is a causal relationship and it is not because confounding variables. If an uncontrolled factor is not addressed, any threats to internal validity may become a conclusion validity in the experiment. These are issues that may indicate a causal relationship when there's none.

- We cannot disregard the maturity of Object-Oriented Paradigm compared to Data Context Interaction. Comparing something new to something already established is our big internal threat to validity. Training subjects in DCI-trygve is required to balance the skill levels in subjects. We were not able to deliver a properly training as we expected. We knew that paradigm shifts take years and introducing subject to a new paradigm would take time. We delivered a 6 pages tutorial document focused on the fundamentals of DCI and how to read trygve language. This threat was not fully eliminate and put DCI-trygve approach in an unfavorable situation over OO-java.

- The instrumentation threat is part of the internal validity. For example if subjects use an IDE with special features, the IDE may become a factor for understanding the code. As we used the same web application to show source code this threat was removed. Before the experiment we added two training system examples for subjects to get used to the web interface and the underlying mechanism of the experiment.

- As our experiment was run over Internet, we were not able to control the surrounded space when subject were running the experiment. This would have been mitigated if we would have run the experiment in a laboratory (Leedy and Ormrod, 2010, p. 235).

- How long is the experiment could influence the results. Subjects mature over the course of the experiment, they can learn about the dynamic of the tasks, they can get bored too. The average time given to subjects in experiments related to code comprehension is 2.1 hs, even though, it appears that there is large variance in duration and that it seems independent of the type of task being performed (Sjøberg et al., 2005, table 10). Our experiment took on average 48 minutes, and
all tasks were presented randomly to dissipate the learning effect between unitary run.

- We avoided showing the experiment as a comparison between a established paradigm a new paradigm that can address the issues in the other. We did not want to perk programmers up during the experiment to do a better job. We approached subjects saying that we are evaluating only DCI. This is related to the novelty effect in human subjects experiments, when the increased interest in a new treatment might produce a better performance.

### 7.1.3 Construct validity

“Threats to construct validity refer to the extent to which the experiment does actually measure what the theory says it does” (Salvaneschi et al., 2014). These are the issues concerned whether the experiment setting actually reflects the constructs under study (Wohlin et al., 2012, p. 103), the issues with the theory and its observation. If we see a causal relationship between treatment and outcome, we must ensure: 1) treatment reflects the construct of the cause, and 2) the outcome reflects the construct of the effect.

- Questions made or the classification of answers to measure source code understanding may not reflect comprehension of the system if not done carefully. We defined specific areas to measure when determining the experiment measurement objectives (Section 5.8) before writing the questions to include in the tasks.

- The representativeness of the system example used is important to reflect the paradigm we are comparing. Three of the DCI-trygve system examples were originally written by professionals that are actually developing trygve language. The spell checker system, Library system, and Bank system. The equivalent java version were reviewed by faculties and professionals to check and maintain its Object-Oriented representativeness.

### 7.1.4 External validity

These threats are related to the validity of generalizing the experiment results. If there’s a causal relationship between the cause and the effect, can the results be generalized outside the scope of the study? Is there a relationship between the treatment and the outcome outside the experiment setting? The external validity is affected by the experiment design, objects and subjects chosen. There are three main risk: wrong participants, wrong environment, and performing the experiment with a timing that affects the results (Wohlin et al., 2012, p. 104).
• The convenience sample with a small number of participants makes hard to generalize the results obtained. Recruiting people is hard and subject drop-outs can unbalance the sample easily. At first we had 26 programmers registered as participants in the experiment, only 17 started the experiment, 15 finished, and finally 12 programmers complete the experiment as expected. We did not use the data from 3 subjects because of different mixed reasons: the skill level of the pre-test was lower than expected showing that fundamental concepts of the paradigm was not understood, and the experiment was complete in an impractical small amount of time. In the samples we had 3 professionals, 3 PhD students, and 6 master students. All master students and 2 PhD students had professional experience, therefore our sample might be representative of professional settings.

• The representativeness of the source for each paradigms also plays a role in the external validity of the results. As we said, the source code for our examples have been reviewed and tested to represent the major characteristics of both approaches. But, another inconvenient is that we are not using real programs or systems, under the experimental lexicon these will be called toy sized systems. We think that larger systems would have been beneficial to observe more clear the differences of both approaches.

7.2 Conclusion

The results presented are preliminary in nature and not suitable of being generalized to other situations. We consider the results preliminary because small sized samples drive to results with low statistical power, i.e. results are sensitive to change when adding new subjects that performs different than the rest. We consider the results difficult of being generalized mostly because we used a convenience sample that hardly represent other scopes. That is why we regard our results a start that serves as a first exploration on the surface of Data Context Interaction paradigm and trygve language regarding code comprehension.

On the other hand, there is a general unbalanced situation against DCI-trygve approach. Java language is ubiquitously used around the world and trygve is just known in a small community. Due to the small sample size and the difference in maturity of the approaches, we consider that it is more probable to make type II error when testing our hypothesis. Type II error is failing to reject a null hypothesis when the alternative hypothesis should have been accepted.

Although the inherent issues above mentioned, both groups resulted well balanced. As we did a pre-test to measure the skill level in the paradigm and language, we computed the results of the tests and obtained a balanced skill level for both groups.
The average for both groups was 8.16 out of 11 questions (74.2%). There were more variability in the scores for the DCI group, from 5 to 11, than in the OO group with a skill level from 7 to 10.

Something we observed about the answers of subjects from the DCI group was that members selected more frequently a “don’t know” option than subjects in the OO group. There are 16 “don’t know” answers submitted by subjects from the DCI group, whereas 9 “don’t know” answers were submitted from the OO group. This was expected, as DCI-trygve is a new approach, we think, subjects felt free to answer with a “don’t know” option. This is actually a good symptom that shows that subjects were not willing to guess when using the new paradigm. The “don’t know” option were given to subject as a way to avoid the noise that guessing would generate in the results. It was followed by a message to make clear that in the experiment we were not grading people, and a “don’t know” answer is always better than a guess.

Regarding the research questions, the correctness analysis shows that in general programmers following the DCI-trygve approach perform better than programmers using the OO-java approach. When particularizing by system examples, only one system shows better results with statistical significance for the DCI group, for the rest of the systems we cannot state any difference. Similar situation resulted when the correctness analysis was particularized for task types. This seems to be an effect of the size of the samples where small values from different partitions get accumulated to reach statistical significance in the whole. This may be a symptom of the low statistical power and the small differences in favor to DCI-trygve approach that might be an indicator of type II error. A way to avoid this issue is to replicate the experiment with a large sample.

When analyzing the correctness per system example, we can observe that the three most complex and large systems get lower p-values when testing the null hypothesis. In the most basic system, the Bank system, OO-java group performed better in term of sample means. Bank system happen to be the smallest system: 72 SLOC for OO-java, and 112 SLOC for DCI-trygve. It represents the larger difference of SLOC between the two versions in the smallest system example. In the Bank system, for each SLOC of java we have approximately 1.6 SLOC of trygve code. On the other hand, the Menu system is the only system which we can state that DCI-trygve group performs better with statistical significance regarding correctness. This system is the larger system example with 232 SLOC for OO-java and 250 SLOC for DCI-trygve. This is aligned to what DCI proponents say, that the benefits of DCI will pay off when applied to large systems with complex interactions. The Data Context Interaction paradigm addresses an underlying problem observed in Object-Oriented systems, this is the lack of separation between data and behavior at system level. Because classes are the fundamental building block in classical OO, if not the unique as in java, a system functionality gets spread
in classes defining small tiny parts of a system operation in each class. “DCI sees a program in two orthogonal projections. The Data projection describes system state. The Context projection describes system behavior; there is one of the latter for each use case scenario” (Reenskaug and Coplien, 2009a). As small system functions have small content spread in very few classes, small system examples might not be indicated to observe DCI in all its power when compared to OO approach. As a recommendation of when to use a DCI approach, Coplien wrote in his book Lean Architecture for Agile Software Development: “If a system or subsystem has a critical mass of scenarios and use cases that reflect sequences of conscious user intents, then design that entire subsystem using the DCI architecture” (Coplien and Bjørnvig, 2011). Unfortunately, today there is no large complex systems written in trygve language to observe code comprehension, but larger source code would allow us to observe better the characteristics of DCI.

When correctness was particularized for the type of task, we have only one case with a statistically significant difference. This is for the task called: look for changed and unchanged objects. As explain in Section 3, trygve language distinguish two type of roles regarding the possibilities to mutate the state of an object by executing a context, these are role and stageprop. Roles establish the behavior of objects regarding the contexts where they are defined and have the information about which data of the role-player object can be accessed during the execution of the context. A stageprop role is capable only to access data through accessor methods defined in the role-player object. Thus, The stageprop role indicates a free-side effect behavior on objects. This seems to be a rich information on understanding the state of objects by reading a single keyword in the code that helps programmers to know if an object can mutate its state during a system operation execution, thus explaining the results we obtained.

For the centrality analysis we conclude that the DCI-trygve context language element is functioning as a central source of information that is essential to understand the system, whereas in OO-java approach this information is spread among class language elements. This conclusion is highly dependent on the characteristic of the source code that describes the system. The DCI-trygve approach concentrates the focus of attention of programmers in one file, this is the most important file regarding the eigenvector centrality metric and the file on which subjects spent more time reading source code. The centrality metric and reading time are strongly correlated in both approaches, but, with the difference that in the OO-java approach higher centrality degree is shared between two and four files. As we use representative system examples for both versions, our results are valid for both paradigms in different conditions than the experimental setting. We also think that this result will be more remarkable if we increase the size and complexity of the experimental units.

The centrality metric applied in the Cognitive Network of Language Elements built by programmers during program comprehension seems to detect the most important
place in the source code regarding the information needed to understand a system operation. This is related to the locality of a system functionalities in DCI, where contexts exacerbates the differences between the importance of files to understand a system operation and stand at the center of attention. In a DCI system we can easily find the location for each system functionality in context declarations. This represents an advantage regarding traceability, where use cases are directly written in the code. Our Cognitive Network of Language element resulted valuable to state with high statistical significance that the context element in a DCI system plays an important role in code comprehension.

7.3 Future work

We gained plenty of knowledge from the experience of preparing, designing, and running the experiment. The knowledge obtained by comparing and developing the both version of a same system was revealing to understand the DCI paradigm. We will continue working in code comprehension, specially in large systems. As future work we plan to:

- Investigate deeply the possibilities that CNLE represents to study the properties of the code regarding code comprehension. We plan to study centrality metrics for whole networks, a.k.a. centralization, to characterize a CNLE as a whole.

- We plan to run a replication of the experiment with an adequate sample size and a better training on subjects to enhance the statistical significance and raise the statistical power of the results.

- We plan to study deeper the characteristics of Data Context Interaction to understand better the benefits of adopting this new paradigm.
Appendix A

Supporting data

A.1 Code metrics

Table A.1 is a quantitative description of all the system example included in the experiment. As each system has two equivalent versions, each item we count has two columns, one for OO-java and one for DCI-trygve. In the last column we show the amount of language elements involved in the description of the system, it should be noted that here we are counting roles separate from the contexts. Even though a context and its roles are indivisible elements, for the system designer different roles denote different behavioral concerns.

<table>
<thead>
<tr>
<th>System</th>
<th>SLOC</th>
<th>classes</th>
<th>interfaces</th>
<th>contexts</th>
<th>roles</th>
<th>files</th>
<th>lang elements*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>java</td>
<td>trygve</td>
<td>java</td>
<td>trygve</td>
<td>java</td>
<td>trygve</td>
<td>java</td>
</tr>
<tr>
<td>Bank</td>
<td>72</td>
<td>112</td>
<td>3</td>
<td>2</td>
<td>n/a</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td>Library</td>
<td>225</td>
<td>206</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Menu</td>
<td>232</td>
<td>250</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Spell checker</td>
<td>237</td>
<td>224</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
<td>202</td>
<td>220</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>total</td>
<td>968</td>
<td>1012</td>
<td>27</td>
<td>16</td>
<td>5</td>
<td>2</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Table A.1:** Simple code metrics per system example.

A.2 Experimental tasks metrics

In Table A.2 is shown all comprehension questions that each tasks comprises for each system example. Each task includes from 2 to 6 comprehension questions. Both version of a system has the same amount of questions but the amount of questions per system sometimes varies, although we maintain the same amount of comprehension tasks among system examples.

<table>
<thead>
<tr>
<th>System</th>
<th>Check implemented features</th>
<th>Describe object interactions</th>
<th>Look for changed and unchanged objects</th>
<th>Sort execution flow</th>
<th>Total questions per system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>Library</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Menu</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>Spell checker</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>total questions per tasks</td>
<td>30</td>
<td>23</td>
<td>14</td>
<td>26</td>
<td>93</td>
</tr>
</tbody>
</table>

**Table A.2:** Experimental tasks per system example.
Appendix A. Supporting data

A.3 Centrality and time

**Figure A.1:** Centrality-Time analysis Bank system.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>creditor.k</td>
<td>0.110</td>
<td>0</td>
<td>0.023</td>
</tr>
<tr>
<td>pay_bill.k</td>
<td>0.245</td>
<td>0</td>
<td>0.034</td>
</tr>
<tr>
<td>account.k</td>
<td>0.253</td>
<td>0.03</td>
<td>0.037</td>
</tr>
<tr>
<td>main.k</td>
<td>0.427</td>
<td>0.15</td>
<td>0.037</td>
</tr>
<tr>
<td>transfer_money.k</td>
<td>0.950</td>
<td>0.84</td>
<td>0.022</td>
</tr>
</tbody>
</table>

**Table A.3:** Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve).

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>creditor.java</td>
<td>0.183</td>
<td>0</td>
<td>0.022</td>
</tr>
<tr>
<td>IAccount.java</td>
<td>0.465</td>
<td>0.16</td>
<td>0.037</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.656</td>
<td>0.47</td>
<td>0.037</td>
</tr>
<tr>
<td>Account.java</td>
<td>0.822</td>
<td>0.61</td>
<td>0.037</td>
</tr>
<tr>
<td>Bank.java</td>
<td>0.863</td>
<td>0.71</td>
<td>0.054</td>
</tr>
</tbody>
</table>

**Table A.4:** Wilcoxon rank signed test for centrality segments - Bank system (OO-java).
Appendix A. Supporting data

Figure A.2: Centrality-Time analysis Library system.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>book.k</td>
<td>0.054</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>screen_printer.k</td>
<td>0.055</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>item_record.k</td>
<td>0.087</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>main.k</td>
<td>0.109</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>mock_card_reader.k</td>
<td>0.121</td>
<td>0.00</td>
<td>0.022</td>
</tr>
<tr>
<td>paper_printer.k</td>
<td>0.156</td>
<td>0.37</td>
<td>0.037</td>
</tr>
<tr>
<td>mock_book_scanner.k</td>
<td>0.227</td>
<td>0.02</td>
<td>0.023</td>
</tr>
<tr>
<td>borrow_library_items.k</td>
<td>1.000</td>
<td>0.99</td>
<td>0.011</td>
</tr>
</tbody>
</table>

Table A.5: Wilcoxon rank signed test for centrality segments - Library system (DCI-trygve).

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypad java</td>
<td>0.073</td>
<td>0.02</td>
<td>0.037</td>
</tr>
<tr>
<td>ScreenPrinter java</td>
<td>0.105</td>
<td>0.03</td>
<td>0.037</td>
</tr>
<tr>
<td>Book.java</td>
<td>0.129</td>
<td>0.08</td>
<td>0.037</td>
</tr>
<tr>
<td>ItemRecord java</td>
<td>0.167</td>
<td>0.03</td>
<td>0.037</td>
</tr>
<tr>
<td>PaperPrinter java</td>
<td>0.280</td>
<td>0.19</td>
<td>0.023</td>
</tr>
<tr>
<td>MockCardReader java</td>
<td>0.427</td>
<td>0.29</td>
<td>0.023</td>
</tr>
<tr>
<td>Screen.java</td>
<td>0.572</td>
<td>0.28</td>
<td>0.037</td>
</tr>
<tr>
<td>MockBookScanner java</td>
<td>0.668</td>
<td>0.38</td>
<td>0.037</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.949</td>
<td>0.84</td>
<td>0.020</td>
</tr>
</tbody>
</table>

Table A.6: Wilcoxon rank signed test for centrality segments - Library system (OO-java).
Appendix A. Supporting data

**Figure A.3:** Centrality-Time analysis Menu system.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>menu_structure.k</td>
<td>0.169</td>
<td>0.27</td>
<td>0.037</td>
</tr>
<tr>
<td>menu_component.k</td>
<td>0.240</td>
<td>0.09</td>
<td>0.023</td>
</tr>
<tr>
<td>menu_item.k</td>
<td>0.261</td>
<td>0.11</td>
<td>0.023</td>
</tr>
<tr>
<td>main.k</td>
<td>0.325</td>
<td>0.56</td>
<td>0.037</td>
</tr>
<tr>
<td>menu_interface.k</td>
<td>0.972</td>
<td>0.91</td>
<td>0.020</td>
</tr>
</tbody>
</table>

**Table A.7:** Wilcoxon rank signed test for centrality segments - Menu system (DCI-trygve).

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuComponent.java</td>
<td>0.436</td>
<td>0.21</td>
<td>0.037</td>
</tr>
<tr>
<td>MenuItem.java</td>
<td>0.453</td>
<td>0.23</td>
<td>0.037</td>
</tr>
<tr>
<td>MenuStructure.java</td>
<td>0.521</td>
<td>0.28</td>
<td>0.037</td>
</tr>
<tr>
<td>MenuInterface.java</td>
<td>0.783</td>
<td>0.59</td>
<td>0.030</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.979</td>
<td>0.93</td>
<td>0.02</td>
</tr>
</tbody>
</table>

**Table A.8:** Wilcoxon rank signed test for centrality segments - Menu system (OO-java).
**Figure A.4:** Centrality-Time analysis Spell Checker system.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>book.k</td>
<td>0.107</td>
<td>0.02</td>
<td>0.023</td>
</tr>
<tr>
<td>book.k</td>
<td>0.145</td>
<td>0.06</td>
<td>0.037</td>
</tr>
<tr>
<td>screen_printer.k</td>
<td>0.260</td>
<td>0.16</td>
<td>0.037</td>
</tr>
<tr>
<td>item_record.k</td>
<td>0.289</td>
<td>0.10</td>
<td>0.037</td>
</tr>
<tr>
<td>main.k</td>
<td>1.000</td>
<td>0.99</td>
<td>0.012 NA</td>
</tr>
</tbody>
</table>

**Table A.9:** Wilcoxon rank signed test for centrality segments - Spell checker system (DCI-trygve).

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDocument.java</td>
<td>0.079</td>
<td>0.16</td>
<td>0.022</td>
</tr>
<tr>
<td>Utilities.java</td>
<td>0.0965</td>
<td>0.03</td>
<td>0.023</td>
</tr>
<tr>
<td>Range.java</td>
<td>0.144</td>
<td>0.01</td>
<td>0.023</td>
</tr>
<tr>
<td>Dictionary.java</td>
<td>0.172</td>
<td>0.07</td>
<td>0.014</td>
</tr>
<tr>
<td>Word.java</td>
<td>0.181</td>
<td>0.02</td>
<td>0.023</td>
</tr>
<tr>
<td>MutableList.java</td>
<td>0.185</td>
<td>0.07</td>
<td>0.037</td>
</tr>
<tr>
<td>Document.java</td>
<td>0.422</td>
<td>0.21</td>
<td>0.037</td>
</tr>
<tr>
<td>OnlineDictionary.java</td>
<td>0.539</td>
<td>0.18</td>
<td>0.037</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.634</td>
<td>0.29</td>
<td>0.014</td>
</tr>
<tr>
<td>SpellChecker.java</td>
<td>0.991</td>
<td>0.97</td>
<td>0.020</td>
</tr>
</tbody>
</table>

**Table A.10:** Wilcoxon rank signed test for centrality segments - Spell checker system (OO-java).
**Figure A.5:** Centrality-Time analysis Tower of H. system.

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rod.k</td>
<td>0.135</td>
<td>0.00</td>
<td>0.014</td>
</tr>
<tr>
<td>disk.k</td>
<td>0.165</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>counter.k</td>
<td>0.181</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>game_board.k</td>
<td>0.257</td>
<td>0.07</td>
<td>0.037</td>
</tr>
<tr>
<td>main.k</td>
<td>0.277</td>
<td>0.03</td>
<td>0.023</td>
</tr>
<tr>
<td>three_rod_solver.k</td>
<td>0.312</td>
<td>0.09</td>
<td>0.037</td>
</tr>
<tr>
<td>play_game.k</td>
<td>1.000</td>
<td>0.99</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

**Table A.11:** Wilcoxon rank signed test for centrality segments - Spell checker system (DCI-trygve).

<table>
<thead>
<tr>
<th>File</th>
<th>Centrality mean</th>
<th>End points</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Interface.java</td>
<td>0.037</td>
<td>0.0</td>
<td>0.022</td>
</tr>
<tr>
<td>Input Interface.java</td>
<td>0.075</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>Input.java</td>
<td>0.081</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>Disk.java</td>
<td>0.152</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>Rod.java</td>
<td>0.196</td>
<td>0.0</td>
<td>0.014</td>
</tr>
<tr>
<td>Output Screen.java</td>
<td>0.246</td>
<td>0.02</td>
<td>0.037</td>
</tr>
<tr>
<td>Three Rod Solver java</td>
<td>0.291</td>
<td>0.15</td>
<td>0.023</td>
</tr>
<tr>
<td>Counter.java</td>
<td>0.388</td>
<td>0.09</td>
<td>0.037</td>
</tr>
<tr>
<td>Main.java</td>
<td>0.437</td>
<td>0.24</td>
<td>0.0016</td>
</tr>
<tr>
<td>GameBoard.java</td>
<td>1.000</td>
<td>0.99</td>
<td>0.007</td>
</tr>
</tbody>
</table>

**Table A.12:** Wilcoxon rank signed test for centrality segments - Tower of H. system (OO-java).
Bibliography


Jacobson, Ivar (1992). “Object oriented software engineering: a use case driven approach”. In:


Reenskaug, Trygve (2003). “The model-view-controller (mvc) its past and present”. In: *University of Oslo Draft*.


Vlissides, John et al. (1995). “Design patterns: Elements of reusable object-oriented software”. In: Reading: Addison-Wesley 49.120, p. 11.


