Characteristics of mental representations in Novices and Experts in Java

by

Mohammadhossein Parastar

Bachelor of Software Engineering, IAU, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Eric Aubanel, PhD, Faculty of Computer Science
Examining Board:	Arash Habibi Lashkari, PhD, Faculty of Computer Science, Chair Dawn MacIsaac , PhD, Faculty of Computer Science Daniel Voyer, PhD, Faculty of Psychology

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

January, 2021

© Mohammadhossein Parastar, 2021

Abstract

How programmers mentally represent code is of interest to researchers who study program comprehension and design new programming languages. In 1993 Wiedenbeck et al. conducted a study on characteristics of mental representation using a procedural language, and five characteristics were introduced. We designed an experiment using JAVA to determine whether programmers using an object-oriented language have the same characteristics as programmers using a procedural language. We considered two alternative definition of expertise: expertise as years of experience and expertise determined by self-assessment. We used the same Multivariate and Univariate analysis as the previous study and in addition used Mixed Effect Logistic Modeling to analyze the results. We found a significant difference between experts and novices defined using self-assessment in Linear Modeling. Our results did not fully agree with the previous research. Our study supports the existence of *recurring basic patterns* and *well-connected* representations. However, we could not find support for *hierarchical structure*, *grounding in the program text*, and found an unexpected result in *mapping code to goals*. Our study had several limitations, such as the Corona-virus pandemic that caused the limit in the number of participants, artificiality of the tasks, and a lack of professional programmers in the experiment sample.

Dedication

I dedicate this thesis to my parents, HamidReza and Vida, for all their love and support.

Acknowledgements

I would first like to thank my supervisor Dr.Eric Aubanel. The door to Dr.Aubanel's office was always open whenever I ran into a trouble or had a question about my research or writing. He consistently allowed this research to be my work but steered me in the right direction whenever he thought I needed it. This thesis was not possible without the expertise, guidance and financial support of Dr.Aubanel.

This experiment involved many people in different stages, thanks to Prof. Leah Bidlake for her comments and help in questionnaire design, all the participants required to make this experiment happen, all my friends who helped me throughout the project, especially Dr.Mohammad Keshavarz for his guidance in analysis and Ms.Samin Fakharian for motivating me in my study.

Thanks to University of New Brunswick and department of Computer Science for giving me this opportunity to continue my study in Master of Computer Science.

Last but not least I would like to thank my family for their support before and after I started my studies as an international student in Canada. I could not continue my study if it wasn't for their support.

Table of Contents

Abstract ii

Dedication iii

Acknowledgments iv

Table of Contents v

List of Figures viii

Abbreviations ix

1 Introduction 1

1.1 Problem statement 1

1.2 Contribution 2

2 Background 4

2.1 Program comprehension 4

2.1.1 What is program comprehension? 4

2.1.2 Why is program comprehension important? 5

2.1.3 Aspects of program comprehension: 6

2.1.3.1 Comprehension as a process 6

2.1.3.2 Comprehension strategies 9

2.1.3.3 Programmer's knowledge in comprehension 9

2.1.4 Program comprehension and mental representation 11

2.2 Definition of expertise 14

3 Related Work 16

3.1 Characteristics of mental representation experiment done for procedural programming 16

3.2 Object-oriented and procedural programming differences 20

4 Work and Objectives 23

4.1 Experiment design 23

4.1.1 Methodology 23

4.1.2 Participants 23

4.1.3 Material 24

4.1.4 Procedure 31

4.1.5 Scoring 32

5 Experimental Results 33

5.1 Demographic Analysis 33

5.2 Expertise discussion 34

5.3 Analysis 1 35

5.4 Analysis 2 35

5.5 Analysis of results 39

6 Conclusion 45

6.1 Conclusion 45

6.2 Limitations 48

6.3 Future Work 49

Bibliography 54

A Program description 55

B Code **56**

C Questions **79**

D R Scripts **83**

Vita

List of Figures

4.1 Battleship program UML 28

5.1 Beginning year of study 34

D.1 R scripts used for analysis 84

List of Symbols, Nomenclature or Abbreviations

OO \Object Oriented

Chapter 1

Introduction

1.1 Problem statement

Dividing programmers into different groups, like experts and novices, is always very hard. This is because there are a lot of different variables that could affect someone's expertise [14]. Additionally, the definition of expertise is always arguable [40]. One of the ways to define expertise is via program comprehension [40]. We can divide programmers into different groups based on their performance in program comprehension [40]. Our study uses the five characteristics of mental representation introduced by Wiedenbeck et al. [40] to characterize expertise in object-oriented programmers. We hypothesize that experts would have these five mental representation characteristics, but novices would not. Debugging or modifying code is an essential part of programming, and programmers need to understand the code to be able to debug or change it. When programmers read lines of code, a representation of the code is made in their minds [40][19]. All the future debugging and modification of the code will be based on the programmer's comprehension and mental representations. A mental representation is based on an internal model of the program [40]. This mental model is made from the information, specifications and documentation

of the target program [40]. In this experiment, we are mostly concerned about the characteristics of mental representations that novices and experts make while they are studying an object-oriented program.

1.2 Contribution

Finding what information and details programmers consider in the comprehension of a program, and the characteristics of their mental representation can help us better understand the tasks related to the process of comprehension[40]. A number of studies have focused on comprehension and mental representation. Wiedenbeck et al. [40] addressed characteristics of mental representations of novice and expert programmers [40], but their empirical study is based on procedural programming. However, the characteristics of mental representation in object-oriented programming (OOP) may be partly or entirely different.

Significance of study

This thesis concerns the characteristics of the mental representation of novice and expert programmers in OOP using Java. This research will help our understanding of the program comprehension of novice and expert object-oriented programmers. It is based on the original paper by Wiedenbeck et al. [40]. The original paper proposed five characteristics for mental representations [40] (p. 2):

1. "Hierarchically structured mental representation."
2. "Explicit mapping between different layers."
3. "Founded on the recognition of basic patterns."
4. "Well-connected internally."
5. "Well-grounded in the program text."

The expertise of participants in our study is based on programmers' years of experience and self assessment. We find the differences in procedural and object-oriented

programmers' mental representations by comparing how they comprehend a program. Wiedenbeck et al. proposed that procedural language experts would have the five characteristics, and novices would lack them [40]. We examine the same hypothesis for novices and experts in OOP. Past studies reveal that procedural and object-oriented language programmers have differences in program comprehension (e.g. [12]), therefore, our study could show different results, and we may not observe all the characteristics found in Wiedenbeck et al.'s study.

Chapter 2

Background

In this chapter, we discuss the theories and previous studies in program comprehension, mental model theories and mental representations. We need to understand what program comprehension means and the importance of studying it. We also address differences in the program comprehension and mental representation of novices and experts. We discuss definitions of expertise and our definition of expertise in this thesis.

2.1 Program comprehension

2.1.1 What is program comprehension?

Program comprehension is a way to gain information about code. By acquiring more information about the code, programmers can perform better in activities like bug correction, enhancement, reuse, and documentation [30]. Since the majority of work on software development is maintaining existing programs, rather than developing new ones [30], program comprehension is critical in the field of software development; Fjeldstad and Hamlen reported that 47% of enhancement time and 65% of correction tasks are dedicated to comprehension tasks[16]. If we want to improve

software development, we should understand program comprehension better [30]. It is hard to define program comprehension exactly, and papers on program comprehension find it is like defining the meaning of reading. This can be because program comprehension includes a vast range of activities and tasks [17]. According to Pennington and Grabowski: "Understanding a program involves assigning meaning to a program text." (p. 54) [27] The definition of program comprehension is not unique because of the wide range of activities it could involve; for example, it can mean comprehending a whole program or looking for specific information from some part of a program or getting a general understanding of the program. It also can mean understanding one's own written program or comprehending someone else's program or comprehending a program for a task. In this thesis, we define program comprehension as an understanding of written code, whether it is a snippet of code from someone else or the programmer's own code. Understanding means finding out how different parts of code work together, and the purpose and overall goal of the code.

2.1.2 Why is program comprehension important?

Program comprehension is essential in any task related to programming and teaching programming languages and concepts to new learners. Many programming courses and languages start with a simple program comprehension printing "Hello World!" so the new learners will comprehend how it is written and try to write similar programs [17]. Thus one of the important usage of program comprehension research can be in teaching. Studying program comprehension can help developers gain better performance in debugging a program, which can be someone else's code, but they need to comprehend it first and then try to debug it. So it is important to understand how developers comprehend a program and what kind of mental models and representations they have and how they build them mentally.

2.1.3 Aspects of program comprehension:

Judith Good has considered different aspects of program comprehension according to the way it has been studied:

- "Comprehension is a process"
- "Comprehension strategies"
- "The role of the programmer's knowledge in comprehension"
- "The programmer's mental representation or model of the program"
- "Comprehension is the search of information in the program" (p. 18) [17]

These aspects are not unique, and some studies will cover more than one aspect, like the studies that Pennington did[25].

2.1.3.1 Comprehension as a process

Some theories of program comprehension describe program comprehension using "temporal theories" [17]. This means that program comprehension is defined based on the assumption of sequence [17], expressing the task and activities programmers do to comprehend a program and the order in which the activities are done. These theories can be divided into three different aspects of program abstraction[17]:

1-Top-down theories: These theories are based on progression from a high level of program abstraction to a lower level.

2-Bottom-up theories: The hypothesis is the progression from low entities in a program to a higher level of abstraction.

3-Mixed theories: The hypothesis concerns higher movement between the abstraction levels of the program.

Top-down models of program comprehension

Some theories approach program understanding based on the top-down model. Brooks [10] discussed program comprehension as the inverse of the coding process. Brooks described coding as a mapping from the problem domain to the programming domain

[17][10] and a series of intermediate domains in between them. Program comprehension means rebuilding of these mappings in programmers' minds. In this model, comprehension is defined as a hypothesis verification process [10]. When programmers hear the name of a program in their mind, they will construct a hypothesis about the program [10]. This hypothesis can progress to construction of a subsidiary tree of hypotheses generated in a top-down and depth-first manner [10]. In this process when the programmer matches the details with the program text, "hypothesis verification will happen by searching for beacons" (p. 20) [17]. Brooks defines a beacon as "typical indicators of the use of a particular operation or structure" (p. 6)[10], so when beacons which relate to the hypothesis are found, it means that the hypothesis is confirmed. When searching for beacons, programmers can find new subsidiary hypotheses and the codes that confirm the subsidiary hypotheses will become connected to them; then, a tree-like formation will be created in which leaf nodes are code segments and then program comprehension is complete [17].

Bottom-up models of program comprehension

In these models of program comprehension, programmers make a representation based on the program text [17]. Pennington [25] defined two distinct but related representations of text that are based on previous works done by Kintsch and van Dijk [36] and van Dijk and Kintsch [37]: the textbase or program model and situation model. Pennington found that the program model is made first based on a procedural reading of the program and expressed using the programming language. The situation model is formed from the program model and features functional connections between domain objects.

In Pennington's theory of program comprehension, programmers comprehend a program from the bottom up, so they divide the program into small control parts (program model) and make assumptions about each part individually [27]. Then they consider function and data flow in each part (situation model).

Mixed Theories of program comprehension

One of the theories in this area is Letovsky's model [35]. This model complements the previous model by Littman [21]. Letovsky's model is considered a mixed model that describes program comprehension both as a top-down and bottom-up model; it is one of the most complete and flexible program comprehension models [17]. Letowsky defined a programmer as a knowledge base understander who has three entities: 1- a knowledge base, which means prior programming knowledge and expertise; 2- a mental model, which is the current understanding of the program in question; 3- an assimilation process, which means constructing the mental model by interaction with the code, documentation and knowledgebase [17][35]. A knowledgebase comprises a wide range of concepts like programming language semantics, plans, objectives and domain knowledge. The mental model is described as a three-layered network: 1- specification layer at top of the model, which contains the goal of the program; 2- implementation layer at the bottom, which describes the actions and structures; 3- annotation level, which connects the two other levels. This model is incomplete at the beginning, and it becomes complete as a programmer develops connections from each side; if it starts from the upper level (specification to implementation), it is a top-down model, and if it starts from the implementation level to specification, it will be considered a bottom-up model.

Letovsky's model describes novice programmers' problems. Studies show that novices are not able to use the top-down model because they do not have enough knowledge to make the hypotheses, so they will comprehend the program by code and use the bottom-up model [21].

The Boehm-Davis model is also a variation of Letovsky's model in many respects [9]. She proposed that programmers follow an iterative segmentation, hypothesis generation and verification process. Programmers will use previous knowledge with the program's information to divide the code into manageable pieces, then construct

the hypotheses about the program function and verify it with the code [9].

2.1.3.2 Comprehension strategies

Many theories and models focus on the programmer comprehending an entire program. Mayrhauser and Vans mentioned that these models might not be useful in a situation where only part of a program needs to be comprehended by the programmer; for example, finding a bug in a very large program [38], where comprehension strategies describe program comprehension based on the debugging of the code.

According to Littman et al. [28], there are two types of strategies in program comprehension: 1- systematic strategy; 2- as-needed strategy. The programmer will try to understand and study the entire program before modifying or debugging it in the systematic strategy. In the as-needed strategy, the programmer will only focus on the part that needs to be modified [35]. Based on these strategies, two types of knowledge could be extracted from the program: 1- static knowledge, which includes the actions and functions of the program and causal knowledge, which includes the connection and interaction between the actions and functions of a program [35]. Causal knowledge can be acquired by mentally running the data-flow and controls between components [35].

The authors also mentioned that strategies could impact on the knowledge that the programmer will get from the program, and this knowledge would be formed as a weak or strong mental model [35]. The weak mental model comes from using an as-needed strategy, while the strong mental model will come from the systematic strategy which contains both static and causal knowledge.

2.1.3.3 Programmer's knowledge in comprehension

Based on Good's [17] division, another aspect of program comprehension covers the programming knowledge of programmers that influences their comprehension. One

of the theories of the whole process of programming comprehension is Shniderman and Mayer's theory, which focuses on different types of memory in programmers (short-term, long-term and working memory) [31]. This theory describes the process and tasks performed in the programmer's long-term memory to comprehend a program. The authors define two types of knowledge in long-term memory: semantic knowledge and syntactic knowledge. Semantic knowledge includes concepts and basics of programming and is not dependent on the language, such as understanding of loop concept or recursive functions in code. Syntactic knowledge is language-dependent, for example, the syntax of a language to define an array [31].

Shneiderman and Mayer believe that programmers have a multi-leveled model of the program made by their syntactic knowledge, and each level consists of a semantic representation of the program[31]. They believe that this structure consists of small to large chunks of information from low to high levels, which constitute the full knowledge of the program[31]. Shneiderman and Mayer believe that full knowledge may be acquired by comprehending a high-level representation of the program but without fully understanding lower levels [31].

Plans

One of the most important concepts used in program comprehension theory is programming plans. Soloway studied the psychological aspects and basis in different studies [32][33][34], and Soloway's definitions are the basis of other studies about program comprehension [17]. The definition of plans is not the same in all of Soloway's studies. Soloway and Ehrlich define plans as "program fragments that represent stereotypic action sequences in programming" (p. 1) [34], while another study defines a plan as "a procedure or strategy in which the key elements of the process have abstracted and represented explicitly." (p. 30) [32]. The latter study also introduces three different plans: strategic plans, tactical plans, and implementation plans [32]. The strategic and tactical plans are language-independent: a strategic plan is about

the higher level or algorithmic level of a program, while the tactical level refers to smaller segments of a program; Implementation plans depend on the language and are used to understand the two other plans in a program [32].

From Soloway et al., there is another type of knowledge related to expert programmers called the rules of programming discourse, which means common knowledge or understanding of coding. For example, using appropriate variable names or making sure that the program is as efficient as possible [33]. Code that obeys these rules are plan-like, and the code or program that violate them are unplan-like [33].

Soloway conducted this experiment by showing participants (one group of novices and one group of experts) two versions of a program: one following the programming discourse and the other violating it. The participants were asked to fill in the blank part of the program. Results showed that experts had a 53% decrease in performance in unplan-like programs, while novices had a 41% decrease. Both novices and experts had more errors and took more time to complete the unplan-like programs. When the programmers did not know the program's goals, they used the bottom-up process and started to read the code from the beginning [17][33], but when the participants had even a slight understanding of the program plans, they used the top-down process in order to comprehend the code. The unplan-like program forced participants to use the bottom-up process because they did not have any understanding or plan knowledge about the program [33].

2.1.4 Program comprehension and mental representation

We discussed program comprehension based on the previous knowledge of a programmer. Every program and line of code that a programmer reads produces a mental model in their mind, helping the programmer comprehend the code. Researchers in this aspect of program comprehension mostly focused on experiments in which a

programmer studies a program and then answers questions related to the program or modifies or debugs it and then finally answers a questionnaire. These empirical studies could help the researchers to understand the mental representation and models of programmers.

Differences between novices and experts

Studies in mental representation often consist of two groups of programmers, novices and experts [8]. Although there are some different considerations in definitions of expertise, they usually consider new programmers and students without work experience as novices [6][40] while experts would be programmers with years of experience in programming, or teaching experience in programming [6] [40].

Adelson proposed that there are differences between novices and experts in mental representation [6]. She believed that novices form a concrete representation of the program that focuses on how the program works, while experts have an abstract representation based on what the program does [6]. In her experiment, Adelson chose novices from undergraduate students who had an introductory course in computer programming and experts were selected from individuals who were teaching programming. She used flowcharts to test her hypothesis about concrete and abstract mental representations in programmers [6]. She defined abstract flowcharts as flowcharts that specify what the program does and do not mention about how the program functions, while the concrete flowcharts show the line by line description of the program without specifying why the action is happening and what is the use of the following line of code in overall goal. She found that experts did much better at abstract questions and novices do better at concrete questions; it suggests that experts focused on the goal of the program and what the program does, although the results of different tasks in Adelson's experiment suggested that experts can have good performance at concrete questions too, but they have learned that it is more important to look for the overall goal of the program [6].

Holt et al. also worked on the differences of mental representation between novices and experts in program comprehension [18]. Their experiment was designed based on three different modification tasks, going from easy to difficult, and in three different designs: 1- in-line code, 2- functional decomposition, 3- object-oriented. The participants were asked to answer questions about remembering the program components and also the relationships and connections between them [18].

Holt et al. found out that there is no significant difference between novices (students) and experts (professional programmers) in these tasks and their structure of mental representations, but they found that the mental representations of experts were more complex while doing complex modification and novices' representations were different based on the structure of the program (serial, functional or object-oriented) [18].

Pennington experiments: Pennington studied mental representations using information types. She used two mental representations of a program: 1-the textbase, 2-the situation model [25]. Pennington defined the textbase model as "a hierarchy of representations consisting of a surface memory of the text, a micro-structure of interrelations among text propositions, and a macro-structure that organizes the text representation" (p. 101) and the situation model is a mental model of what the text is about referentially [25]. Information types have been described as different kinds of information in the text that must be detected to comprehend the program [25]. Pennington mentioned that information types are based on formal analyses of programs [25]. She defined five types of information:

Function: This information type include the purpose and goal of the program [25]. These goals can be hierarchical and can also occur in different orders, but this information type does not indicate how the program is implemented.

Control flow: This information type defines the sequence of execution of a program. For example: what happens after using a method and what happened before

entering a For loop. This information type focuses on the control sequence of events, not the flow of data in the program [25].

Data flow: This information type examines the transformation and sequence of data. These data can include data dependencies and data structures information. For example: Does passing X to a method affect the return value of that method? [25]

Operations: This information type describes the actions and effect of a particular part of a code, and can be as small as a single line. Pennington described it as actions that happen in a program [25]. For example: Does a variable increase after each loop cycle? [25]

2.2 Definition of expertise

There are two approaches to the study of expertise: the absolute approach and the relative approach. The first approach measures people's expertise by their exceptional understanding in their domain of expertise [24]. The other approach studies expertise in comparison to novices. This method defines expertise as a level of proficiency that novices can attain. Different measures, like academic qualifications, can assess the proficiency level [12][24].

One important advantage of the relative method is that you do not need to be so precise about someone's expertise. Ericson and Smith defined expertise as outstanding performance [14]. Expertise can be developed as the years of experience in a specific domain increase [24]. However, in more advanced developers, more experience does not necessarily mean higher performance. Also, high performers are not necessarily more experienced [24]. In our research, we use the relative approach, and we define the expertise in two different approaches. First, as a specific qualification

that each participant should have, in our case, years of experience in a programming job. Specifically students who have more than two terms of co-op or participants who had experience working as a programmer using an Object-Oriented Language for more than a year, were considered to be experts. Second, we used self estimation of expertise extracted from the demographic questionnaire. Feigenspan et al. discussed measuring programming experience with different methods of assessment like years of experience, education, self estimation, questionnaire, size of code and etc. [15]. They found that self estimation of participants is a reliable way of measuring programming experience [15].

Chapter 3

Related Work

Program comprehensions and mental models have been studied for many years and the newer studies are mostly based on the previous models and theories. Some of the theories and models are applicable to newer programming paradigms and some are not. In this chapter we discuss the Wiedenbeck et al. [40] study and studies of differences between object-oriented programming and procedural programming comprehension.

3.1 Characteristics of mental representation experiment done for procedural programming

Wiedenbeck et al. studied characteristics of mental representation in novices and experts. They defined mental representation as "the understander's internal model of the target program. It is built up from the study of the program text and other information that the understander has available" (p. 1) [40]

They mentioned that it is important to know what kind of information programmers will pay attention to and how that information is characteristically organized. Wiedenbeck et al. designed an empirical study to find characteristics of mental rep-

resentation with a special focus on differences between novices and experts; they designed their experiment based on previous studies of Adelson and Holt [18].

Wiedenbeck et al. proposed that an expert programmer's mental representation consists of five abstract characteristics:

"1-It is hierarchical and multi-layered;

2-It contains explicit mappings between the different layers;

3-It is founded on the recognition of basic patterns;

4-It is well connected internally;

5-It is well grounded in the program text;" (p. 2)[40]

These characteristics are features that are covering the mental representation altogether, so they are different from the Adelson's [6] and Pennington's [25] information types, which focused the content of the mental representation. In Wiedenbeck et al.'s study, these features are called abstract features. These features show why specific categories of information are collected and mostly focus on the reason for the mechanisms that programmers use when comprehending a program.

Hierarchically structured:

Wiedenbeck et al. define this feature based on Letowsky's model. A hierarchically structured mental representation is a multilayered network which consists of different elements of a program in different layers. The network has a different depth and breadth based on the structure of elements of a program [40][21]. According to this characteristic, an expert programmer will find goals and sub-goals of a program in which they are all linked together. Wiedenbeck et al. argued based on the studies of Jeffries [19] and Nanja and Cook [23] that advanced programmers read a program in the order that it will be executed and this strategy can lead to the development of a hierarchical representation of the program. Wiedenbeck et al. hypothesized that only experts will have this feature in their mental representation, however Holt, et al. [18] found that there is no significant difference in "depth" of mental representations

of novices and experts[18].

Explicit mapping:

Explicit mapping exists between layers of representation[40]. In Letowsky's model,the highest level of representation of a program is the specification which includes the main goal of the program and the lowest level is the implementation of the program which consists of data structures and functions of the program [21]. This level is also available to the programmer but the problem in comprehending the program is making the connection and mapping between the highest level and the lowest level [21]. Wiedenbeck et al. also mentioned the Pennington experiment [25], which found that when experts read a program first they will make a hypothesis about the program and will verify their hypothesis by connecting their idea to the specific part of the code; in contrast, the novices will make different hypotheses using only segments of codes or variable names and they do not verify their hypothesis using the code like the experts [25]. Based on these previous studies, Wiedenbeck et al. expected significant difference between novices and experts in this characteristic and found support as expected.

Recurring basic patterns:

Wiedenbeck et al. used Soloway et al.'s plan concept in order to define this feature [32][34]. Programmers use plans in their memory in order to use basic operations like looping in their programs. These basic plans are used in writing and comprehension of programs by programmers and based on Soloway, et al.'s experiments experts had problems comprehending the unplan-liked part of programs [32]. In the Soloway, et al.'s experiment, participants who have seen unplan-like programs recall them as plan-like programs. The hypothesis was that novices and experts would have a difference in connecting the program code with plan labels except in very simple plans and they found support for it [40].

Well connected:

Wiedenbeck et al. defined a well-connected representation as “ One where the programmer understands how parts of the program interact with one another.”[40] They used Soloway’s delocalized plans concept to discuss this feature[32]. Delocalized plans are the plans that are not obvious like a module in the body of code, and programmers need to follow the interaction between segments of code in the program [32]. Wiedenbeck et al. argues based on the previous study of Jeffries et al, that experienced programmers have more attention compared to novices, on the part of code that involves interaction with other segments [20]. So in this feature experts will try to gather the information related to interaction and connections in the comprehension process while novices will lack this feature and they found support for this characteristic as well [40] .

Well grounded:

Wiedenbeck et al. stated that good mental representations are well grounded in program code [40]. It means that when a programmer has a good mental representation of the program they will know the location of certain structures and the location where the specific actions are happening. They proposed this feature based on Jeffries’s study [19] which found that experts had a very good performance at locating information in tasks which they have seen previously and novices did not perform well; they used line by line search or random search to locate the information[19]. Therefore Wiedenbeck et al. hypothesized that experts would have well grounded representations and they found support for it.

Wiedenbeck et al. study

In Wiedenbeck et al.’s paper, participants were asked to read 135 lines of a Pascal program and answer the researchers’ questions. The questions were designed to find out about the characteristics mentioned earlier, and most of the questions were written in pairs; one question could be harder for novices to answer and they could perform better on the other one, so that the authors could find any differ-

ences reflecting the characteristics and programming knowledge [40]. In the study, 40 programmers took part in the experiment: half experts and half novices [40]. The novices were undergraduate students who had taken a Pascal programming course. None of the novices had work experience as a programmer. Expert participants were very distinct from the novice group. They were all full-time programmers, and they had prior work experience between 2-13 years [40]. The program consisted of 7 procedures in the Pascal language. The program goal was modifying student data. The program was printed in 3 pages and 135 lines of codes. No documentation was available for participants [40]. They were asked to study the program in detail. At the end of the study time, the program papers were taken, and they received the question booklet [40].

Based on the MANOVA test run on all questions to find performance difference between novices and experts (score as dependent variable and questions as independent variables), the experts' score were significantly higher. An ANOVA test was run on each question to find out if the null hypothesis was rejected , and if there were a significant differences between novices and experts.

Each pair of questions were related to one of the mentioned characteristics. The analysis showed that novices and experts had significant difference in hard questions in each pair and they mostly did not show significant difference in other questions. So Wiedenbeck et al. found support for the existence of all 5 characteristics.

3.2 Object-oriented and procedural programming differences

Object-oriented programming is quite common nowadays based on repositories and contributions in "Github" [3] and activities on "Stackoverflow" [5]. Many universities and colleges teach students an object-oriented language as a first programming lan-

guage. This paradigm of programming is built around the concept of objects that can be related to objects in the real world, so it would be more easily for new learners to understand programming. There are many differences between object-oriented and procedural programming and these differences have been studied before in a matter of the design, program comprehensions and mental models built by object-oriented programmers. Mainly the studies were designed around the difference in the size of object-oriented programs and expertise of programmers.

Pennington's model is based on procedural programming [26], but there are many fundamental differences between procedural and object-oriented programming and likely, differences in program comprehension of object-oriented programs vs procedural programs.

Burkhard, et al. studied mental representations of experts and novices using object-oriented languages [11]. They used Pennington's model of program comprehension that included the situation model and program model [25]. Their results confirmed the distinction between program model and situation model in object-oriented programmers [11]. Their results indicate that the situation model results is better than the program model, based on the correctness of the questions they answered in each part, and expertise level was related to the construction of the situation model but not to the program model. In Burkhard et al. experiment, participants developed a situation model fully at the early stages of comprehending and this was in contrast with results of Pennington in procedural languages in which program models were developed more at the early stages [11]. Burkhard, et al. mentioned that this contrast can be related to the paradigm of object-oriented programming. They also mentioned that experts performed better in developing the situation model, so that means experience and prior domain knowledge are also important in object-oriented program comprehension [11].

OOP is a more top-down approach compared to procedural programming [26]. Ex-

pert object-oriented programmers developed a domain-based abstraction in terms of objects, functions, and relationships between functions when forming programming comprehension [39].

In Corritore and Wiedenbeck et al.'s experiment [12], object-oriented programmers had better performance (based on correctness percent) in the domain model (situation model), including better understanding of relationships between program objects, at least in smaller programs, but they had a poorer understanding of particular operations and control flow compared with procedural programmers [12]. In their second experiment, object-oriented programmers did not have better performance, neither in the program model nor the domain model. However, after doing modification tasks, object-oriented programmers had equal program level knowledge when compared with procedural programmers. Corritore et al. mention that there are two reasons for the difficulties in longer programs: 1- The object-oriented programs functionality is broadened over different classes and parts compared to procedural programs so it can take longer to gain knowledge. 2- The object-oriented paradigm has a longer learning curve for novices [12].

Considering the differences mentioned between procedural and object-oriented languages we should expect a different result in each characteristic of mental representations. We will discuss our expectations for each question in the next chapter. Based on the results of the mentioned studies, novice object-oriented programmers develop the mental representation more easily at early stages. Based on the more understandable nature of these programming languages we expect that would be less distinction between an early object-oriented programmer(novice) and an object-oriented expert. Also since this type of programming language emphasizes top-down structure, even novices can have an understanding of the overall goals of the program at an early stage of comprehension.

Chapter 4

Work and Objectives

4.1 Experiment design

4.1.1 Methodology

In Wiedenbeck et al.'s [40] research the programmers studied a program consisting of 135 lines of codes in a procedural language; in our study, we asked the participants to read a Java program. The code is a Battleship game program developed in six Java classes. After studying the program, participants answered a questionnaire explicitly designed for the program to find and understand the mental representation characteristics of novices and experts. Our aim was to determine if there are any differences between the two groups on each characteristic and overall.

4.1.2 Participants

In our study, we had 34 participants: We had 16 novice and 18 expert programmers based on years of experience and 17 novices and 17 experts based on self assessment. There were 19 undergraduate and 13 graduate students. On average participants were 23.58 years old (SD=2.7 and ranging from 19 to 30), objective experts age mean was 24.4 years and Novices mean was 22.6 years. We had 29 male participants and 5

female participants . All participants received a 10$ gift card for their participation. This experiment was approved by University of New Brunswick's Research Ethics Board as file number of "REB 2019-134".

4.1.3 Material

The program chosen for our experiment is a well-known game named Battleship, developed in Java by Yuval Marcos [22], with some enhancement and editing in some parts. The program is fully implemented with 6 different classes; each class is no longer than 500 lines of code: `Battleship, Ship, Randomizer, Player, Location, Grid` classes. The `Battleship` class is the main class of the program which sets up all the other classes. Participants could understand a lot of the program structure and they could have an overall understanding of the program by studying the code of this class. The `Ship` class contains the details of the ship as an object in the program such as location and type of the ship in the matter of size. The `Randomizer` class is a simple randomizer performing the opponents' moves in the game. The `Player` class defined the player object and attributes like the number of ships and different methods that players can use in the game. The `Location` class contains the attributes of every specific location on the grid, such as whether or not the location contains a ship. The `Grid` class defines the game board and it contains every function related to the specific location on the board. Participants get a description of the game before reading the code because some of the participants may not have previous knowledge of battleship, and we wanted everyone to start with an understanding of the game. The program should be easy to understand both for novices and experts. We piloted the test with three participants; two experts and one novice. One of the experts had access to the code while answering the questions, in order to determine if questions were fully answerable when an expert has access to the code and unlimited time. The other two participants answered the questions

with promising results which gave us enough confidence to run the experiment.

The questionnaire was modelled on the one from the Wiedenbeck et al. study. The questions involved similar concepts as the procedural study, but only question 11 was same as the Wiedenbeck et al. study. One set of questions were designed for experts, based on the superior characteristics of their mental representations, and the other set were designed to be easy for both groups. Based on differences between procedural languages and object-oriented languages which we discussed in chapter 3, we changed questions to be suitable for our experiment (table 4.1). We considered the attributes of object-oriented languages, like the class (Object) concept and inheritance in the first pair of questions and considered encapsulation, polymorphism and abstraction in other parts.

Demographic questionnaire: Finally the participants answered some questions about their programming backgrounds and demographics questions to document the origin of participants and assist in data interpretation. The questions are available in Appendix C.

Characteristic	Question Number	Question	Answer
Hierarchically structured	1	Which classes don't need the Ship class to compile?	1-Location Class (1) 2-Randomize Class (2) (2 points)
Hierarchically structured	2	Name all the classes in the program.	1-Battleship (1) 2-Grid (2) 3-Location (3) 4-Player (4) 5-Randomize (5) 6-Ship (6) (6 points)
Explicit Mapping	3	Describe the Location and the Battleship classes. What are the goals of each class?	Battleship: Driver for the game (1). Implements game logic (2), setup and IO (3). Location: Location objects represent each position in a grid (4). Keeps track of hits and misses on the board (5). (5 points)
Explicit Mapping	4	Does the player get an extra turn when guessing the right location?	No (1) (1 point)
Recurring Pattern	5-A	Briefly describe the purpose of the following code:	This code makes sure that the ship is not too long to fit in the grid horizontally (1) (1 point)
Recurring Pattern	5-B	Briefly describe the purpose of the following code	This is how the "computer" player randomly adds and places its ships on the grid (1) (1 point)
Recurring Pattern	6-A	Describe what the following method is doing:	It converts alphabet into integer. For example, A for row 0 and B for row 1 (1). (1 point)

Characteristic	Question Number	Question	Answer
Recurring Pattern	6-B	Describe what the following code is doing	It is used to setup locations in a grid. All location are initialized as unguessed and 'hasShip' = false. (1) (1 point)
Well Connected	7	How is a "user hit" determined in the Battleship class? Which methods in which class are called to record a hit.	The isHit() function (1) from the Grid class is called which itself (2) calls the isHit() function form the location class. (these are being called on the player's oppGrid) (3) (2 points in analysis 1)
Well Connected	8	Write the names of the Grid instances in the Player class.	1-Public Grid PlayerGrid (1) 2-Public Grid oppGrid (2) (2 points)
Well Grounded	9	Match the methods to the appropriate class.	1-Ship (1) 2-Grid (2) 3-Player (3) 4-Battleship (4) 5-Location (5) (5 points)
Well Grounded	10	Match each instance variable to the class, where it is declared.	1-Location (1) 2-Ship (2) 3-Battleship (3) 4-Location (4) 5-Grid (5) 6-Ship (6) 7-Location (7) 8-Location (8) 9-Ship (9) (9 points)
Well Grounded	11	Where is the exact location of the toString method in the Ship class?	At the end of the class (1) (1 point)

Table 4.1: Questions and best answers from different participants' data. Numbers in the brackets after each part of answers shows the scoring rubric of the second analysis and how the answers are divided, and the points at the end of each answer are used for the scoring rubric of the first analysis

1-Hierarchical

The first pair of questions pertained to the questions about the hierarchical characteristics of mental representations. They were designed in a way that one of the questions would be easy to answer for both groups and we expected that experts would answer the other question better. The program contained six Java classes and

Figure 4.1: Battleship program UML

we tried to examine the hierarchical understanding of the participants in question 1 by asking them about the classes that are not needed to compile the `ship` class

(Figure 4.1). The participants needed a well developed hierarchical understanding of the different layers of the program's class implementation to answer this question. Question 2 asked the participants to name all the classes in the program without specifying any order or details. We used the participants' scores in this question to determine whether the participants used memorization to answer question 1 or their hierarchical mental model. If we do not find a significant difference between experts and novices in this question, we will reject the idea that they used memorization.

2-Well developed mapping of the code to goal

The second pair of questions were designed in order to find the ability of the participants to build a well-developed mapping of the code to the goal of the program. Each class of the program does a specific task in order to reach the final goal of the application. In question 3, participants were asked to describe the goals of two classes of the program (The Battleship and The `Location` class). We expect both novices and experts to perform well in this question and have a good understanding of the goal of each class. We needed to determine the participant's overall understanding of the goal of the program, but since we described the game before they started the experiment and they had an understanding of the overall goal, we asked more detailed questions about the classes.

Since the program is about a well-known game, participants had previous knowledge of the rules of the game, but some rules can be different in a turn-based game and they are usually defined by the players at the beginning of the game; therefore, we asked about the rule of a bonus turn when a player guess a right location in the game.

3-Recurring patterns

In recurring patterns, we tried to get to the understanding of participants in the program using simple plan knowledge. Questions 5 and 6 each had 2 parts and contained a piece of code from the program. The codes used different plan knowledge.

In question 5 we asked about the purpose of code that contained simple plans like assigning variables and if-statements and a while loop. We did not want the participants to describe line by line the function of code, but we asked the brief purpose of the code, where they would use their plan knowledge, how they understood the code, and we expected both groups to perform well in this question.

In question 6 we asked about how the code worked in two selected code segments from the program. The segments contained more complex patterns like the understanding of ASCII codes and how they can be converted to integers and also nested loops requiring participants to use recurring patterns and previous programming knowledge they had in order to answer the question. As in the previous question, we did not accept line by line description of code and definition of syntaxes as answers. We expected that this question would be more difficult for novices and that experts would perform better on this question.

4-Well-connected representation

These questions were designed to find the understanding of connections between different parts of the program in the participant's mental representation. In object-oriented languages, the data pass through different classes by using instances of different classes or simply by making an object in certain classes. We examined the well-connected representation in questions 7 and 8. In question 7, participants were asked to find how the program determines a hit and how the data-passing happens in the specific functions that are involved in this process. They were also asked to name the classes involved in this process. We expected experts to perform better in this question and have a better representation of connections in the program.

Question 8 was designed to examine the well-connected representation of participants by asking them simply write the name of the instance variable defined in a specific class. We asked this question to assess participants' understanding of the connections between classes, and we anticipated that both groups would perform

well in this question.

5-Well-grounded representation

In question 9, 10 and 11 we asked the participants about their understanding of the information in the program text to determine how well-grounded their representation is. In question 9 we designed a multiple-choice question about selecting methods for the classes in which they were defined. The answers to this question would indicate how well-grounded the participants' representation was in order to remember the location in which the method was defined, and a good performance shows that participants have an overall view of the methods and class in their mental representation.

Question 10 asked the participants to connect the instance variables to the appropriate class; we expected that this question would be easier for experts since the last question uses methods that can be self-explanatory but instance variables need to be well-grounded in order to be remembered.

Question 11 asked the participants to write the exact location of a specific method in a class, and the location of that method is not based on specific knowledge or rule. This will reveal how well the participants understood the program text and whether they remembered it as a structured text.

4.1.4 Procedure

Participants used a Windows operated computer in a lab for the experiment. They were assigned their study group number and their identification number; then they read the description of the game and had 30 minutes to study the program. They had access to all classes and code from beginning to the end of the study time but they could not run the code. After 30 minutes, they no longer had access to the code and the questions showed up. They did see the questions in a specific order, which is designed to avoid revealing any answers to other questions. Participants could not

go back to previous questions, but they had unlimited time to answer them.

In the end, they answered a background questionnaire (Appendix C). We used an online service named "Formsite" [4] to do the experiment and collecting answers and the program code was shown to participants in Visual Studio Code [1] editor.

4.1.5 Scoring

The participants' score in comprehension questions were independently evaluated by two judges. The judges scored all the questions using the scoring criteria (Table 4.1). The question with descriptive answers needed judgment and we used the average score of the judges for the first analysis presented in chapter 5. The level of agreement for the questions was 97 percent. Since there can be lots of wrong answer in these questions, Wiedenbeck et al. discuss two possible sources: 1) They can be the results of wrong information in their mental representation, caused by misunderstanding; 2) The wrong answers could be the result of guessing when participants do not have the information in their mental representation [40]. Because there is no way to find the source of the wrong answer we decided to use the same conservative approach that Wiedenbeck et al. used by deducting 1/2 for each wrong answer in the mentioned questions in the first round of analysis. We ignored the guessing effects in the second round of analysis, which is based on mixed modeling and each part of a question that had a specific point counted as an individual component so the scores for each components would be 0s and 1s to make it possible to use them as inputs for mixed effect logistic models. The scoring rubric is provided in table 4.1.

Chapter 5

Experimental Results

The evaluation of experimental results was done by using two methods. At first, we did the same type of tests and analysis as Wiedenbeck et al.'s study, where expertise was defined by years of experience, then in the second round of analysis, we used a more modern approach using "Mixed model logistic modeling" [13] with both definition of expertise (objective and subjective). We designed the second round of analysis based on the Logistic model since we had limited data with high noise and challenging to decide outliers. Our data were not completely numerical or categorical, so we changed our rubric to a more suitable type of data, in our case, binary data, which can be fit in a logistic model.

5.1 Demographic Analysis

The distribution of participants based on the study group and the year they started their university studies/program is provided in Figure 5.1. The expert group has an average experience of more than two years, and the novice group average is less than half a year. Since we did this experiment in a university lab and we anticipated that most of our participants would be students, we asked about the participants' co-op experience and found that 15 participants had coop experience in total. We used

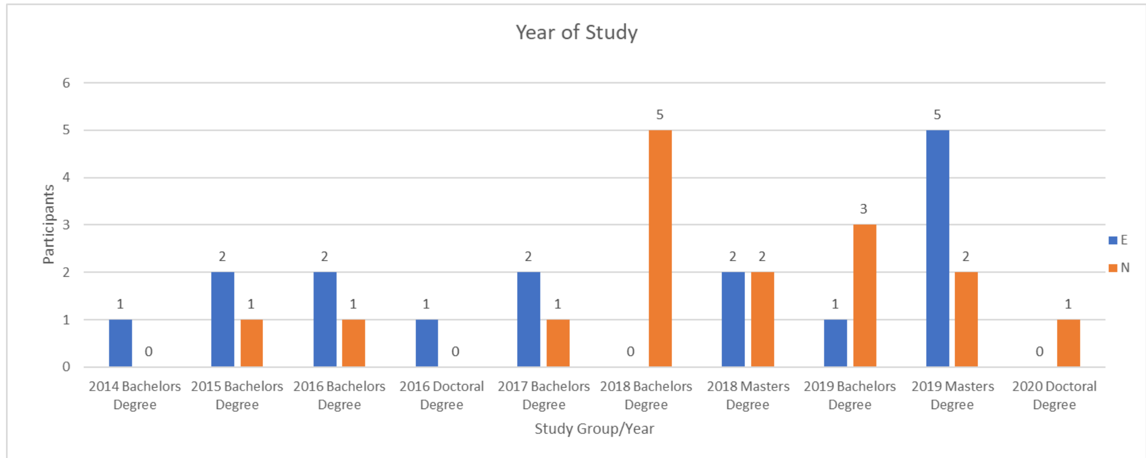

Figure 5.1: Beginning year of study

the co-op experience along with work experience in expertise definition.

5.2 Expertise discussion

Our demographic questionnaire asked participants what they think about their expertise in programming (subjective). Some participants considered themselves other than we classified them (objective). We had 18 objective experts based on experience, of which 7 assessed themselves as novices; we had 16 objective novices, of which 6 assessed themselves as experts. We analyzed both groups using the chi-square test. The results showed that there is a significant relationship between the two groups chi-squared (1, N = 34) = 70.836, p-value $< .001$. Fisher's Exact Test for Count Data showed an odds ratio of 2.616867, so for every participant who did not have the same self assessment as the objective assessment, there were 2.6 participants whose self-assessment agreed with the objective one.

The reason for the difference between objective and subjective group could be because the participants were all students, and most of their experience was coop or academic in nature, and they may have been unsure how to rate their expertise. A better approach for the self assessment could be to ask about the participant's ex-

pertise in comparison with their colleagues or classmates [15]. Since we had a broad range of students with working experience in different companies or different years of entrance, we could not ask this question.

5.3 Analysis 1

We used SPSS for our analysis and statistical tests. In analysis one, we used similar tests and analyses as the Wiedenbeck et al. study. They used MANOVA (multivariate test) to determine if two groups have significant differences, and since they had, they used ANOVA (univariate test) to find the difference between groups in each question. We found that the two groups did not show significant difference considering all questions and the objective expertise definition, f-value(11,22)=1.54, p-value=0.187; Wilk's Lambda=0.565. So, we took another approach in analysis 2.

5.4 Analysis 2

In the second round of analysis, we used another approach. We used repeated measure design and mixed effect logistic models to analyze our data. Mixed effects logistic modeling is used to model binary outcome variables, where the log odds of the results are modeled as a linear combination of the predictor variables when data are clustered [2]. In order to be able to use the data in the logistic mixed modeling, we changed our scoring. We divided each question into several components based on the grading schema so each component would have 1 point and scores would be 0 for a wrong answer and 1 for a correct answer (Table 4.1). Therefore we did not deduct any points from the wrong answer as mentioned in chapter 4.1.5. The independent variables in this model were "expertise" and "questions" and the dependent variable was "score" achieved in each component. Since we had binary scores, we followed the recommendation of Dixon [13]: the accuracy of data was analyzed by generalized

logistic mixed modeling. By considering repeated-measure design, our data were analyzed using a mixed linear model where questions were considered random effects in both kinds of expertise, but the models did not converge, probably due to the small size of the sample [7]. Therefore we continued our analysis with fixed effect models (Appendix D). We used R language and RStudio for this analysis (Appendix D) [29]. As shown in table 5.1 the main effects are significant(p-value < 0.05); it shows that objective expertise and questions had a significant effect in our model, but since in our study the interaction between these two was essential and the interaction is not significant we can not use this model for further analysis. In table 5.2 that shows subjective expertise, the questions have a significant effect, and the interaction is also significant so we can analyze the interaction further. We used the chi-square test using test-interaction to find questions with significant differences between the two groups. Questions 4, 6, and 8 showed significant differences between novices and experts based on the objective expertise (table 5.3).

	Chisq	Df	p
Objective Expertise	5.5211	1	0.01879
Questions	197.6698	10	$<$ 2e-16
Objective expertise $\times$ Questions	8.7457	10	0.55639

Table 5.1: Summary of Results for Linear Mixed Models in Objective Expertise Grouping

	Chisq	Df	P
Subjective Expertise	2.7048	1	0.10004
Questions	192.4952	10	< 2e-16
Subjective expertise × Questions	18.6056	10	0.04557

Table 5.2: Summary of Results for Linear Mixed Models in Subjective Expertise Grouping

Question	**Value**	**Df**	**Chisq**	**Pr(>Chisq)**
1	0.67	1	1.3393	0.247
2	0.40	1	0.2765	0.599
3	0.65	1	1.8576	0.1729
4	0.04	1	6.118	0.01338
5	0.80	1	2.3587	0.12459
6	0.87	1	6.4015	0.0114
7	0.61	1	0.8504	0.35643
8	0.77	1	4.3744	0.03643
9	0.61	1	1.2966	0.25484
10	0.57	1	0.6059	0.43634
11	0.63	1	0.4663	0.4947

Table 5.3: Summary of Test of Expertise by Question Interaction in Fixed Effect Models in Subjective Grouping; Note, highlights showing questions with significant difference

We analyzed questions based on each group's means in each question (table 5.4). In question 4, we expected a significant difference between groups, which they presented, but novices performed significantly better than experts. In question 6, we expected a significant difference between groups, which they presented, and experts performed better than novices. In question 8, we did not expect a significant difference between groups but based on our analysis, they had. Experts scored significantly better than novices in this question.

Question	Group	N	Mean	SD
1	E	34	0.7353	0.4478
1	N	34	0.5882	0.4996
2	E	102	0.9412	0.2365
2	N	102	0.9510	0.2170
3	E	85	0.8118	0.3932
3	N	85	0.6941	0.4635
4	E	17	0.5294	0.5145
4	N	17	0.9412	0.2425
5	E	34	0.9412	0.2388
5	N	34	0.7941	0.4104
6	E	34	0.9118	0.2879
6	N	34	0.6176	0.4933
7	E	51	0.5490	0.5025
7	N	51	0.4510	0.5025
8	E	34	0.6176	0.4933
8	N	34	0.3529	0.4851
9	E	85	0.6235	0.4874
9	N	85	0.5176	0.5027
10	E	153	0.3595	0.4814
10	N	153	0.3137	0.4655
11	E	17	0.6471	0.4926
11	N	17	0.5294	0.5145

Table 5.4: Summary of Mean and Standard Deviation for each Group in each Question. Note, Group = Novice and Expert based on subjective groups; N = number of answers based on components of each question; SD = Standard Deviation.Highlights showing questions with significant difference

5.5 Analysis of results

The results of our study did not support all the characteristics in Wiedenbeck et al.'s study. Our study and experiment design were conducted by following Wiedenbeck et al.'s study, but as we discussed in section 3.2, the mental representations in object-oriented programmers can be different in several aspects. Although we found a lack of evidence for some characteristics this does not mean they do not exist. Our MANOVA test in analysis one did not show any significant difference, so we decided to continue our analysis in the new approach in analysis two. The main effects of analysis 2 showed that our two groups have a significant difference considering subjective expertise. We have some results that can support the existence of some characteristics, and we will discuss the explanations of the differences with the previous study and the interpretation that we can make about the available data for each question.

A program is a text in which each statement represents a meaningful action, and when someone reads the program with knowledge of the language, they can run the function of each piece in their mind. The ability to run the code is the part that makes the difference between a regular text and a program. These sets of statements in a program have some hierarchical design based on the multi-layered and multi-level goals of actions in a program. The hierarchical and multi-layered design of programs is not a theory that all researchers agree on, but the previous study of characteristics in a procedural language found support for this characteristic [40]. In our study, we expected a significant difference between novices and experts to support the existence of procedural languages' characteristics of representations in object-oriented languages, which our results did not confirm. In question 1 we asked a question (Appendix C) about the classes which do not need the `Ship` class to compile. The participants needed to understand the goals and connections of each class to answer these questions. Even though the two groups did not show a significant difference,

experts performed better than novices. In question 2, experts and novices performed very similarly (Table 5.4). This question asks for all classes' names, which does not require any hierarchical understanding of the program. The result in question 2 is similar to the previous study of procedural languages. The difference may have been more significant in question 1 if we had more participants or a more distinctive group of participants (if the experts' group had more experience or would have been professionals in this field or if all novices were second-year students). We can also discuss the possibility of an easier understanding of the hierarchical design of this particular program or, even more broadly, easier construction of mental models in object-oriented languages, which leads to less distinction between two groups. By considering the small differences between the two groups for both questions, we can state that both groups can recall the program's classes and layers well and understand each layer's level of actions and goals. To have more confidence in the results, more experiments using different programs in OO languages and different sizes and the number of classes would be helpful.

Our results did not support the well-developed mapping of the code to goal characteristic, as the pair of questions related to this characteristic did not show any significant difference in the way we anticipated. In question 3, we asked about the goal in two of the classes appropriately named. Based on the results, both groups performed well, although experts group mean was higher. The analysis tests did not show any significant difference (Table 5.4). We cannot reject these characteristics based on our results, nor can we support them. Nevertheless, considering the scores, both groups understood each class's goals well. Since the results diverge from the previous study on Procedural programming, we can hypothesize that the object-oriented classes result in a better understanding of the code's goal and mapping the program's sub-goal to the code. Since we described the game to the participants, we could not ask them about the program's overall goal, so we can not discuss the

understanding of the overall goal and the mapping of it to the classes. In question 4, we asked a question about a rule of the game that was not discussed in the description and can be different in different versions of the game, so the participants could not answer based on previous knowledge. This question required understanding the goal of a specific part of the code and also required the participants to map this goal to code. We expected that experts would perform better in this question since this question required the participants to pay attention to the very detailed functionality of the code, and it was a sub-goal of the program. There was a significant difference as we expected between the two groups, but interestingly novices performed better than experts in this task. This result might have occurred because of the unsuitable question design, or because the rule of the game was not important to experts, or because they were more focused on the overall goal of the functions or spent their time on the code, not the concept of the game implemented in the code. Either way, the experts did not perform well in this question, performing significantly lower than novices, which could be further analyzed in future studies. Also, we think that previous game experience could result in less attention in parts where the game's logic was declared since experts could recognize these parts, which do not need specific attention in the implementation of the code, so they just skipped them. This result could support the more apparent sub-goal implementation of code in object-oriented languages, which makes the novices able to map the program's code and goal very well, and when they get more experienced, they prefer to focus on other parts. There is a need for more experiments to confidently support this characteristic in an object-oriented language that targets more distinct participants and notably different programs with a different level of clarity in implementing the code.

The result for the third pair of questions(5 and 6) supported the existence of the recurring pattern characteristics. The questions in this pair had two parts, and each part was asking about a snippet of code related to the program. In the first question,

participants just mentioned the purpose of the code, and in the second question of this pair, they also described the snippet of code. In the first question, both groups performed well in finding the purpose of the code, and they could use their previous programming knowledge. The participants understood the simple plans used to implement specific functions and could write about it easily. In the second question, we used more complex patterns, which needed a better understanding of the program and more details. Experts performed significantly better and supported the idea that experienced programmers could recognize recurring patterns better. Novices in some parts mostly were translating line by line of code to English, which was not acceptable since it showed a lack of understanding of the meaning of the code.

The results of our study show support for well-connected characteristics(questions 7 and 8) in experts but not as we expected. We asked two questions in this pair, and like other pairs, we designed an easier part which we expected both groups to answer correctly and a more challenging question for which experts would have a better score. However, both of our questions seemed hard for participants. Well-connectedness in representation of programs is related to understanding data connections in different program parts and the code's data flow. We asked about how different classes work together and connect to record a "user hit" and we expected the answers to contain the different methods and classes used in order to record this action. The experts and novices did not have significant differences, but experts performed better. We think that the question was too hard for both groups. In question 8, we asked about the names of a class's instances in another class, we expected both groups to perform the same in this question, but they had a significant difference in this question. Experts were better than novices. Our results in this part support the existence of well-connected representations in object-oriented programmers but not in the way we expected. In this question, we asked about the name of a class instance, and we think that novices did not study the code in detail and did not memorize the names

since they thought in a real situation they can find the instances names just by searching in the IDE. These reasons could be the cause of the unexpected significant difference. Like other questions, the type of program and code can cause different results, and using different programs with different lengths can lead to better support of this idea.

This study did not show support for the well-grounded(questions 9, 10 and 11) characteristic of experts. Novices and experts showed some differences in the three questions of these characteristics, but they were not statistically significant. When asked about linking the methods to the classes where they were defined, experts were better than novices in this question, but both groups scored low. In question 10, we asked about instance variables linked to the classes where they were defined, and like the previous question, both groups performed poorly. In question 11, we asked about the location of a specific method in a class, and in this question, experts were better, but this difference was not significant. We used a radio button matrix type question for questions 9 and 10 in which participants could select the answer by just clicking on the radio button based on the column and row matching names. We think that this poor performance could result from guessing the answers of these questions. Experts performed better in all questions in this part, which means that they had a better well-grounded understanding of program text, but the difference with novices was not statistically significant. We can consider object-oriented languages' attributes, which causes both groups to not focus on the methods' locations in the program's text. They mostly focus on the code's functionality and goal and the data flow of different parts of the code. Wiedenbeck et al. state that [40] lack of this characteristic causes difficulty in debugging and complex maintenance of code. However, we did not see this characteristic in our experts; it could mean that in object-oriented languages and modern IDE with the capability of search and finding different objects and methods and tracing, this characteristic is not as important. We need more studies with

programs in different object-oriented languages and program lengths to have enough confidence in this claim.

Chapter 6

Conclusion

6.1 Conclusion

In this thesis, we designed and carried out an experiment consisting of different types of questions related to object-oriented program comprehension based on an earlier study of procedural program comprehension done in the 1990s. We considered two types of expertise definition, the objective definition which we defined based on the years of experience and the subjective definition which is based on the self-assessment of participants. We used two approaches to analyse our data, the first approach was based on the older methodology of using MANOVA and ANOVA tests that Wiedenbeck et al. used, and the other approach was using mixed logistic models analysis. Our experiment showed support for some of the characteristics and did not support the existence of other characteristics. The lack of support could be the result of difference in programming paradigms and the limitations we had in the experiment. We found support for the differences between characteristics of mental representations of novices and experts considering expertise based on self-assessment but we did not find support for the differences based on objective expertise. The differences suggests that expert programmers have characteristics that the novices lack. The

existence of some characteristics would provide evidence on why experts perform better in some tasks and the novice programmers do not perform well. For example, in questions 5 and 6 which are related to **recurring patterns**, we found support for the existence of this characteristic in experts. The statistical tests show both novices and experts performed well in question 5 which was the easier question and only experts performed well in the harder question. We can use the evidence and support for this characteristic to understand object-oriented languages comprehension and to focus on this characteristic when teaching novices, or even use it to distinguish between programmers selected to work on a specific task. Our study shows a lack of support for **hierarchical**, **well-developed mapping** and **well-grounded** characteristics, where we did not agree with the earlier study. The earlier study showed significant difference for all of the characteristics and experts performed significantly better in harder questions. But in our study the experts were not always better as we discussed in the previous chapter. In the **recurring patterns** characteristic our results agrees with the earlier study since there is a significant difference in the harder question between the two groups. In the **well-connected** representation our results are different, both groups performed poorly in the harder question but they had a significant difference in the easier question. The disagreement which we have between our study and the previous study could be the result of two different factors: 1-The differences between the two language paradigms 2-The limitations which caused the lack of support in our results.

This lack of support could be the result of unsuitable questions, like the questions which were too easy or too hard for both groups. These questions did not allow us to discriminate between two groups. Also, code, or the coding style could have been unusual for programmers in places. It is also possible that the object-oriented developers may not have all the characteristics. As we mentioned in section 3.2 about the object-oriented and procedural programming languages differences, in Corritore

et al. object-oriented programmers had difficulties in longer programs because of two reasons [12]: 1-The object-oriented programs functionality is broadened over different classes and parts in compare to procedural programs so it can take longer to gain knowledge. 2-The object-oriented paradigm has a longer learning curve for novices. Our result is consistent with these findings. In **hierarchical structure**, the experts and novices did not show any difference and it can be related to both reasons, first, our program was long and experts could not show significantly better performance due to reason 1 and also since our groups were not perfectly distinct since the learning curve is longer (not enough experts) we could not find the same result as the earlier study. In **well-developed mapping** and **well-grounded** also we have the same effects of long OO programs. But in **recurring patterns** since we used snippets of codes to ask questions, which means smaller programs the mentioned reasons did not affect our study and the experts performed significantly better than the novices and we could find support for this characteristic. We had a significant difference in **well-connected** but not in the way we expected, this unexpected result could be the result of a hard question which we discussed earlier. Although the experts still performed better in these questions with a significant difference in the easier question we can state that experts have this characteristic despite the difficulties in long programs. In question 4 (**well-developed mapping**) we found significant difference between the two groups but in further analysis we found that novices performed better than experts in this question. This difference could be the result of an unsuitable question or it could be difference in programming language paradigm. Since we do not have enough support we suggest more study of this characteristic.

The characteristics, question type and scoring for analysis 1 in this study were taken from previous research done by Wiedenbeck et al., and we did not find significant difference between the two groups in MANOVA, therefore we could not continue our

analysis using uni-variate tests like ANOVA. We used a more recent and suitable analysis which helped us to find better support in our experiment. In conclusion, based on our results some of the characteristics were useful and meaningful for object-oriented program comprehension.

6.2 Limitations

Our study had several limitations, in common with other studies in this field. As we mentioned in the last chapter, one of the important factors which affected our experiment is the lack of enough distinction between participants. We advertised our experiment at the University of New Brunswick Fredericton campus, mainly to Computer Science students, and some software companies, however all of the participants were students. We only grouped the participants based on the limited work experience which they had, and some of them only had co-op experience. We think that more distinct participants in groups like in Wiedenbeck et al.'s study, would result in more significant results; professional developers and second-year students could be better experts and novices for our experiment. Also at the time of running this experiment, the world experienced a pandemic (Covid-19, 2020) that caused the closure of campus and cancellation of in-person classes. Since the closure happened in the middle of running the experiment we could not change our procedure to solve the limitation and we analyzed the results with available data. Certainly more participants could lead to more significant results by increasing statistical power and allowing the random effect models to converge.

We encountered the same limitation as Wiedenbeck et al. in our study. She names the lack of "naturalness" of the task. We asked the participants to study the code to answer some questions about it. The participants expected more details about how to study and what should they consider in their study of code. Some of them asked

whether they should memorize or write down code or take notes and we could not give an appropriate answer to them because it could influence the results. Experts may have expected to write code, modify or debug the code as usually happens in development tasks but novices who had not experienced would only memorize the code. Also, previous experience of the participants could affect the way they study the code, for example someone with experience using large code or enterprise software may only concentrate on the goals of each class. We think that asking participants to do modification on code or debugging would change the results because participants would study some parts in details and could leave out some other parts.

6.3 Future Work

We had several limitations in our study which can be addressed by using different methods. By increasing the number of participants we may get lower standard error and more confidence in our results. Also by choosing more professional participants with more years of experience or using other qualification in programming, like the participants of programming competitions, we can come to a new grouping condition which can be named as ”true expert”. We can have students who have just finished a programming course but do not have any programming experience other than academic experience as ”true novices”. By having these two groups we can perform a better experiment with less noise and outliers which would result in less variance. Also by using different programs with different size of codes and structure (like more or less number of classes) we may have better confidence in our results. By using the better conditions for experts and novices, designing an experiment in which only true experts perform well, we could create a standard questionnaire and also a clustering model which could be used to identify novice and expert programmers by giving

them a score. This score could be used in interviews for programming jobs.

Bibliography

[1] *code.visualstudio.com*, 2019.

[2] *https://stats.idre.ucla.edu/r/dae/mixed-effects-logistic-regression/*, 2019.

[3] *octoverse.github.com/*, 2019.

[4] *www.formsite.com*, 2019.

[5] *insights.stackoverflow.com/survey/2020*, 2020.

[6] Beth Adelson, *When novices surpass experts: The difficulty of a task may increase with expertise.*, Journal of Experimental Psychology: Learning, Memory, and Cognition **10** (1984), no. 3, 483.

[7] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker, *Fitting linear mixed-effects models using lme4*, Journal of Statistical Software **67** (2015), no. 1, 1–48.

[8] Leah Bidlake, Eric Aubanel, and Daniel Voyer, *Systematic literature review of empirical studies on mental representations of programs*, Journal of Systems and Software (2020), 110565.

[9] D.A Boehm-Davis, *Software comprehension*, (1988), 107–121.

[10] Ruven Brooks, *Towards a theory of the comprehension of computer programs*, International journal of man-machine studies **18** (1983), no. 6, 543–554.

[11] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck, *Mental representations constructed by experts and novices in object-oriented program comprehension*, Human-Computer Interaction INTERACT'97, Springer, 1997, pp. 339–346.

[12] Cynthia L Corritore and Susan Wiedenbeck, *Mental representations of expert procedural and object-oriented programmers in a software maintenance task*, International Journal of Human-Computer Studies **50** (1999), no. 1, 61–83.

[13] Peter Dixon, *Models of accuracy in repeated-measures designs*, Journal of Memory and Language **59** (2008), no. 4, 447–456.

[14] K Anders Ericsson and Jacqui Smith, *Toward a general theory of expertise: Prospects and limits*, Cambridge University Press, 1991.

[15] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg, *Measuring programming experience*, 2012 20th IEEE International Conference on Program Comprehension (ICPC), IEEE, 2012, pp. 73–82.

[16] Richard K Fjeldstad, *Application program maintenance study*, Report to Our Respondents, Proceedings GUIDE **48** (1983).

[17] Judith Good, *Programming paradigms, information types and graphical representations: Empirical investigations of novice program comprehension.*, Ph.D. thesis, 07 1999.

[18] Boehm-Davis D. A. Holt, R. W. and A. C Shultz, *Mental representations of student and professional programmers*, Empirical Studies of Programmers: Second Workshop, Ablex, 1987, pp. 33–46.

[19] RA Jeffries, *Comparison of debugging behavior of novice and expert programmers*, AERA Annual Meeting, 1982, p. 216.

[20] Robin Jeffries, Althea A Turner, Peter G Polson, and Michael E Atwood, *The processes involved in designing software*, Cognitive skills and their acquisition **255** (1981), 283.

[21] S Letovsky, *Cognitive processes in program comprehension: First workshop. e. soloway and s. iyengar eds*, 1986.

[22] Yuval Marcos, *Battleship*, 2015.

[23] Murthi Nanja and Curtis Cook, *An analysis of the on-line debugging process*, (1987), 172–184.

[24] Nancy J Nersessian, *How do scientists think? capturing the dynamics of conceptual change in science*, Cognitive models of science **15** (1992), 3–44.

[25] Nancy Pennington, *Comprehension strategies in programming*, Empirical studies of programmers: second workshop, Ablex Publishing Corp., 1987, pp. 100–113.

[26] Nancy Pennington, *Stimulus structures and mental representations in expert comprehension of computer programs*, Cognitive psychology **19** (1987), no. 3, 295–341.

[27] Nancy Pennington and Beatrice Grabowski, *The tasks of programming*, Psychology of programming, Elsevier, 1990, pp. 45–62.

[28] DC Pinto, Stanley Letovsky, Elliot Soloway, S Iyengar, et al., *Mental models and software maintenance*, Empirical Studies of Programmers: First Workshop, 1986.

[29] R Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2020.

[30] Spencer Rugaber, *Program comprehension*, Encyclopedia of Computer Science and Technology **35** (1995), no. 20, 341–368.

[31] Ben Shneiderman and Richard Mayer, *Syntactic/semantic interactions in programmer behavior: A model and experimental results*, International Journal of Computer & Information Sciences **8** (1979), no. 3, 219–238.

[32] Ehrlich Kate Bonar Greenspan Soloway, Elliot, *What do novices know about programming?*, Directions in Human/Computer Interaction (1982), 27–54.

[33] Elliot Soloway, Beth Adelson, and Kate Ehrlich, *Knowledge and processes in the comprehension of computer programs*, The nature of expertise (1988), 129–152.

[34] Elliot Soloway and Kate Ehrlich, *Empirical studies of programming knowledge*, IEEE Transactions on software engineering (1984), no. 5, 595–609.

[35] Elliot Soloway and Sitharama Iyengar, *Papers presented at the first workshop on empirical studies of programmers on empirical studies of programmers*, Ablex Publishing Corp., 1986.

[36] Teun A Van Dijk and Walter Kintsch, *Toward a model of text comprehension and production*, Psychological review **85** (1978), no. 5, 363–394.

[37] Teun Adrianus Van Dijk, Walter Kintsch, et al., *Strategies of discourse comprehension*, (1983).

[38] A Von Mayrhauser and AM Vans, *Program understanding–a survey (tech. rep. cs-94-120)*, Department of Computer Science, Colorado State University (1994).

[39] Anneliese von Mayrhauser and A Marie Vans, *Comprehension processes during large scale maintenance*, Proceedings of 16th International Conference on Software Engineering, IEEE, 1994, pp. 39–48.

[40] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz, *Characteristics of the mental representations of novice and expert programmers: an empirical study*, International Journal of Man-Machine Studies **39** (1993), no. 5, 793–812.

Appendix A

Program description

The following description was shown to participants before beginning of experiment:

Please read the following description of the battleship game carefully! The object of Battleship is to try and sink all of the other player's ship before they sink all of your ships. All of the other player's ships are somewhere on his/her board. You try and hit them by calling out the coordinates of one of the squares on the board. The other player also tries to hit your ships by calling out coordinates. Neither you nor the other player can see the other's board, so you must try to guess where they are. Each board in the physical game has two grids: the lower (horizontal) section for the player's ships and the upper part (vertical during play) for recording the player's guesses.

Appendix B

Code

Battleship class

```
import java.util.Scanner;
public class Battleship
{
 public static Scanner reader = new Scanner(System.in);
 public static void main(String[] args)
 {
 System.out.println("JAVA BATTLESHIP");
 System.out.println("\nPlayer SETUP:");
 Player userPlayer = new Player();
 setup(userPlayer);
 System.out.println("Computer SETUP...DONE...PRESS ENTER TO CONTINUE...
    ");
 reader.nextLine();
 reader.nextLine();
 Player computer = new Player();
 setupComputer(computer);
 System.out.println("\nCOMPUTER GRID (FOR DEBUG)...");
 computer.playerGrid.printShips();
 String result = "";
 while(true)
```

```
20   {
21     System.out.println(result);
22     System.out.println("\nUSER MAKE GUESS:");
23     result = askForGuess(userPlayer, computer);
24     if (userPlayer.playerGrid.hasLost())
25     {
26      System.out.println("COMP HIT!...USER LOSES");
27      break;
28     }
29     else if (computer.playerGrid.hasLost())
30     {
31      System.out.println("HIT!...COMPUTER LOSES");
32      break;
33     }
34      System.out.println("\nCOMPUTER IS MAKING GUESS...");
35      compMakeGuess(computer, userPlayer);
36     }
37  }
38  private static void compMakeGuess(Player comp, Player user)
39  {
40     Randomizer rand = new Randomizer();
41     int row = rand.nextInt(0, 9);
42     int col = rand.nextInt(0, 9);
43     while (comp.oppGrid.alreadyGuessed(row, col))
44      {
45       row = rand.nextInt(0, 9);
46       col = rand.nextInt(0, 9);
47      }
48      if (user.playerGrid.hasShip(row, col))
49     {
50         comp.oppGrid.markHit(row, col);
51         user.playerGrid.markHit(row, col);
52         System.out.println("COMP HIT AT "
```

```
53          + convertIntToLetter(row) + convertCompColToRegular(col));
54      }
55      else
56      {
57       comp.oppGrid.markMiss(row, col);
58       user.playerGrid.markMiss(row, col);
59       System.out.println("COMP MISS AT "
60       + convertIntToLetter(row) + convertCompColToRegular(col));
61       }
62       System.out.println("\nYOUR BOARD...PRESS ENTER TO CONTINUE...");
63       reader.nextLine();
64       user.playerGrid.printCombined();
65       System.out.println("PRESS ENTER TO CONTINUE...");
66       reader.nextLine();
67  }
68  private static String askForGuess(Player p, Player opp)
69  {
70   System.out.println("Viewing My Guesses:");
71   p.oppGrid.printStatus();
72   int row = -1;
73   int col = -1;
74   String oldRow = "Z";
75   int oldCol = -1;
76   while(true)
77   {
78     System.out.print("Type in row (A-J): ");
79     String userInputRow = reader.next();
80     userInputRow = userInputRow.toUpperCase();
81     oldRow = userInputRow;
82     row = convertLetterToInt(userInputRow);
83     System.out.print("Type in column (1-10): ");
84     col = reader.nextInt();
85     oldCol = col;
```

```
86       col = convertUserColToProCol(col);
87       if (col >= 0 && col <= 9 && row != -1)
88           break;
89       System.out.println("Invalid location!");
90    }
91
92       if (opp.playerGrid.hasShip(row, col))
93       {
94        p.oppGrid.markHit(row, col);
95        opp.playerGrid.markHit(row, col);
96        return "** USER HIT AT " + oldRow + oldCol + " **";
97       }
98       else
99        {
100        p.oppGrid.markMiss(row, col);
101        opp.playerGrid.markMiss(row, col);
102        return "** USER MISS AT " + oldRow + oldCol + " **";
103       }
104 }
105
106  private static void setup(Player p)
107  {
108   p.playerGrid.printShips();
109   System.out.println();
110   int counter = 1;
111   int normCounter = 0;
112   while (p.numOfShipsLeft() > 0)
113     {
114      for (Ship s: p.ships)
115      {
116          System.out.println("\nShip #" + counter +
117          ": Length-" + s.getLength());
118          int row = -1;
```

```
119        int col = -1;
120        int dir = -1;
121        while(true)
122        {
123         System.out.print("Type in row (A-J): ");
124         String userInputRow = reader.next();
125         userInputRow = userInputRow.toUpperCase();
126         row = convertLetterToInt(userInputRow);
127
128         System.out.print("Type in column (1-10): ");
129         col = reader.nextInt();
130         col = convertUserColToProCol(col);
131
132         System.out.print("Type in direction (0-H, 1-V): ");
133         dir = reader.nextInt();
134
135
136         if (col >= 0 && col <= 9 && row != -1 && dir != -1)
137         {
138             if (!hasErrors(row, col, dir, p, normCounter))
139       {
140                 break;
141             }
142         }
143
144        System.out.println("Invalid location!");
145        }
146
147        p.ships[normCounter].setLocation(row, col);
148        p.ships[normCounter].setDirection(dir);
149        p.playerGrid.addShip(p.ships[normCounter]);
150        p.playerGrid.printShips();
151        System.out.println();
```

```
152          System.out.println("You have " +
153          p.numOfShipsLeft() + " remaining ships to place.");
154
155          normCounter++;
156          counter++;
157          }
158      }
159 }
160  private static void setupComputer(Player p)
161  {
162    System.out.println();
163    int counter = 1;
164    int normCounter = 0;
165
166    Randomizer rand = new Randomizer();
167
168    while (p.numOfShipsLeft() > 0)
169    {
170        for (Ship s: p.ships)
171        {
172            int row = rand.nextInt(0, 9);
173            int col = rand.nextInt(0, 9);
174            int dir = rand.nextInt(0, 1);
175
176
177            while (hasErrorsComp(row, col, dir, p, normCounter))
178        {
179                row = rand.nextInt(0, 9);
180                col = rand.nextInt(0, 9);
181                dir = rand.nextInt(0, 1);
182            }
183            p.ships[normCounter].setLocation(row, col);
184            p.ships[normCounter].setDirection(dir);
```

```
185             p.playerGrid.addShip(p.ships[normCounter]);
186 
187             normCounter++;
188             counter++;
189         }
190     }
191 }
192 private static boolean hasErrors(int row, int col,
193     int dir, Player p, int count)
194 {
195 int length = p.ships[count].getLength();
196 if (dir == 0)
197 {
198     int checker = length + col;
199     if (checker > 10)
200     {
201         System.out.println("SHIP DOES NOT FIT");
202         return true;
203     }
204 }
205 
206 if (dir == 1)
207 {
208     int checker = length + row;
209     if (checker > 10)
210     {
211         System.out.println("SHIP DOES NOT FIT");
212         return true;
213     }
214 }
215 
216 if (dir == 0)
217 {
```

```
218  for (int i = col; i < col+length; i++)
219  {
220       if(p.playerGrid.hasShip(row, i))
221       {
222           System.out.println("THERE IS ALREADY A SHIP AT THAT LOCATION");
223           return true;
224       }
225  }
226 }
227 else if (dir == 1)
228 {
229 for (int i = row; i < row+length; i++)
230 {
231 if(p.playerGrid.hasShip(i, col))
232 {
233 System.out.println("THERE IS ALREADY A SHIP AT THAT LOCATION");
234 return true;}}
235 return false;
236 }
237 private static boolean hasErrorsComp(int row, int col, int dir,
238      Player p, int count)
239      {
240 
241          int length = p.ships[count].getLength();
242 
243          if (dir == 0)
244          {
245              int checker = length + col;
246              if (checker > 10)
247              {
248                  return true;
249              }
250          }
```

```
251
252         if (dir == 1)
253         {
254             int checker = length + row;
255             if (checker > 10)
256             {
257                 return true;
258             }
259         }
260
261         if (dir == 0)
262     {
263             for (int i = col; i < col+length; i++)
264             {
265                 if(p.playerGrid.hasShip(row, i))
266                 {
267                     return true;
268                 }
269             }
270         }
271         else if (dir == 1)
272         {
273             for (int i = row; i < row+length; i++)
274             {
275                 if(p.playerGrid.hasShip(i, col))
276                 {
277                     return true;
278                 }
279             }
280         }
281
282         return false;
283     }
```

```
284
285     private static int convertLetterToInt(String val)
286     {
287         int toReturn = -1;
288         char ch=val.charAt(0);
289         if (ch>='A'&& ch <='J') {
290           toReturn=ch-65;
291         }
292
293         return toReturn;
294     }
295     private static String convertIntToLetter(int val)
296     {
297         String toReturn = "Z";
298         char res = 'A';
299         if (val>=0 && val<10)
300         {
301          res += val;
302         }
303         toReturn = Character.toString(res);
304         return toReturn;
305     }
306
307     private static int convertUserColToProCol(int val)
308     {
309         int toReturn = -1;
310         if (val >=1 && val<=10) {
311          toReturn=val-1;
312         }
313         return toReturn;
314     }
315
316     private static int convertCompColToRegular(int val)
```

```
317      {
318          int toReturn = -1;
319          if (val >=0 && val<=9) {
320          toReturn=val+1;
321          }
322          return toReturn;
323      }
324  }
```

Grid Class

```
1  public class Grid
2  {
3  private Location[][] grid;
4  private int points;
5  public static final int NUM_ROWS = 10;
6  public static final int NUM_COLS = 10;
7
8  public Grid()
9  {
10
11     grid = new Location[NUM_ROWS][NUM_COLS];
12
13     for (int row = 0; row < grid.length; row++)
14     {
15         for (int col = 0; col < grid[row].length; col++)
16         {
17             Location tempLoc = new Location();
18             grid[row][col] = tempLoc;
19         }
20     }
21   points = 0;
```

```
22 }
23 public void markHit(int row, int col)
24 {
25     grid[row][col].markHit();
26     points++;
27 }
28
29 public void markMiss(int row, int col)
30 {
31     grid[row][col].markMiss();
32 }
33
34 public void setStatus(int row, int col, int status)
35 {
36     grid[row][col].setStatus(status);
37 }
38
39 public int getStatus(int row, int col)
40 {
41     return grid[row][col].getStatus();
42 }
43
44 public boolean alreadyGuessed(int row, int col)
45 {
46     return !grid[row][col].isUnguessed();
47 }
48
49 public void setShip(int row, int col, boolean val)
50 {
51     grid[row][col].setShip(val);
52 }
53
54 public boolean hasShip(int row, int col)
```

```
55 {
56     return grid[row][col].hasShip();
57 }
58
59 public Location get(int row, int col)
60 {
61     return grid[row][col];
62 }
63
64 public int numRows()
65 {
66     return NUM_ROWS;
67 }
68 public int numCols()
69 {
70     return NUM_COLS;
71 }
72
73 public void printStatus()
74 {
75     generalPrintMethod(0);
76 }
77
78 public void printShips()
79 {
80     generalPrintMethod(1);
81 }
82
83 public void printCombined()
84 {
85     generalPrintMethod(2);
86 }
87 public boolean hasLost()
```

```
88 {
89      if (points >= 17)
90          return true;
91      else
92          return false;
93 }
94
95 public void addShip(Ship s)
96 {
97      int row = s.getRow();
98      int col = s.getCol();
99      int length = s.getLength();
100     int dir = s.getDirection();
101
102 if (!(s.isDirectionSet()) || !(s.isLocationSet()))
103 throw new IllegalArgumentException
104 ("ERROR! Direction or Location is unset/default");
105     if (dir == 0)
106     {
107     for (int i = col; i < col+length; i++)
108     {
109         grid[row][i].setShip(true);
110         grid[row][i].setLengthOfShip(length);
111         grid[row][i].setDirectionOfShip(dir);
112     }
113 }
114     else if (dir == 1)
115     {
116         for (int i = row; i < row+length; i++)
117         {
118             grid[i][col].setShip(true);
119             grid[i][col].setLengthOfShip(length);
120             grid[i][col].setDirectionOfShip(dir);
```

```
121            }
122        }
123        }
124 public int switchCounterToIntegerForArray (int val)
125 {
126        int toReturn = -1;
127        if(val >= 65 && val <= 90){
128            toReturn=(val-65);
129        }
130        if (toReturn == -1)
131        {
132         throw new IllegalArgumentException("ERROR OCCURED IN SWITCH");
133        }
134            return toReturn;
135 }
136 }
```

Location class

```
1 public class Location
2 {
3      public static final int UNGUESSED = 0;
4      public static final int HIT = 1;
5      public static final int MISSED = 2;
6      private boolean hasShip;
7      private int status;
8      private int lengthOfShip;
9      private int directionOfShip;
10
11     public Location()
12     {
13         status = 0;
```

```
14          hasShip = false;
15          lengthOfShip = -1;
16          directionOfShip = -1;
17      }
18      public boolean checkHit()
19      {
20          if (status == HIT)
21              return true;
22          else
23              return false;
24      }
25      public boolean checkMiss()
26      {
27          if (status == MISSED)
28              return true;
29          else
30              return false;
31      }
32      public boolean isUnguessed()
33      {
34          if (status == UNGUESSED)
35              return true;
36          else
37              return false;
38      }
39      public void markHit()
40      {
41          setStatus(HIT);
42      }
43
44      public void markMiss()
45      {
46          setStatus(MISSED);
```

```
47      }
48
49      public boolean hasShip()
50      {
51          return hasShip;
52      }
53
54      public void setShip(boolean val)
55      {
56          this.hasShip = val;
57      }
58      public void setStatus(int status)
59      {
60          this.status = status;
61      }
62      public int getStatus()
63      {
64          return status;
65      }
66      public int getLengthOfShip()
67      {
68          return lengthOfShip;
69      }
70
71      public void setLengthOfShip(int val)
72      {
73          lengthOfShip = val;
74      }
75
76      public int getDirectionOfShip()
77      {
78          return directionOfShip;
79      }
```

```
80
81      public void setDirectionOfShip(int val)
82      {
83          directionOfShip = val;
84      }
85  }
```

Player class

```
1   public class Player
2   {
3    private static final int[] SHIP_LENGTHS = {2, 3, 3, 4, 5};
4    private static final int NUM_OF_SHIPS = 5;
5
6    public Ship[] ships;
7    public Grid playerGrid;
8    public Grid oppGrid;
9
10   public Player()
11   {
12       ships = new Ship[NUM_OF_SHIPS];
13       for (int i = 0; i < NUM_OF_SHIPS; i++)
14       {
15           Ship tempShip = new Ship(SHIP_LENGTHS[i]);
16           ships[i] = tempShip;
17       }
18       playerGrid = new Grid();
19       oppGrid = new Grid();
20   }
21   public void addShips()
22   {
23       for (Ship s: ships)
```

```
24          {
25              playerGrid.addShip(s);
26          }
27      }
28      public int numOfShipsLeft()
29      {
30          int counter = 5;
31          for (Ship s: ships)
32          {
33              if (s.isLocationSet() && s.isDirectionSet())
34                  counter--;
35          }
36          return counter;
37      }
38  public void chooseShipLocation(Ship s, int row, int col, int direction)
39      {
40          s.setLocation(row, col);
41          s.setDirection(direction);
42          playerGrid.addShip(s);
43      }
44  }
```

Randomizer class

```
1   import java.util.*;
2
3   public class Randomizer
4   {
5
6       public static Random theInstance = null;
7
8       public Randomizer(){
```

```
9
10 }
11
12 public static Random getInstance(){
13  if(theInstance == null){
14   theInstance = new Random();
15  }
16  return theInstance;
17 }
18
19 public static boolean nextBoolean(){
20  return Randomizer.getInstance().nextBoolean();
21 }
22
23 public static boolean nextBoolean(double probability){
24  return Randomizer.nextDouble() < probability;
25 }
26
27 public static int nextInt(){
28  return Randomizer.getInstance().nextInt();
29 }
30
31 public static int nextInt(int n){
32  return Randomizer.getInstance().nextInt(n);
33 }
34
35 public static int nextInt(int min, int max){
36  return min + Randomizer.nextInt(max - min + 1);
37 }
38
39 public static double nextDouble(){
40  return Randomizer.getInstance().nextDouble();
41 }
```

```
42
43  public static double nextDouble(double min, double max){
44   return min + (max - min) * Randomizer.nextDouble();
45  }
46
47
48 }
```

Ship class

```
1 public class Ship
2 {
3     private int row;
4     private int col;
5     private int length;
6     private int direction;
7
8     public static final int UNSET = -1;
9     public static final int HORIZONTAL = 0;
10    public static final int VERTICAL = 1;
11
12    public Ship(int length)
13    {
14        this.length = length;
15        this.row = -1;
16        this.col = -1;
17        this.direction = UNSET;
18    }
19
20    public boolean isLocationSet()
21    {
22        if (row == -1 || col == -1)
```

```
23              return false;
24          else
25              return true;
26      }
27
28      public boolean isDirectionSet()
29      {
30          if (direction == UNSET)
31              return false;
32          else
33              return true;
34      }
35
36      public void setLocation(int row, int col)
37      {
38          this.row = row;
39          this.col = col;
40      }
41
42      public void setDirection(int direction)
43      {
44      if (direction != UNSET && direction
45      != HORIZONTAL && direction != VERTICAL)
46      throw new IllegalArgumentException
47      ("Invalid direction.It must be -1, 0, or 1");
48      this.direction = direction;
49      }
50
51      public int getRow()
52      {
53          return row;
54      }
55
```

```
56      public int getCol()
57      {
58          return col;
59      }
60
61      public int getLength()
62      {
63          return length;
64      }
65
66      public int getDirection()
67      {
68          return direction;
69      }
70
71      private String directionToString()
72      {
73          if (direction == UNSET)
74              return "UNSET";
75          else if (direction == HORIZONTAL)
76              return "HORIZONTAL";
77          else
78              return "VERTICAL";
79      }
80
81      public String toString()
82      {
83          return "Ship: " + getRow() + "," + getCol() + " with length "
84          + getLength() + "and direction" + directionToString();
85      }
86  }
```

Appendix C

Questions

1-Which classes don't need the Ship class to compile?

2- Name all the classes in the program.

3-Describe the Location and the Battleship classes. What are the goals of each class?

4- Does the player get an extra turn when guessing the right location?

5- A- Briefly describe the purpose of the following code:

```
if (dir == 0)
{
int checker = length + col;
if (checker > 10)
{
return true;
}}
```

5-B-Briefly describe the purpose of the following code:

```
while(p.numOfShipsLeft() > 0)
{
for (Ship s: p.ships)
{
  int row = rand.nextInt(0, 9);
```

```
6   int col = rand.nextInt(0, 9);
7   int dir = rand.nextInt(0, 1);
8 while (hasErrorsComp(row, col, dir, p, normCounter))
9 {
10    row = rand.nextInt(0, 9);
11    col = rand.nextInt(0, 9);
12    dir = rand.nextInt(0, 1);
13    p.ships[normCounter].setLocation(row, col);
14    p.ships[normCounter].setDirection(dir);
15    p.playerGrid.addShip(p.ships[normCounter]);
16    normCounter++;
17    counter++;
18   }
19  }
20 }
```

6-a- Describe what the following method is doing:

```
1 private static int X(String val)
2 {
3 int toReturn = -1;
4 char ch=val.charAt(0);
5 if (ch>='A'&& ch <='J')
6  {
7   toReturn=ch-"A";
8  }
9 return toReturn;
10 }
```

6-b-Describe what the following code is doing:

```
1 for(int row = 0; row < grid.length; row++)
```

```
{
 for(int col = 0; col < grid[row].length; col++)
 {
   Location tempLoc = new Location();
   grid[row][col] = tempLoc;
 }
}
```

7-How is a "user hit" determined in the Battleship class? Which methods in which class are called to record a hit.

8- Write the names of the Grid instances in the Player class.

9- Match the methods to the appropriate class.

10- Match each instance variable to the class, where it is declared.

11- Where is the exact location of the toString method in the Ship class?

Demographic questions

Age

Gender

Education Level

Field of study

Are you a student right now? The year your program started:

Have you been in a COOP program? How many terms and what was your position? If it is for more than one provide details for all.

How many programming languages do you know? Name all of them.

How many years of object-oriented programming do you have?

Do you see yourself as an expert or novice in programming?

What was the first language you used for programming?

Are you currently working as a developer?

Do you have experience programming in any procedural language?

Do you want to receive the results of the experiment by email?

Appendix D

R Scripts

R script for fixed effect model and objective expertise

```
1 glmer(acc ~ as.factor(Group)* as.factor(Question) + (1|ID), Data4,
    family=binomial,  control =glmerControl(optimizer="bobyqa"))
```

R script for random effect model and objective expertise

```
1 glmer(acc ~ as.factor(Group)* as.factor(Question) + (1+as.factor(
    Question)|ID),Data4, family=binomial, control =glmerControl(
    optimizer="bobyqa"))
```

R script for fixed effect model and subjective expertise

```
1 glmer(acc ~ as.factor(expertise)* as.factor(Question) + (1|ID), Data4,
    family=binomial, control =glmerControl(optimizer="bobyqa"))
```

R script for random effect model and subjective expertise

```
1 glmer(acc ~ as.factor(expertise)* as.factor(Question) + (1+as.factor(
    Question)|ID), Data4, family=binomial, control=glmerControl(
    optimizer="bobyqa"))
```

R script for interaction test

```
1 testInteractions(accall2, fixed="as.factor(Question)", across="as.
    factor(Group)", adjustment="none")
```

R script for summary of data

```
1 summarySE(Data4, measurevar="acc", groupvars=c("Question","Group"))
```

Figure D.1: R scripts used for analysis

Vita

Candidate's full name: Mohammadhossein Parastar

University attended:

University of New Brunswick – Fredericton, NB, CA
2018 – 2021
Masters of Computer Science

Islamic Azad University of Saveh - Saveh, MA, Iran
2014-2017
Bachelor of Software Engineering